CompSc 1

20 Sep 2021

Course description:

- **Syllabus (Programming) Recommended Language: C** Basic abilities of writing, executing, and debugging programs. Basics: Conditional statements, loops, block structure, functions and parameter passing, single and multi-dimensional arrays, structures, pointers. Data Structures: stacks, queues, linked lists, binary trees. Simple algorithmic problems: Some simple illustrative examples, parsing of arithmetic expressions, matrix operations, searching and sorting algorithms.
- Depending on how the class goes, we may or may not cover all the above.
- This is the old syllabus; look up the new syllabus on the website; most of it is the same as above.

Text Book

- The **C Programming Language** | Second Edition | By Pearson. by Brian W. Kernighan / Dennis Ritchie.
- You can try to get pdf online. E.g.
- http://cslabcms.nju.edu.cn/problem_solving/images/c/cc/The_C_Programming_Language_%282nd_Edition_Ritchie_Kernighan%29.pdf

- Many other resources available online. Will specify some as we progress.
- http://www.marcusramos.com.br/univasf/pc-2008-2/cb-sp-dahl.pdf - Structured Programming
- https://seriouscomputerist.atariverse.com/media/pdf/book/Science%20of%20Programming.pdf – Science of Programming.

Basic structure of a Computer Program

- What does a computer program do?
- Takes input, processes input, generates output.
- The entire process is written as “code”. The code specifies how to accept inputs (from keyboard or from files) how to process and how to generate output.
- A language to code like C is written in a English like language with its own grammar.
- The code has to be “compiled”, “linked” etc. before it is run or executed. – Will not discuss this process in detail at this time.
- So we need a compiler to run and execute C code.

Structure of a C Program via Examples

- Suppose you want to use C to write the following sentence on your screen " I am feeling great today", what would you need to do?
- Obviously you need to know what the command in C that prints something on screen. For reasons that will become obvious we will call most of these commands as **functions**.
- Obviously you will also need to know how to put that command/function in a few lines so as to obey the grammar of C so that it can be compiled and executed to print that line on your screen .
- By the way why do I keep underlining "on your screen"?

- Lets assume the function that writes on the screen has a name. It is called “printf” in C. Suppose we write something like

  printf “I am feeling great today”

and try to execute this.

- That will not work. The correct piece of code that will work is

```
#include <stdio.h>
main ()

{
printf(“I am feeling great today”);
}
```

- We will need to “compile and run the code” next. Before that discussion on the code.

- Why do we need the first line?
  - How does the program compiler, and execution know what “printf” is?
  - A compiler will, among other things, look for this program in available libraries that come with the C programming language.
  - You have to tell the compiler where, which library, to look for printf
  - stdio.h is such a library. #stdio.h tells the compiler to include all the functions available in that “header file”
- What is main?
  - main is a function that EVERY C program must have and will start with.
    - Within main there can be other functions.
- Let us quote from the text book here.

Now for some explanations about the program itself. A C program, whatever its size, consists of *functions* and *variables*. A function contains *statements* that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions of Fortran or the procedures and functions of Pascal. Our example is a function named `main`. Normally you are at liberty to give functions whatever names you like, but "`main`" is special—your program begins executing at the beginning of `main`. This means that every program must have a `main` somewhere.

`main` will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program,

```
#include <stdio.h>
```

tells the compiler to include information about the standard input/output library; this line appears at the beginning of many C source files. The standard library is described in Chapter 7 and Appendix B.

How to Compile

- You need to (you must for this course) install a C compiler on your computer. There are many choices. What I recommend here for those who have a Windows machine is to please download “code blocks” from

  http://www.codeblocks.org/downloads

- UNLESS YOU ARE VERY SOPHISTICATED, download the binary from the above site.
- For quick purposes you can compile and run small programs from the cloud such as https://

  www.onlinegdb.com/online_c_compiler

We will show you both.

Homework- 22Sep2021
(you need to do but not submit)

- Get a copy of the text book and read just upto and including section 1.1
- Download and install codeblocks and run the program discussed in the class.

Code::Blocks

Code::Blocks - The IDE with all the features you need, having a consistent look, feel and operation across platforms.

Main

- Home
- Features
- Screenshots
- Downloads
  - Binaries
  - Source
  - SVN
- Plugins
- User manual
- Licensing
- Donations

Quick links

- FAQ
- Wiki
- Forums
- Forums (mobile)
- Nightlies
- Ticket System
- Browse SVN
- Browse SVN log

Downloads

There are different ways to download and install Code::Blocks on your computer:

- **Download the binary release**

  This is the easy way for installing Code::Blocks. Download the setup file, run it on your computer and Code::Blocks will be installed, ready for you to work with it. Can't get a that!

  - **Download a nightly build**: There are also more recent so-called *nightly builds* available in the **forums**. Please note that we consider nightly builds to be *stable*, usua stated otherwise.
  - Other distributions usually follow provided by the **community** (big "Thank you!" for that!). If you want to provide some, make sure to announce in the forums such tha on the official C::B homepage.

- **Download the source code**

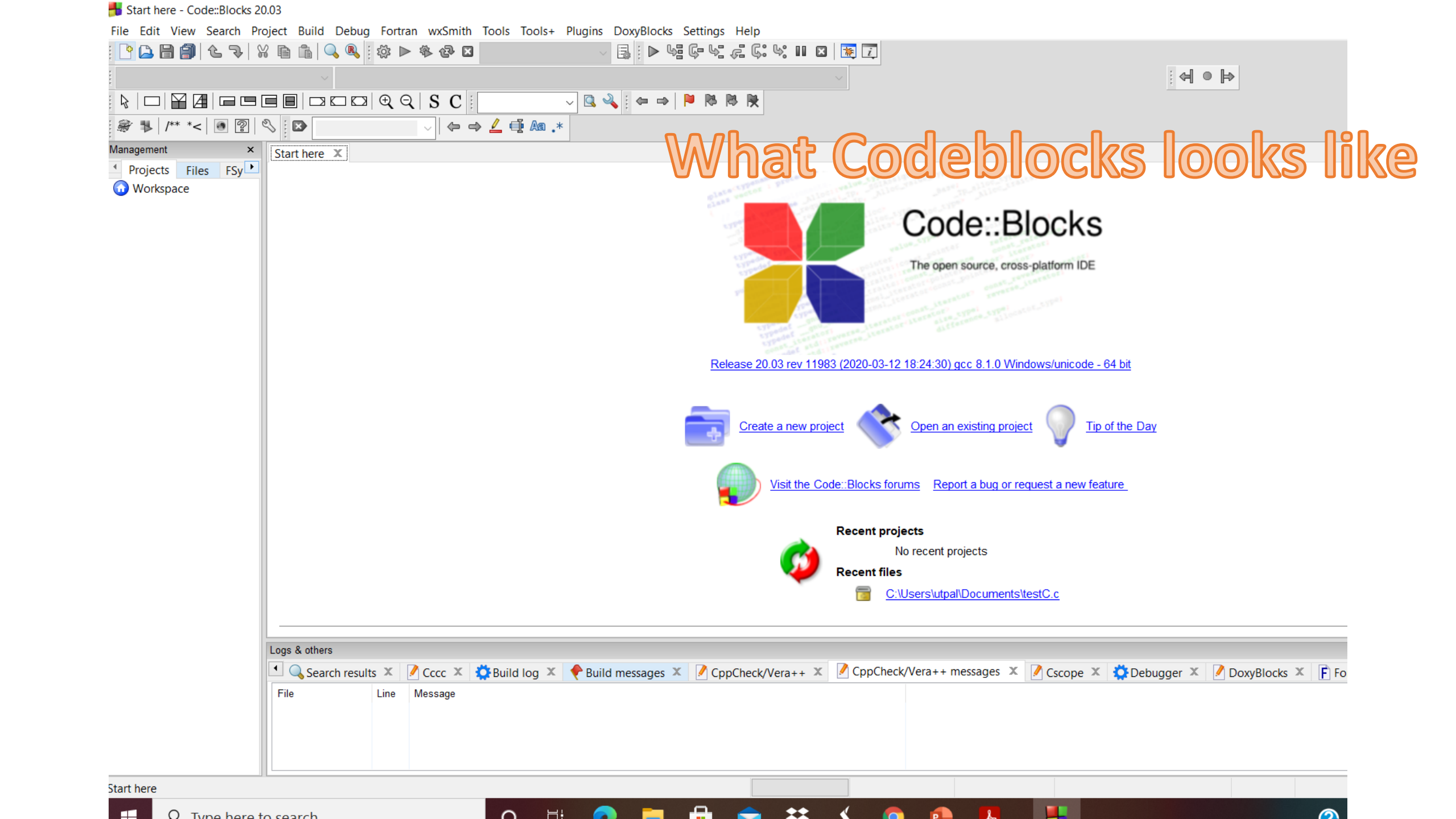
  If you feel comfortable building applications from source, then this is the recommend way to download Code::Blocks. Downloading the source code and building it yourself p control and also makes it easier for you to update to newer versions or, even better, create patches for bugs you may find and contributing them back to the community so e benefits.

- **Retrieve source code from SVN**

  This option is the most flexible of all but requires a little bit more work to setup. It gives you that much more flexibility though because you get access to any bug-fixing we d do it. No need to wait for the next stable release to benefit from bug-fixes!

Besides Code::Blocks itself, you can compile extra plugins from contributors to extend its functionality.

Thank you for your interest in downloading Code::Blocks!


What Codeblocks looks like
Start here - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Files FSy
Workspace
Start here
Code::Blocks
The open source, cross-platform IDE
Release 20.03 rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit
Create a new project
Open an existing project
Tip of the Day
Visit the Code::Blocks forums
Report a bug or request a new feature
Recent projects
No recent projects
Recent files
C:\Users\utpal\Documents\testC.c
Logs & others
Search results
Cccc
Build log
Build messages
CppCheck/Vera++
CppCheck/Vera++ messages
Cscope
Debugger
DoxyBlocks
File
Line
Message
Start here
Type here to search

Evaluations

- In Class Quizzes (typically once in two /three weeks, the exact date to be communicated in the previous classes. Look for Moodle notifications.
- We will do 7 -10 quizzes. We will discard the lowest two marks.
- All quizzes will have similar portion but may not have the exact same total marks. We will normalize each quiz to 20 marks when we compare quizzes to decide which to discard.
- 50% from the Quizzes; Rest 50% in the final exam.
- You can contact me in utpal@isibang.ac.in contacting me over moodle is preferred.

Today we will focus on

- Functions
- Variables
- Data Types
- Naming variables
- Keywords in C

Functions in C

- We have already seen two functions
  - main()
  - printf(“”)

- Generic structure of a function in C

  (function return type) name_of_function (argument1, argument2)

  {

  statement 1;

  statement 2;

  }

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

include information about standard library

define a function named `main` *that receives no argument values*

statements of `main` *are enclosed in braces*

`main` *calls library function* `printf` *to print this sequence of characters;* `\n` *represents the newline character*

The first C program.

example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list `()`.

The statements of a function are enclosed in braces `{}`. The function `main` contains only one statement,

```
printf("hello, world\n");
```

The statements of a function are enclosed in braces `{}`. The function `main` contains only one statement,

```
printf("hello, world\n");
```

A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n"`. `printf` is a library function that prints output, in this case the string of characters between the quotes.

A sequence of characters in double quotes, like `"hello, world\n"`, is called a *character string* or *string constant*. For the moment our only use of character strings will be as arguments for `printf` and other functions.

Variables in C

- What is a variable?
- Suppose you want to compute factorial of a number. Say 100! You must tell the computer that you will provide (input) the program with an integer whose value is 100. And then instruct the code what to do with that input. The computer must provide for a space in the memory to keep that integer which it must be able to refer to by a name. This is done by declaring a variable by specifying its type and its name. e.g. in the above example int apx; is a declaration that the variable whose name is apx will be an integer. Its data type is int

Variable

- A **variable** is nothing but a name given to a storage area that our programs can manipulate. Each **variable in C** has a specific type, which determines the size and layout of the **variable's** memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the **variable**.
- **Variable** is the name of memory location. Unlike constant, **variables** are changeable, we can change value of a **variable** during execution of a program. A programmer can choose a meaningful **variable** name. **Example** : average, height, age, total etc.

Basic Data types in C

K&R Chapter 2.

2.2 Data Types and Sizes

There are only a few basic data types in C:

`char`	a single byte, capable of holding one character in the local character set.
`int`	an integer, typically reflecting the natural size of integers on the host machine.
`float`	single-precision floating point.
`double`	double-precision floating point.

In addition, there are a number of qualifiers that can be applied to these basic types. `short` and `long` apply to integers:

```
short int sh;
long int counter;
```

The word `int` can be omitted in such declarations, and typically is.

Variable Names (From two sources on internet; but check your latest C reference book)

Rules to construct a valid variable name:

1. A variable name may consists of letters, digits and the underscore (_) characters.
2. A variable name must begin with a letter. Some system allows to starts the variable name with an underscore as the first character.
3. ANSI standard recognizes a length of 31 characters for a variable name. However, the length should not be normally more than any combination of eight alphabets, digits, and underscores.
4. Uppercase and lowercase are significant. That is the variable **Totamt** is not the same as **totamt** and **TOTAMT.**
5. The variable name may not be a C reserved word (keyword).

Rules for naming a variable

1.A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2.The first letter of a variable should be either a letter or an underscore.
3.There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Keywords in C

C Keywords			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

Write a program to add two numbers

- In the following program, three variables of int type declared and assigned values.
- The = sign is not the same as in math, it is an assignment operator
- Notice printf

```
#include <stdio.h>

int main()
        {
        int x, y;
         x= 1;
        y=2;
        int sum;
        sum =  x + y;
        printf("%d", sum);
        return 0;
        }
```

Write a program to add two numbers

- Same program using a function.

int add : name of the function.
(int i, int j)
{
}

```
#include <stdio.h>

int add(int i, int j)
{   int sum;
    sum= i+j;
    return sum;
}

int main()
        {
        int x, y;
         x= 1;
        y=2;
        int sum;
        sum = add(x,y);
        printf("%d", sum);
        return 0;
        }
```

declaration of a function

4 Oct 2020

- Quiz on 6 Oct 2021; during class hours.
- Look for notification in Moodle

Today we will

- Discuss more library functions for standard io
- Introduce conditional execution of statements

A property of C functions – worth repeating again and again

- A C function returns a value belonging to a datatype.
- If it returns an integer, then the datatype of the function will be int
- If it does not return a value then its type is void.
- The printf function also returns a value as we will see later.
- Two more functions we will introduce today getchar and putchar, both are integer type.

How to accept input from the standard input

- getchar(), and scanf() functions
- The C library function **int getchar()** gets a character, one character at a time (an unsigned char), from stdin.

  - The function returns the integer representing the character
- int scanf() function is used to read character, string, numeric data from keyboard.
  - The function returns the number of items of the argument list successfully read. If a reading error happens or the end-of-file is reached while reading, the proper indicator is set (feof or ferror) and, if either happens before any data could be successfully read, EOF is returned.

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
int c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
int c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

The getchar() function obtains a character from stdin. It returns **the character that was read in the form of an integer or EOF if an error occurs**.

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
char c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

This will still work. Why?

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
char c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

Description

The C library function **int getchar(void)** gets a character (an unsigned char) from stdin. This is equivalent to **getc** with stdin as its argument.

Declaration

Following is the declaration for getchar() function.

```c
int getchar(void)
```

Parameters

- **NA**

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
int c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

```c
#include <stdio.h>
int main ()
{
char any;
printf("Enter a character: ");
scanf ("%c" ,&any);
printf("Character entered: %c",any);
return(0);

}
```

Simple Examples using getchar, scanf

```c
#include <stdio.h>
int main ()
{
int c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);
return(0);

}
```

```c
#include <stdio.h>
int main ()
{
char any;
printf("Enter a character: ");
scanf ("%c" ,&any);
printf("Character entered: %ch",any);
return(0);

}
```

What does &any mean in the above program?
The variable name is any. &any represents the memory location of the variable any (declared as char) where scanf stores the entered character.

Putchar

- Opposite of getchar is putchar.
- putchar accepts an integer and returns the character the integer represents.

```c
#include <stdio.h>
int main ()
{
int c;
printf("Enter character: ");
c = getchar();
printf("Integer representing Character entered: %d",c);

putchar(c);
return(0);
}
```

- What will be the output if you enter A ?

Integer representing Character A is 65
A

Putting these together: output an input file

read a character
while (*character is not end-of-file indicator*)
    output the character just read
    read a character

Converting this into C gives

```
#include <stdio.h>

/* copy input to output; 1st version */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
```

More on printf (formatted output)

Program

```
#include <stdio.h>
int main()
{
  char ch = 'A';
  char str[20] = “LearningC";
  float pi_upto2digits = 3.14;
  int no = 150;
  double dbl = 20.123456;
  printf("Character is %c \n", ch);
  printf("String is %s \n" , str);
  printf("Float value is %f \n", flt);
  printf("Integer value is %d\n" , no);
  printf("Double value is %lf \n", dbl);
  printf("Octal value is %o \n", no);
  printf("Hexadecimal value is %x \n", no);
  return 0;
}
```

More on printf (formatted output)

Program

```
#include <stdio.h>
int main()
{
  char ch = 'A';
  char str[20] = “LearningC";
  float pi_upto2digits = 3.14;
  int no = 150;
  double dbl = 20.123456;
  printf("Character is %c \n", ch);
  printf("String is %s \n" , str);
  printf("Float value is %f \n", flt);
  printf("Integer value is %d\n" , no);
  printf("Double value is %lf \n", dbl);
  printf("Octal value is %o \n", no);
  printf("Hexadecimal value is %x \n", no);
  return 0;
}
```

Output

```
Character is A
String is LearningC
Float value is 3.14
Integer value is 150
Double value is 20.123456
Octal value is 226
Hexadecimal value is 9
```

More on printf (formatted output)

Program

```
#include <stdio.h>
int main()
{
  char ch = 'A';
  char str[20] = “LearningC";
  float pi_upto2digits = 3.14;
  int no = 150;
  double dbl = 20.123456;
  printf("Character is %c \n", ch);
  printf("String is %s \n" , str);
  printf("Float value is %f \n", flt);
  printf("Integer value is %d\n" , no);
  printf("Double value is %lf \n", dbl);
  printf("Octal value is %o \n", no);
  printf("Hexadecimal value is %x \n", no);
  return 0;
}
```

Output

```
Character is A
String is LearningC
Float value is 3.14
Integer value is 150
Double value is 20.123456
Octal value is 226
Hexadecimal value is 9
```

Note the following:

1. Need \n so that the outputs are not in different lines.
2. What will the following statement print?

   printf(“Integer value of %5d\n”,no);

3. printf replaces values of the variables in the order they appear. What will the following statement print?

   printf(“Integer value of %d and Octal value is %o\n”,no,no);

4. AFIK printf does not support printing binary representation of an integer. Investigate how you can write a function to do that.

More on printf

- The function returns an integer. The return value of printf is the total number of characters it printed.
- What will be the output of the following code? Try to answer this without running the code.

```
#include <stdio.h>
int main(){
char str[] = "I am a student at ISI";
printf(“\nThe value returned by printf() for the above string is : %d", printf("%s", str));
return 0;
}
```

More on printf

- The function returns an integer. The return value of printf is the total number of characters it printed.
- What will be the output of the following code? Try to answer this without running the code.

```
#include <stdio.h>
int main(){
char str[] = "I am a student at ISI";
printf(“\nThe value returned by printf() for the above string is : %d", printf("%s", str));
Return 0;
}
```

I am a student at ISI

The value returned by printf() for the above string is : 21

Conditional execution of statement

- While, for, if are used to control execution of statements, either to repeat them until a condition is fulfilled or to execute the statement at all

- Loops in C
- Conditional execution of statements
- Constants in C

while and do while

```
while (condition)
statement;
```

OR

```
while (condition)
{
statement 1;
statement2;
}
```

DO..WHILE –
DO..WHILE loops are useful for things that want to loop at least once. The structure is

```
do {
} while ( condition );
```

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once.

If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once.
A while loop says "Loop while the condition is true, and execute this block of code", a do..while loop says "Execute this block of code, and then continue to loop while the condition is true".

for

```
for ( variable initialization; condition; variable update ) {
  statements to  execute while the condition is true
}
// variable updated after each loop BEFORE checking condition
```

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

If else, else if

if (condition)

statement1; //statement1 will be executed only if condition is true

statement 2; (will be executed whether condition is true or not??)

OR

if(condition)

statement1; //statement1 will be executed only if condition is true

else statement2; //statement2 will be executed only if condition is false

statement3; // will be executed whether condition is true of false

Else-if

3.3 Else-If

The construction

```
if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement
```

occurs so often that it is worth a brief separate discussion. This sequence of `if` statements is the most general way of writing a multi-way decision. The *expressions* are evaluated in order; if any *expression* is true, the *statement* associated with it is executed, and this terminates the whole chain. As always, the code for each *statement* is either a single statement, or a group in braces.

The last `else` part handles the "none of the above" or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing

```
else
    statement
```

can be omitted, or it may be used for error checking to catch an "impossible" condition.

A simple example of else if

```
#include<stdio.h>
int main() {
int marks=83;
if(marks>75)
{ printf("First class"); }

else if(marks>65)
{ printf("Second class"); }

else if(marks>55)
{ printf("Third class"); }

else{ printf("Fourth class"); }
return 0; }
```

11Oct2021

Using CONSTANTS for better coding

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

1.4 Symbolic Constants

A final observation before we leave temperature conversion forever. It's bad practice to bury "magic numbers" like 300 and 20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to give them meaningful names. A `#define` line defines a *symbolic name* or *symbolic constant* to be a particular string of characters:

`#define` *name replacement text*

Using Symbolic Constants

```
#include <stdio.h>

#define   LOWER  0       /* lower limit of table */
#define   UPPER  300     /* upper limit */
#define   STEP   20      /* step size */

/* print Fahrenheit-Celsius table */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Character Constants

What is the difference between

putchar(65);

and

putchar('A');

Character Constants

What is the difference between

putchar(65);

and

putchar('A');

BOTH WILL (likely) write A on the standard output.

Character Constants

What is the difference between

putchar(65);

and

putchar('A');

BOTH WILL (likely) write A on the standard output.

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a *character constant*, although it is just another way to write a small integer. So, for example, 'A' is a character constant; in the ASCII character set its value is 65, the internal representation of the character A. Of course 'A' is to be preferred over 65: its meaning is obvious, and it is independent of a particular character set.

- In the definition below what is the data type of ‘E ?’
- #define E 2.718
- Is it float, integer, or will it generate an error?

- Write a C Program to change cases of alphabets in input using all that we have discussed.

e.g. the input

I see a RED bull, only ONE (1)

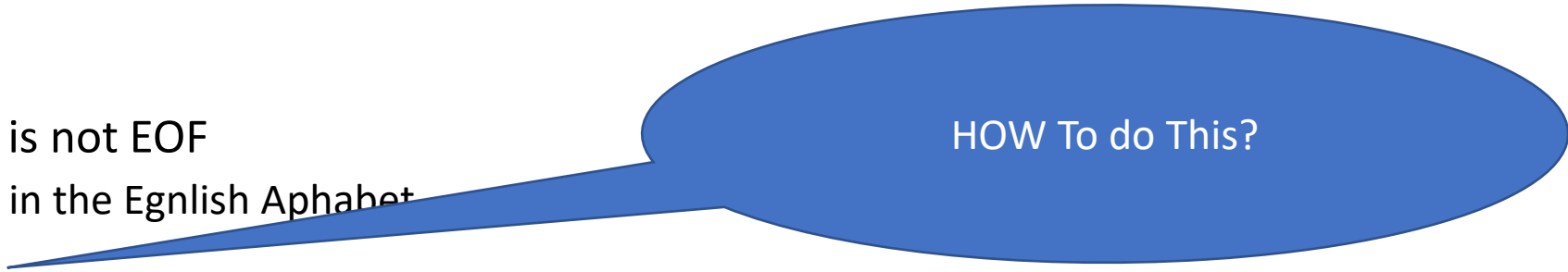
will become

i SEE A red BULL,ONLY one(1)

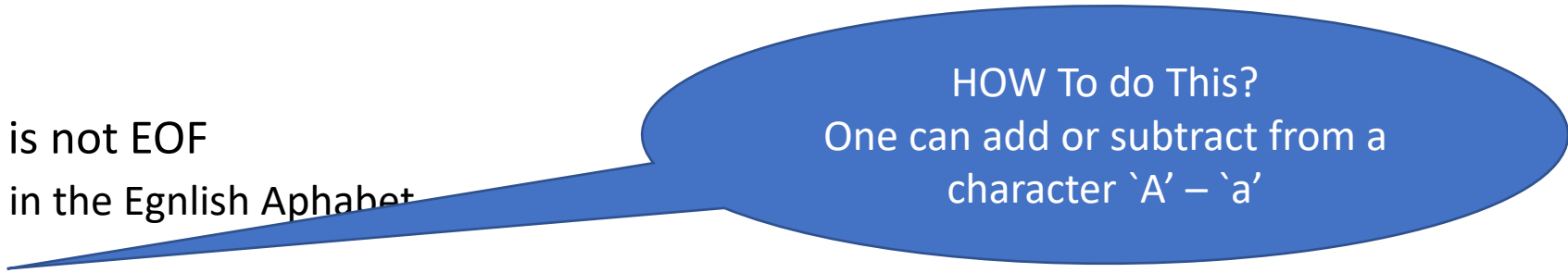
(Assume that all the upper case (lower case too) letters are represented by contiguous letters)

- Psuedo code (very important to write something like this before starting to code)
  - Get a character
    - While the character is not EOF
      - If it is a character in the Egnlish Aphabet,
        - change case
        - Write to stdio or a file whatever
      - Else if read next character
    - Continue with While
  - End while

- Psuedo code (very important to write something like this before starting to code)
  - Get a character
    - While the character is not EOF
      - If it is a character in the Egnlish Aphabet
        - change case
        - Write to stdio or a file whatever
      - Else if read next character
    - Continue with While
  - End while

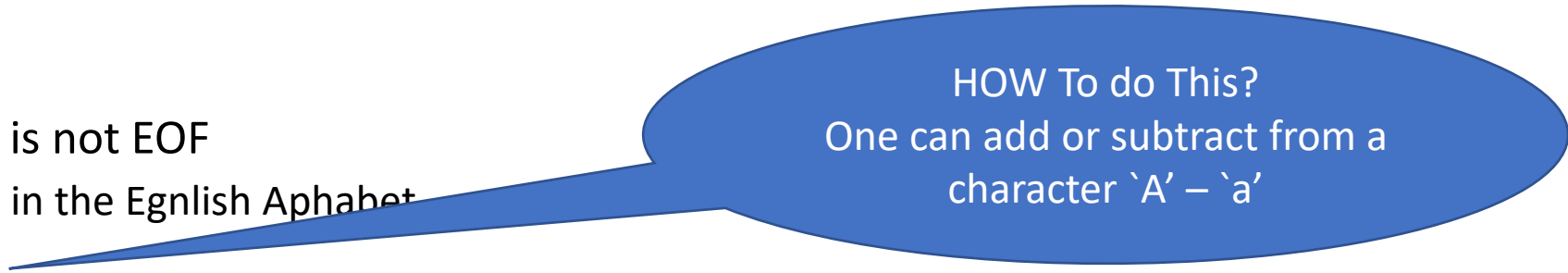


- Psuedo code (very important to write something like this before starting to code)
  - Get a character
    - While the character is not EOF
      - If it is a character in the Egnlish Aphabet
        - change case
        - Write to stdio or a file whatever
      - Else if read next character
    - Continue with While
  - End while



Write the complete C program now.

- Psuedo code (very important to write something like this before starting to code)
  - Get a character
    - While the character is not EOF
      - If it is a character in the Egnlish Aphabet
        - change case
        - Write to stdio or a file whatever
      - Read next character
    - Continue with While
  - End while



A Program to Change cases of alphabets in input

```
#include <stdio.h>
int main(){
int nxtlet;
nxtlet = getchar();

while (nxtlet !=EOF)
{
if ((nxtlet>='A') && (nxtlet<='Z'))
putchar(nxtlet +'a'-'A');

        else if((nxtlet>='a') && (nxtlet<='z'))
        putchar(nxtlet -'a'+'A');

        else putchar(nxtlet);

nxtlet= getchar();
}
return 0;
}
```

Be careful about for syntax

Look at the for statement carefully

The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

What is an expression in C?

Grammatically, the three components of a `for` loop are expressions. Most commonly, $expr_1$ and $expr_3$ are assignments or function calls and $expr_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If $expr_1$ or $expr_3$ is omitted, it is simply dropped from the expansion. If the test, $expr_2$, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an "infinite" loop, presumably to be broken by other means, such as a `break` or `return`.

Look at the for statement carefuly

The `for` statement

`for` ($expr_1$; $expr_2$; $expr_3$)
statement

What is an expression in C?

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b - c	a * b - c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Look at the for statement carefuly

The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

What is an expression in C?

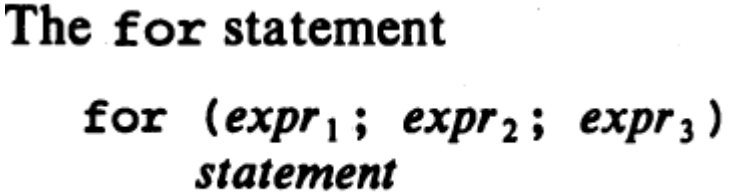
Grammatically, the three components of a `for` loop are expressions. Most commonly, $expr_1$ and $expr_3$ are assignments or function calls and $expr_2$ is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If $expr_1$ or $expr_3$ is omitted, it is simply dropped from the expansion. If the test, $expr_2$, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an "infinite" loop, presumably to be broken by other means, such as a `break` or `return`.

For and While loops are equivalent

The for statement



The `for` statement

`for` (*expr*$_1$; *expr*$_2$; *expr*$_3$)
statement

is equivalent to

……;

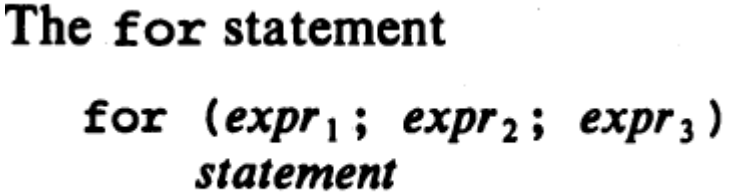
while …

….;

….;

For and While loops are equivalent

The for statement



The `for` statement

`for` (*expr*$_1$; *expr*$_2$; *expr*$_3$)
statement

is equivalent to

……;

while …

….;

….;

expr1;
While (expr2)
statement;
expr3;

For and While loops are equivalent

The for statement

The `for` statement

`for` (*expr*$_1$; *expr*$_2$; *expr*$_3$)
*statement***

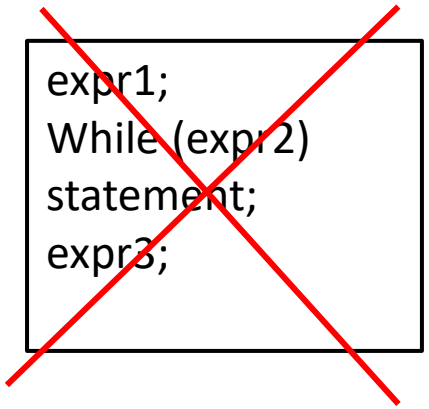
is equivalent to

……;

while …

….;

….;



For and While loops are equivalent

The for statement

The `for` statement

```
for (expr1 ; expr2 ; expr3 )
     statement
```

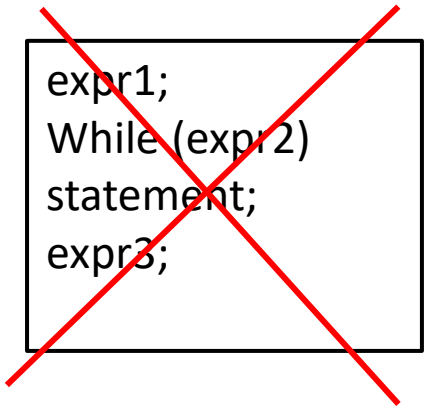
is equivalent to

……;

while …

….;

….;




```
expr1 ;
while (expr2 ) {
     statement
     expr3 ;
}
```

Program to count characters in input

Declare and initialize to zero an int variable which will count number of characters

Every time a new character is read, count goes up UNTIL EOF

Print the value of the variable

Program to count characters in input

Declare and initialize to zero an int variable which will count number of characters

Every time a new character is read, count goes up UNTIL EOF

Print the value of the variable

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
    printf("%.0f\n", nc);
}
```

What will this print?

Program to count characters in input

Declare and initialize to zero an int variable which will count number of characters

Every time a new character is read, count goes up UNTIL EOF

Print the value of the variable

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
    double nc;

    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

The line

```
nl = nw = nc = 0;
```

sets all three variables to zero. This is not a special case, but a consequence of the fact that an assignment is an expression with a value and assignments associate from right to left. It's as if we had written

```
nl = (nw = (nc = 0));
```

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

The operator || means OR, so the line

```
if (c == ' ' || c == '\n' || c == '\t')
```

says "if c is a blank *or* c is a newline *or* c is a tab". (Recall that the escape sequence \t is a visible representation of the tab character.) There is a corresponding operator && for AND; its precedence is just higher than ||. Expressions connected by && or || are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known.

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

How will you test the program?

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

What is purpose of testing a code?

input file contains 1 enormous word without any newlines
input file contains all white space without newlines
input file contains 66000 newlines
input file contains word/{huge sequence of whitespace of different kinds}/word
input file contains 66000 single letter words, 66 to the line
input file contains 66000 words without any newlines

Word count program

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

What is purpose of testing a code?

NEED TO CHECK FOR THESE SITUATIONS + SOME MORE

input file contains 1 enormous word without any newlines
input file contains all white space without newlines
input file contains 66000 newlines
input file contains word/{huge sequence of whitespace of different kinds}/word
input file contains 66000 single letter words, 66 to the line
input file contains 66000 words without any newlines

HW: Modify the Word count program to count number of words of different lengths

```
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

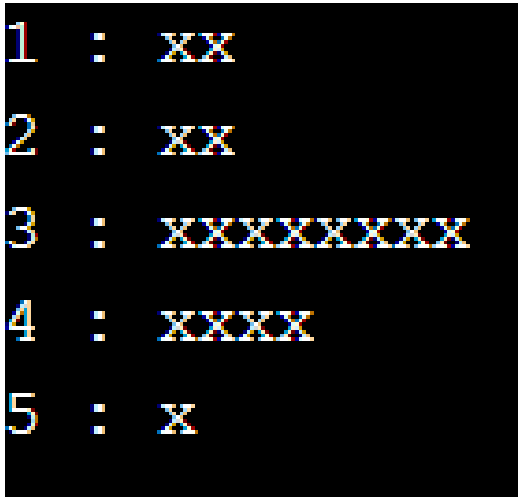
Modify the program so you can generate the following report

Word Length **Number of Words of that length**

Arrays, Functions call, modular programs

Now we will focus on data structure array and more on writing functions

- Count the number of each vowel in the input (stdio)
- Draw a histogram showing the frequency of each vowel. E.g.



- Here 1,2,3,4,5 stand for a/A,e/E,i/I,o/O,u/U

First Write a program to count the number of each vowel

- A,a,E,e,I,i,O,o,U,u
- Stat with counting a or A
- You have to count both A and a, and give a total count of A and a, etc.

First Write a program to count the number of each vowel

- A,a,E,e,I,i,O,o,U,u
- Stat with counting a or A
- You have to count both A and a,

  and give a total count of A and a

```c
#include <stdio.h>

int main (){

    int c=0;
    int n_a=0;

    while ((c=getchar()) != EOF)

    {
        if (c=='a' || c== 'A')

        ++n_a;


    }
    printf("No. of a and A is %d\n",n_a);

    return 0;
}
```

First Write a program to count the number of each vowel

```
{
    int c=0;
    int n_a =0; int n_e = 0; int  n_i = 0; int n_o =0; int n_u = 0;

    while ((c=getchar()) != EOF)

    if (c=='a' || c=='A')
    ++n_a;
    else if(c=='e' || c=='E')
    ++n_e; S
    else if(c=='i' || c=='I')
    ++n_i;
    else if(c=='o' || c=='O')A
    ++n_o;
    else if(c=='u' || c=='U')
    ++n_u;
    else ;

    printf("Number of a and A is %d\n", n_a);
    printf("Number of e and E is %d\n", n_e);
    printf("Number of i and I is %d\n", n_i);
    printf("Number of o and O is %d\n", n_o);
    printf("Number of u and U is %d\n", n_u);


    return 0;
}
```

Reviewing a program to count number of vowels

```c
#include <stdio.h>

int main()

{
int c=0;
int n_a , n_e, n_i, n_o,n_u;

n_a= n_e =  n_i =  n_o= n_u=0;

while ((c=getchar()) != EOF)

    {
        if(c == 'a' || c=='A')
            n_a= n_a+1;
        else if (c == 'e' || c=='E')
            n_e= n_e+1;
        else if (c == 'i' || c=='I')
            n_i= n_i+1;
        else if (c == 'o' || c=='O')
            n_o= n_o+1;
        else if (c == 'u' || c=='U')
            n_u= n_u+1;
    }

    printf("The number of a and A is %d \n",n_a);
    printf("The number of e and E is %d \n",n_e);
    printf("The number of i and I is %d \n",n_i);
    printf("The number of o and O is %d \n",n_o);
    printf("The number of u and U is %d \n",n_u);


return 0;
}
```

```c
#include <stdio.h>

int main()

{
int c=0;
int n_a , n_e, n_i, n_o,n_u;

n_a= n_e =  n_i =  n_o= n_u=0;

while ((c=getchar()) != EOF)

    {
        if(c == 'a' || c=='A')
            n_a= n_a+1;
        else if (c == 'e' || c=='E')
            n_e= n_e+1;
        else if (c == 'i' || c=='I')
            n_i= n_i+1;
        else if (c == 'o' || c=='O')
            n_o= n_o+1;
        else if (c == 'u' || c=='U')
            n_u= n_u+1;
    }

    printf("The number of a and A is %d \n",n_a);
    printf("The number of e and E is %d \n",n_e);
    printf("The number of i and I is %d \n",n_i);
    printf("The number of o and O is %d \n",n_o);
    printf("The number of u and U is %d \n",n_u);


return 0;
}
```

```c
int n_v[5]={0,0,0,0,0}; // holds numbers of vowels a/A,e/E,i/I,o/O,u/U
char vowel_lc[ ] = "aeiou";
char vowel_uc[ ] = "AEIOU";
```

Using an array to store the number of each vowel

```c
for (int i =0; i<5; i=i+1)

printf("The number of %c and %c is %d \n", vowel_lc[i], vowel_uc[i], n_v[i]);
```

Drawing a horizontal histogram
Explain how this program works:

Assume that n_v={4,9,5,3,2}

How many variables would you need to do this?

How many (for) loops you have to run through?

Drawing a horizontal histogram
Explain how this program works:

Assume that n_v={4,9,5,3,2}

What will this print?

```
// display horizontal histogram

   for (int j=0; j<5; j= j+1)
   {
   for ( int k=n_v[j]; k>0; k=k-1) printf("x");

   printf("\n");
   }
```

Drawing a horizontal histogram
Write a function that does that.

Assume that n_v={4,9,5,3,2}

```
// display horizontal histogram

   for (int j=0; j<5; j= j+1) // first loop controlling going over each of the FIVE vowels
   {
   for ( int k=n_v[j]; k>0; k=k-1) printf("x"); //second loop printing as many x as the number of each vowel

   printf("\n");
   }
```

Drawing a horizontal histogram
Write a function that does that.

Assume that n_v={4,9,5,3,2}

// display horizontal histogram

```
for (int j=0; j<5; j= j+1)
{
for ( int k=n_v[j]; k>0; k=k-1) printf("x");

printf("\n");
}
```

Declare the function:
- function type
- function parameters

Define the function

Calling the function

Drawing a horizontal histogram
Write a function that does that.

Assume that n_v={4,9,5,3,2}

// display horizontal histogram

```
for (int j=0; j<5; j= j+1)
{
for ( int k=n_v[j]; k>0; k=k-1) printf("x");

printf("\n");
}
```

Declare the function:
- function type
- function parameters

Define the function

Calling the function

Drawing a horizontal histogram
Write a function that does that.

Assume that n_v={4,9,5,3,2}

// display horizontal histogram

```
for (int j=0; j<5; j= j+1)
{
for ( int k=n_v[j]; k>0; k=k-1) printf("x");

printf("\n");
}
```

Declare the function:

function type

function parameters

```
void draw_hist_h(int entry, int data[])
```

Drawing a horizontal histogram
Write a function that does that.

Assume that n_v={4,9,5,3,2}

// display horizontal histogram

```
for (int j=0; j<5; j= j+1)
{
for ( int k=n_v[j]; k>0; k=k-1) printf("x");

printf("\n");
}
```

Declare the function:

function type

function parameters

void draw_hist_h(int entries, int entry_values[])

```
void draw_hist_h(int entries, int entry_values[])

{

for (int j=0; j<entries; j= j+1)
   {


   for ( int k=entry_values[j]; k>0; k=k-1) printf("x");

   printf("\n");
   }

}
```

……..

```
draw_hist_h(5, n_v);
```

```
#include <stdio.h>

void draw_hist_h(int entries, int entry_values[])

 {
 for (int j=0; j<entries; j= j+1)
    { for ( int k=entry_values[j]; k>0; k=k-1) printf("x");

    printf("\n");
    }
 }
//----------------------------------------
int main()

{
int c=0;

int n_v[5]={0,0,0,0,0}; // holds numbers of vowels a/A,e/E,i/I,o/O,u/U
char vowel_lc[] = "aeiou";
char vowel_uc[] = "AEIOU";

while ((c=getchar()) != EOF)
    {
        if(c == 'a' || c=='A')
            n_v[0]= n_v[0]+1;
        else if (c == 'e' || c=='E')
            n_v[1]= n_v[1]+1;
        else if (c == 'i' || c=='I')
            n_v[2]= n_v[2]+1;
        else if (c == 'o' || c=='O')
            n_v[3]= n_v[3]+1;
        else if (c == 'u' || c=='U')
            n_v[4]= n_v[4]+1;
    }
    for (int i =0; i<5; i=i+1)
    printf("The number of %c and %c is %d \n",vowel_lc[i],vowel_uc[i], n_v[i]);
    // Draw histogram using the function draw_hist_h, the parameter entries 5 and the entry_value[] is n_v
draw_hist_h(5, n_v);
return 0;}
```

13Oct2021

Arrays in C

- Arrays is a datatype consisting of other (usually but not necessarily more primitive) datatypes.
- An array is a collection of similar data items stored at contiguous memory locations
- Elements of an array can be accessed randomly using indices of the array – In C, the index begins with 0.
- They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C can store derived data types such as the structures, pointers etc.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

In this example, memory space is created to hold 10 items of double data type. At this stage, they are NOT initialized and the memory locations may have garbage/unusuable data .

Another way to declare and initialize, for example; int marks[]={5,9,3,10,2}. This will create an array of just five elements initialized to these values.

Accessing Array elements

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary

Note that the variable salary is declared and initialized through an assignment.

Also, it is important that balance[9] has already been initialized.

In general, it is important to initialize all the array elements to some default value as soon as possible.

```c
#include <stdio.h>
int main ()

{ int n[ 10 ]; /* n is an array of 10 integers */

int i,j; /* initialize elements of array n to 0 */

for ( i = 0; i < 10; i++ )
{ n[ i ] = i + 100; /* set element at location i to i + 100 */
}

/* output each array element's value */
for (j = 0; j < 10; j++ )
{ printf("Element[%d] = %d\n", j, n[j] ); }
return 0; }
```

What is the output?

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

```c
#include <stdio.h>
int main ()
{ int n[ 10 ]; /* n is an array of 10 integers */
int i,j; /* initialize elements of array n to 0 */
for ( i = 0; i < 10; i++ )
{ n[ i ] = i + 100; /* set element at location i to i + 100 */ }
/* output each array element's value */
for (j = 0; j < 10; j++ )
{ printf("Element[%d] = %d\n", j, n[j] ); }
return 0; }
```

What is the output?

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

- Suppose we wish to store the marks of the students of a class in an integer array so that we can calculate the average, the max etc.
- Keep in mind the class size differs from class to class but you can assume a maximum of 100 students.
- Assume that for a particular class we have 25 students and their marks are populated in the first 25 elements in the array.
- We want to write a generic function find the maximum mark that works for other classes too. What should be the arguments for such a function

  int maxMarks(????)
- We will continue this discussion in the next class. YOU MAY TRY TO WRITE A COMPLETE PROGRAM WITH SOME DUMMY MARKS, to see if you can write such a function and if it works.
- This is a prelude to working with character arrays and strings which we will be discussing in the next few classes.

Quiz on 20Oct2021

- Everything from minus infinity to today.

18Oct2021

- Passing arguments to a function
- Call by Value vs call by reference. Introducing Pointers.
- Operator Precedence
- Passing an array to a function
- Character Arrays; Strings
- String Operations

Quiz on 20Oct2021

- Everything from minus infinity to last class but today.

Arguments to functions – from the KR book

One aspect of C functions may be unfamiliar to programmers who are used to some other languages, particularly Fortran. In C, all function arguments are passed "by value." This means that the called function is given the values of its arguments in temporary variables rather than the originals. This leads to some different properties than are seen with "call by reference" languages like Fortran or with `var` parameters in Pascal, in which the called routine has access to the original argument, not a local copy.

The main distinction is that in C the called function cannot directly alter a variable in the calling function; it can only alter its private, temporary copy.

Example: The power function

```
/* power:  raise base to n-th power; n >= 0 */
/*         (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Example: The power function

```
/* power:  raise base to n-th power; n >= 0 */
/*         (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

In function declaration it is better to have int power(int base, int n)
Here the variable i is redundant.

Example: The power function

```
/* power:  raise base to n-th power; n >= 0 */
/*         (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

In function declaration it is better to have int power(int base, int n)
Here the variable i is redundant.

```
/* power:  raise base to n-th power; n>=0; version 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Example: The power function

```
/* power:  raise base to n-th power; n >= 0 */
/*         (old-style version) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

In function declaration it is better to have int power(int base, int n)
Here the variable i is redundant.

```
/* power:  raise base to n-th power; n>=0; version 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

The parameter `n` is used as a temporary variable, and is counted down (a `for` loop that runs backwards) until it becomes zero; there is no longer a need for the variable `i`. Whatever is done to `n` inside `power` has no effect on the argument that `power` was originally called with.

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the *address* of the variable to be set (technically a *pointer* to the variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. We will cover pointers in Chapter 5.

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array—there is no copying of array elements. By subscripting this value, the function can access and alter any element of the array. This is the topic of the next section.

Call by value and Call by reference

- Call by value in C
  - In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
  - In call by value method, we can not modify the value of the actual parameter by the formal parameter.
  - In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
  - The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

- Call by reference in C
  - In call by reference, the address of the variable is passed into the function call as the actual parameter.
  - The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
  - In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Call by value and Call by references

```
1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12.return 0;
13.}
```

Call by value and Call by references

```
1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", )um;
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12.return 0;
13.}
```

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

```
1. #include <stdio.h>
2. void swap(int , int); //prototype of the function'
3. void swap(int a; int b)
4. {
5.     int temp;
6.     temp = a;
7.     a=b;
8.     b=temp;
9.     printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
10.}
11.
12.int main()
13.{
14.    int a = 10;
15.    int b = 20;
16.    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
17.    swap(a,b);
18.    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in c
    all by value, a = 10, b = 20
19.}
20.
```

CALL BY REFERENCE does not swap:

```
1. #include <stdio.h>
2. void swap(int , int); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(a,b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the for
   mal parameters in call by value, a = 10, b = 20
10.}
11.void swap (int a, int b)
12.{
13.    int temp;
14.    temp = a;
15.    a=b;
16.    b=temp;
17.    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
18.}
```

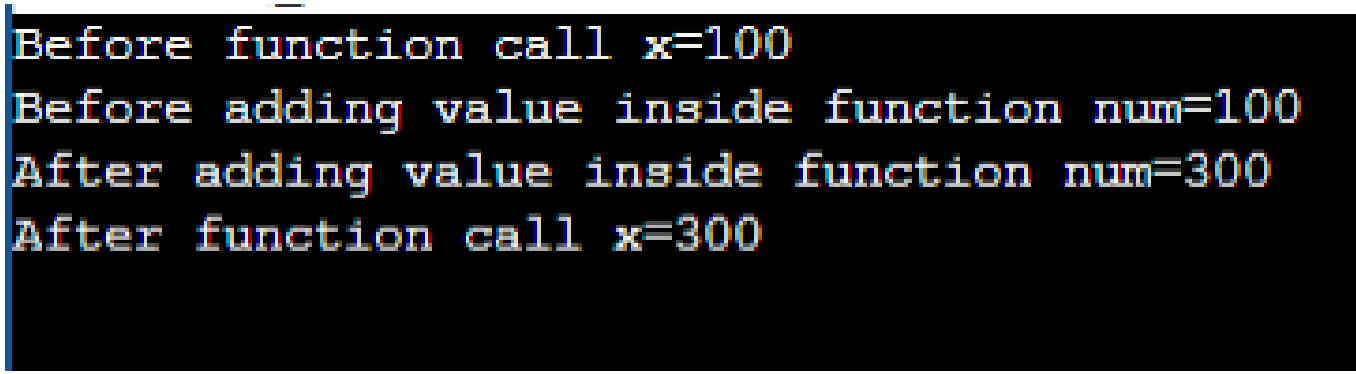
```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference – a example.

```
1.  #include<stdio.h>
2.  void change(int* num) {
3.      printf("Before adding value inside function num=%d \n",*num);
4.      (*num) += 200;
5.      printf("After adding value inside function num=%d \n", *num);
6.  }
7.  int main() {
8.      int x=100;
9.      printf("Before function call x=%d \n", x);
10.     change(&x);//passing reference in function
11.     printf("After function call x=%d \n", x);
12. return 0;
13. }
```

Call by reference – a example.

```
1.  #include<stdio.h>
2.  void change(int* num) {
3.      printf("Before adding value inside function num=%d \n",*num);
4.      (*num) += 200;
5.      printf("After adding value inside function num=%d \n", *num);
6.  }
7.  int main() {
8.      int x=100;
9.      printf("Before function call x=%d \n", x);
10.     change(&x);//passing reference in function
11.     printf("After function call x=%d \n", x);
12. return 0;
13. }
```



Call by reference does the swap! INTRODUCING POINTER VARIABLES:

```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call
   by reference, a = 10, b = 20
10.}
11.void swap (int *a, int *b)
12.{
13.    int temp;
14.    temp = *a;
15.    *a=*b;
16.    *b=temp;
17.    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
18.}
```

Call by reference does the swap! INTRODUCING POINTER VARIABLES

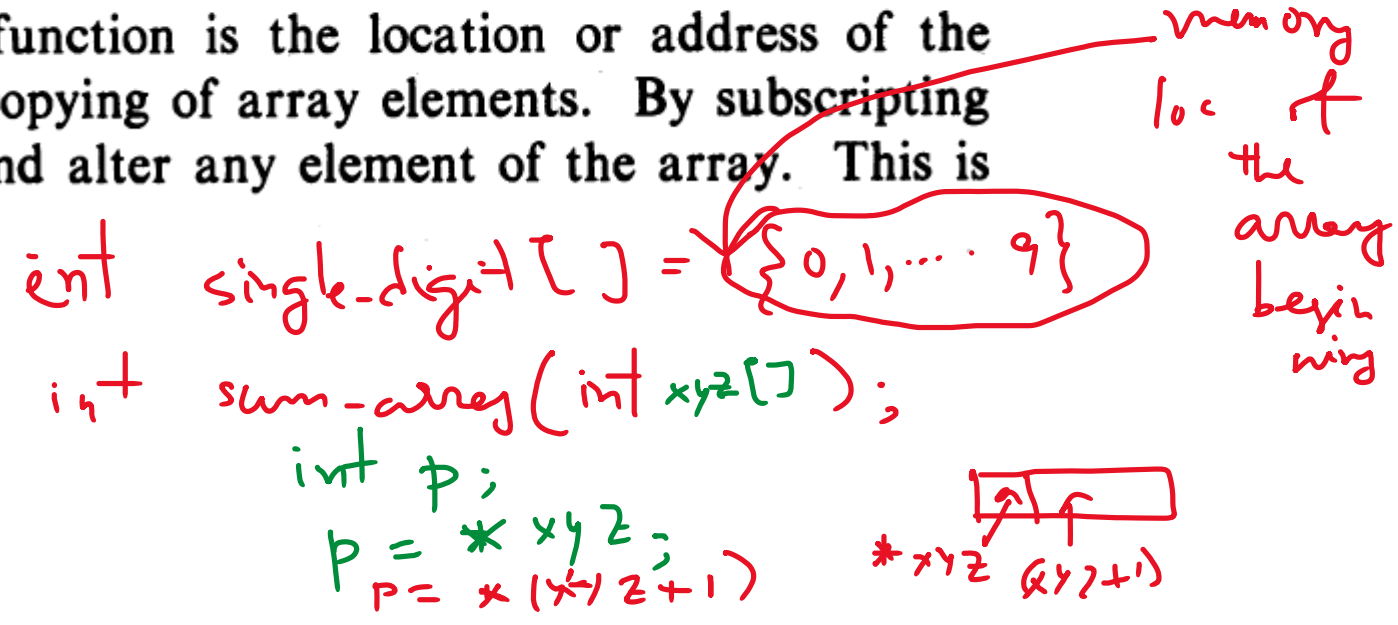
```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call
   by reference, a = 10, b = 20
10.}
11.void swap (int *a, int *b)
12.{
13.    int temp;
14.    temp = *a;
15.    *a=*b;
16.    *b=temp;
17.    printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
18.}
```

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Passing an array to a function in C

When necessary, it is possible to arrange for a function to modify a variable in a calling routine. The caller must provide the *address* of the variable to be set (technically a *pointer* to the variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. We will cover pointers in Chapter 5.

The story is different for arrays. When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array—there is no copying of array elements. By subscripting this value, the function can access and alter any element of the array. This is the topic of the next section.



HOMEWORK

- Please read up on precedence of Operators from the text book on your own. The following slides will serve as a guide

Operators

- Arithmetic Operators
- Relational and Logical Operators
- Increment and Decrement Operators
- Bitwise Operators
- Assignment Operators and Expressions
- Conditional Expressions
- Precedence and Order of Evaluaion

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

The unusual aspect is that `++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix (after the variable: `n++`). In both cases, the effect is to increment `n`. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. If `n` is 5, then

```
x = n++;
```

sets `x` to 5, but

```
x = ++n;
```

sets `x` to 6. In both cases, `n` becomes 6. The increment and decrement operators can only be applied to variables; an expression like `(i+j)++` is illegal.

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

The unusual aspect is that `++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix (after the variable: `n++`). In both cases, the effect is to increment `n`. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used. This means that in a context where the value is being used, not just the effect, `++n` and `n++` are different. If `n` is 5, then

```
x = n++;
```

sets `x` to 5, but

```
x = ++n;
```

sets `x` to 6. In both cases, `n` becomes 6. The increment and decrement operators can only be applied to variables; an expression like `(i+j)++` is illegal.

For example,

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

can be written more compactly as

```
if (c == '\n')
    s[i++] = c;
```

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

2.10 Assignment Operators and Expressions

Expressions such as

```
i = i + 2
```

in which the variable on the left hand side is repeated immediately on the right, can be written in the compressed form

```
i += 2
```

The operator `+=` is called an *assignment operator*.

Most binary operators (operators like `+` that have a left and right operand) have a corresponding assignment operator *op*`=`, where *op* is one of

```
+   -   *   /   %   <<   >>   &   ^   |
```

If $expr_1$ and $expr_2$ are expressions, then

$expr_1$ *op*`=` $expr_2$

is equivalent to

$expr_1$ `=` `(`$expr_1$`)` *op* `(`$expr_2$`)`

except that $expr_1$ is computed only once. Notice the parentheses around $expr_2$:

```
x *= y + 1
```

means

```
x = x * (y + 1)
```

rather than

```
x = x * y + 1
```

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

2.11 Conditional Expressions

The statements

```
if (a > b)
    z = a;
else
    z = b;
```

compute in `z` the maximum of `a` and `b`. The *conditional expression*, written with the ternary operator "`?:`", provides an alternate way to write this and similar constructions. In the expression

$expr_1$ `?` $expr_2$ `:` $expr_3$

the expression $expr_1$ is evaluated first. If it is non-zero (true), then the expression $expr_2$ is evaluated, and that is the value of the conditional expression. Otherwise $expr_3$ is evaluated, and that is the value. Only one of $expr_2$ and $expr_3$ is evaluated. Thus to set `z` to the maximum of `a` and `b`,

```
z = (a > b) ? a : b;     /* z = max(a, b) */
```

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

We will discuss the rest of the operands like sizeof, (type) *, & and BIT WISE OPERATORS LATER

27Oct2021

- Pointers
- Passing an array to a function
- Character Arrays; Strings
- String Operations

Detailed Discussion on Pointers

- Pointer Arithmetic
- Array of pointers
- Pointer to Pointers
- Passing pointers to functions (already doing it)
- Return a pointer from functions in C

Pointer Arithmetic

- Basic Idea is this: A pointer in c is an address, which is a numeric value. Therefore, One can perform arithmetic operations on a pointer just as one can on a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and –

Pointer Arithmetic

- Basic Idea is this: A pointer in c is an address, which is a numeric value. Therefore, One can perform arithmetic operations on a pointer just as one can on a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and –
- Suppose we have

int *pt1;

pt1 = 1000;

char *pt2; pt2 = 5000;

What will be the values of

pt1=++pt1; pt2= ++pt2

Pointer Arithmetic

- Basic Idea is this: A pointer in c is an address, which is a numeric value. Therefore, One can perform arithmetic operations on a pointer just as one can on a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and –
- Suppose we have

int *pt1 = 1000; char *pt2 = 5000

What will be the values of

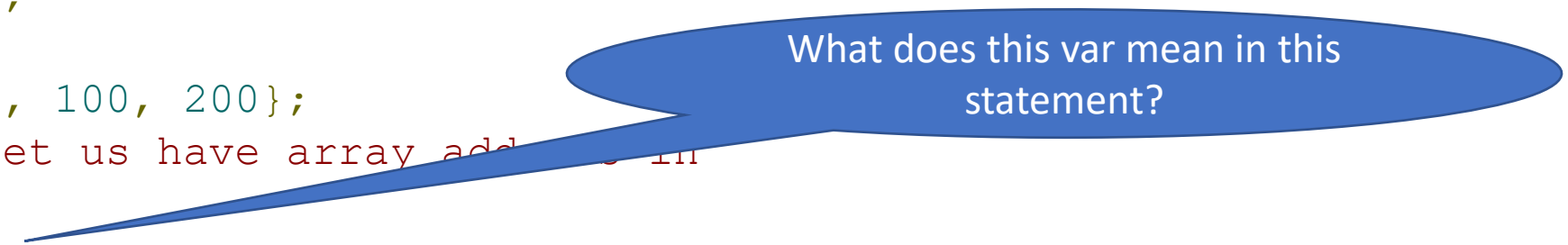
pt1=++pt1; pt2= ++pt2; [NOT advisable without proper casting or appropriate context – will discuss in a later slide]

Pointer arithmetic: An example

```
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};
int i, *ptr; /* let us have array address in
pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++; } return 0; }
```

Pointer arithmetic: An example

```
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};
int i, *ptr; /* let us have array add
pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++; } return 0; }
```



Pointer arithmetic: An example

```
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};
int i, *ptr; /* let us have array address in
pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++; } return 0; }
```

In C an array name is the address to the first element of the array; it is a pointer

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};
int i, *ptr; /* let us have array address in
pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++; } return 0; }
```

```
Address of var[0] = bf882b30
Value of var[0] = 100
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{ int var[] = {10, 100, 200};
int i, *ptr; /* let us have array address in
pointer */
ptr = var;
for ( i = 0; i < MAX; i++) {
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++; } return 0; }
```

```
Address of var[0] = bf882b30
Value of var[0] = 100
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

At least two mistakes in this slide

- Type Conversion

- Type Conversion

```
/* lower:  convert c to lower case; ASCII only */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

- Type Conversion

```
/* lower:  convert c to lower case; ASCII only */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Comparing 2 data types in the same expression.
An example of implicit conversion

Type Casting

- A type cast is basically a conversion from one type to another.
- Implicit Type Conversion – done by the compiler on its own
- Explicit Type Casting

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer to floating point in an expression like `f + i`. Expressions that don't make sense, like using a `float` as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

Type Casting

- A type cast is basically a conversion from one type to another.
- Implicit Type Conversion – done by the compiler on its own
- Explicit Type Casting

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c

        // y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;

        // x is implicitly converted to float
        float z = x + 1.0;

        printf("x = %d, z = %f", x, z);
        return 0;
}
```

Type Casting

- A type cast is basically a conversion from one type to another.
- Implicit Type Conversion – done by the compiler on its own
- Explicit Type Casting

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c

        // y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;

        // x is implicitly converted to float
        float z ; z= x + 1.0;

        printf("x = %d, z = %f", x, z);
        return 0;
}
```

Type Casting

- A type cast is basically a conversion from one type to another.
- Implicit Type Conversion – done by the compiler on its own
- Explicit Type Casting

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c

        // y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;

        // x is implicitly converted to float
        float z ; z= x + 1.0;

        printf("x = %d, z = %f", x, z);
        return 0;
}
```

This is also known as integer promotion

Type Casting

- A type cast is basically a conversion from one type to another.
- Implicit Type Conversion – done by the compiler on its own
- Explicit Type Casting

All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double
```

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z ; z= x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}
```

Explicit type casting

- `Syntax: (type) expression`

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
        double x = 1.2;

        // Explicit conversion from double to int
        int sum = (int)x + 1;

        printf("sum = %d", sum);

        return 0;
}
```

5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

5.3 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.

The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

```
pa = &a[0];
```

`pa` and `a` have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment `pa=&a[0]` can also be written as

```
pa = a;
```

Strings, String functions, Pointers ….

- In C programming, a string is a sequence of characters terminated with a null character \0
- Example: char c[]=“c string”;
- When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default
- char c[5]; is a declaration. Then you can feed a string into it subject to the storage you have specified for space for \0

Common string functions

Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

Strings handling functions are defined under `"string.h"` header file.

```
#include <string.h>
```

Simple program to find string length

```
#include <stdio.h>
int main() {
char s[] = “C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string: %d", i);
return 0; }
```

Simple program to find string length

- Convert this program into a function
- What should be the function declaration?
- What are the parameters?
- What is the function definition?

```
#include <stdio.h>
int main() {
char s[] = “C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string: %d", i);
return 0; }
```

Simple program to find string length

- Convert this program into a function
- What should be the function declaration?
- What are the parameters
- What is the function definition

```
int StringL(char s[])
{

for (int i=0; s[i] != ‘\0’; ++i);
return i;
}
```

```
#include <stdio.h>
int main() {
char s[] = “C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string:
%d", i); return 0; }
```

Simple program to find string length

- Convert this program into a function
- What should be the function declaration?
- What are the parameters
- What is the function definition

```
int StringL(char s[])
{

for (int i=0; s[i] != ‘\0’; ++i);
return i;
}
```

There is a FATAL Mistake in the definition of the function StringL

```
#include <stdio.h>
\\ include def of StringL here
int main() {
char s[] = “C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string: %d",
StringL(s));
return 0; }
```

Simple program to find string length

- Convert this program into a function
- What should be the function declaration?
- What are the parameters
- What is the function definition

```
int StringL(char s[])
{

for (int i=0; s[i] != '\0'; ++i);
return i;
}
```

```
#include <stdio.h>
int main() {
char s[] = "C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string: %d",
StringL(s));
return 0; }
```

Rewrite this program using pointers

Simple program to find string length

- Convert this program into a function
- What should be the function declaration?
- What are the parameters
- What is the function definition

```
int StringL(char s[])
{

for (int i=0; s[i] != ‘\0’; ++i);
return i;
}
```

```
#include <stdio.h>
int main() {
char s[] = “C Programming";
int i;
for (i = 0; s[i] != '\0'; ++i);
printf("Length of the string:
%d", i); return 0; }
```

```
int String_L(char*s) /* s=&str[0] */
{
  int count = 0;
  while (*s != '\0') {
    count++;
    s++;
  }
  return count;
}
```

Now we can write a number of sophisticated programs/functions with arrays.

- Sort an integer array
- Insert into an array
- Reverse a string
- Copy a string
- Sorting an array. Merge sort, Quick sort.

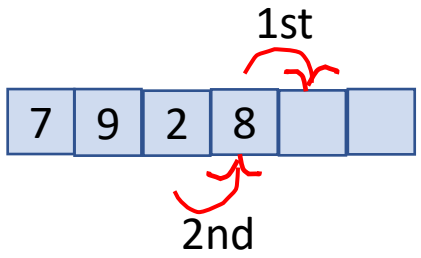
Insert a number in an array of integers

void insert(int a[], int n, int i,) // insert i in the beginning, zeroth place

PLEASE WRITE THE FUNCTION NOW

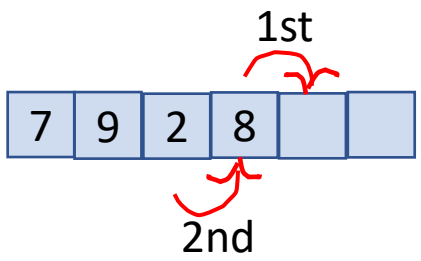
Insert a number in an array of integers

void insert(int a[], int n, int i,) // insert in in the beginning, zeroth place



Insert a number in an array of integers

```
void insert(int a[], int n, int i,) // insert in in the beginning, zeroth place


void insert(int a[], int n, int i)
//n is length of array, insert i in the 0th place
{     for (int k=n-1; k >= 0; k--)
{    *(a+k+1)=*(a+k);        }

*a=i;
}
```



Insert a number in an array of integers

REWRITE THE FUNCTION TO insert at the jth place (array[j-1])

void insert(int a[], int n, int i, int j) // insert integer i at the j-th place

Insert a number in an array of integers

REWRITE THE FUNCTION TO insert at the jth place (array[j-1])

void insert(int a[], int n, int i, int j) // insert integer i at the j-th place
VERY SIMPLE MODIFICATION TO

```
{      for (int k=n-1; k >= 0; k--)
{    *(a+k+1)=*(a+k);        }

*a=i;
}
```

Insert a number in an array of integers

```
void insert(int a[], int n, int i,) // insert in in the beginning, zeroth place


void insert(int a[], int n, int i)
//n is length of array, insert i in the jth place
{     for (int k=n-1; k >= j; k--)
{    *(a+k+1)=*(a+k);        }

*(a+j)=i;
}
```

Sample Code1

```
#include <stdio.h>
void insert(int a[], int n, int i) //n is length of array, insert i in the 0th place
{
for (int k=n-1; k >= 0; k--)
{    *(a+k+1)=*(a+k);       }
*a=i;}

int main(){
   int N[50] = {3,7};
for(int i=0; i<2; i++)
{    printf("%d\n",*(N+i));
}
printf("\nwill insert a number now and then print it out\n");

 insert(N,2,5);

 for(int i=0; i<3; i++){
printf("%d\n",*(N+i));
 }
return 0;
}
```

Sample Code2

```
#include <stdio.h>
void insert(int a[], int n, int i, int j) //n is length of array, insert i in the jth place
{
for (int k=n-1; k >= j; k--)
{    *(a+k+1)=*(a+k);       }
*(a+j)=i;
}

int main(){
   int N[50] = {3,7};
for(int i=0; i<2; i++)
{    printf("%d\n",*(N+i));
}
printf("\nwill insert a number now and then print it out\n");

 insert(N,2,5,1);

 for(int i=0; i<3; i++){
printf("%d\n",*(N+i));
 }
return 0;
}
```

What is the time complexity of insertion in an array?

- On the average how many operations needed to insert at a random array index?

What is the time complexity of insertion in an array?

- On the average how many operations needed to insert at a random array index?
- O(n), Can be substantial if n is large.
- How can we make it O(1), that is independent of the length of the array?
- This leads us to a discussion on linked lists, and structures in general which we will discuss later in the course.

Some loose ends to be covered

- Scope of Variables
- How to read and write into files….
- Break, switch…. Etc.

Later to come

- Structures
  - Defining structures
  - calling structures
  - memory allocation, deallocation,
- Some basic data structures
- Some basic algorithm and basic algorithmic complexity considerations.
- Some of these will not be covered ibn the K&R book, and supplementary material available on the net will be used for these.

1Nov2021

- Scope of Variables

Variable Scope

- An object is recognized by its identifier or name. The object may be a variable of basic type or a function, a structure, or a union.
- The scope of a variable is the range of program statements that can access that variable.
- A variable is visible within its scope and invisible or hidden outside it
- We will Look at an example.

Global Variables

Variables that are declared outside of a function block and can be accessed inside the function is called global variables.

Global Variable Initialization

After defining a local variable, the system or the compiler won't be initializing any value to it. You have to initialize it by yourself. It is considered good programming practice to initialize variables before using. Whereas in contrast, global variables get initialized automatically by the compiler as and when defined. Here's how based on datatype; global variables are defined.

datatype	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

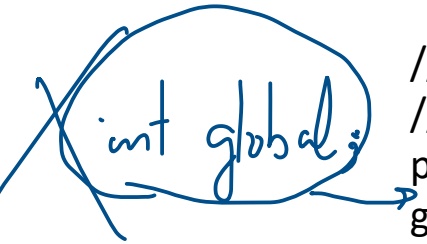
Variable Scope, Global variables

In C every variable defined in scope. You can define scope as the section or region of a program where a variable has its existence; moreover, that variable cannot be used or accessed beyond that region.

In C programming, variable declared within a function is different from a variable declared outside of a function. The variable can be declared in three places. These are:

Position	Type
Inside a function or a block.	local variables
Out of all functions.	Global variables
In the function parameters.	Formal parameters

```c
// C program to illustrate the global scope

#include <stdio.h>

// Global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
        printf("%d\n", global);
}

// main function
int main()
{
        printf("Before change within main: ");
        display();

        // changing value of global
        // variable from main function
        printf("After change within main: ");
        global = 10;
        display();
}
```



extern int global;

Extern is a keyword in C programming language which is used to declare a global variable that is a variable without any memory assigned to it. It is used to declare variables and functions in header files. Extern can be used access variables across C files.

Syntax:

```
extern <data_type> <variable_name>;
// or
extern <return_type> <function_name>(<parameter_list>);
```

- Declaration

Declaration of a variable means that the compiler knows that the variable exists but no memory or data has been assigned to it. To stop at this step, we need extern keyword like:

```
extern int opengenus;
```

External variables are also known as global variables. These variables are defined outside the function. These variables are available globally throughout the function execution. The value of global variables can be modified by the functions. “extern” keyword is used to declare and define the external variables.

Scope − They are not bound by any function. They are everywhere in the program i.e. global.

Default value − Default initialized value of global variables are Zero.

Lifetime − Till the end of the execution of the program.

Here are some important points about extern keyword in C language,

- External variables can be declared number of times but defined only once.
- “extern” keyword is used to extend the visibility of function or variable.
- By default the functions are visible throughout the program, there is no need to declare or define extern functions. It just increase the redundancy.
- Variables with “extern” keyword are only declared not defined.
- Initialization of extern variable is considered as the definition of the extern variable.

```c
#include <stdio.h>
extern int x = 32;
int b = 8;
int main() {
   auto int a = 28;
   extern int b;
   printf("The value of auto variable : %d\n", a);
   printf("The value of extern variables x and b : %d,%d\n",x,b);
   x = 15;
   printf("The value of modified extern variable x : %d\n",x);
   return 0;
}
```

Output

```
The value of auto variable : 28
The value of extern variables x and b : 32,8
The value of modified extern variable x : 15
```

```c
#include <stdio.h>
extern int x = 32;
int b = 8;
int main() {
   auto int a = 28;
   extern int b;
   printf("The value of auto variable : %d\n", a);
   printf("The value of extern variables x and b : %d,%d\n",x,b);
   x = 15;
   printf("The value of modified extern variable x : %d\n",x);
   return 0;
}
```

Output

```
The value of auto variable : 28
The value of extern variables x and b : 32,8
The value of modified extern variable x : 15
```

Time Permitting we will discuss storage classes in C later in the course. IN that context we will discuss static and auto variables.

Structure

- A structure is a user defined data type. A structure creates a data type that can be used to group items of possibly different types into a single type.
- Example:

```
struct address
{
char street[100];
char city[50];
char state[20];
int pin;
};
```

```
struct ind_address
{
char name[50];
struct address name_add;
};
```

Declaring Structure (Example)

```
// A variable declaration with structure declaration.
struct Point
{
int x, y;
} p1; // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
int x, y;
};

int main()
{
struct Point p1; // The variable p1 is declared like a normal variable
}
```

Initializing Structures

- Wrong

```
struct Point
{
int x = 0; // COMPILER ERROR: cannot initialize members here
int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

- Right

```
struct Point
{
int x, y;
};
int main()
{
// A valid initialization. member x gets value 0 and y
// gets value 1. The order of declaration is followed.
struct Point p1 = {0, 1};
}
```

Accessing structure elements

```
#include<stdio.h>


struct Point
{
int x, y;
};


int main()
{
struct Point p1 = {0, 1};
// Accessing members of point p1
p1.x = 20;
printf ("x = %d, y = %d", p1.x, p1.y);
printf(“Area= %d”, int Area = (p1.x*p1.y));  //WRONG, Area needs to be declared
return 0;
}
```

Array of Structure

```
#include<stdio.h>
struct Point
{
int x, y;
};
int main()
{
// Create an array of structures
struct Point arr[10];
// Access array members
arr[0].x = 10;
arr[0].y = 20;
printf("%d %d", arr[0].x, arr[0].y);
return 0;
}
```

Pointer to a Structure

```
#include<stdio.h>
struct Point
{
int x, y;
};

int main()
{
struct Point p1 = {1, 2};

// p2 is a pointer to structure p1
struct Point *p2 = &p1;
// Accessing structure members using structure pointer
printf("%d %d", p2->x, p2->y);
return 0;
}
```

Nested Structure

- Nested structure in C is structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure. (give an example)
- The structure variables can be a normal structure variable or a pointer variable to access the data.
- More on these later.

```
struct rect
{
struct Point
{
int x, y;
}

int perimeter;
Int area ;
// you can refer to the structure Point if Point declared outside and global
}
```

Important Data Structures

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Binary Trees
- Heaps
- We will cover some of these

8Nov2021

Important Data Structures

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Binary Trees
- Heaps
- We will cover some of these

Typical Operations involving Data Structures

- Arrays, Linked Lists, Stacks, Queues, Trees, Binary Trees, Heaps
- All data structures typically have these following functions
  I. Create
  II. Insert
  III. Delete
  IV. Read (Retrieve data (information))

Pointers (continued discussion)

- There are a few important operations with pointers very frequently.
  - **(a)** define a pointer variable,
  - **(b)** assign the address of a variable to a pointer and
  - **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

- NULL Pointers
- It is a good practice to assign a NULL value to a pointer variable in case an exact address is yet to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries.

- In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing

- To check that the ptr is pointing to something at all, we can and should that the pointer is not NULL

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

OR

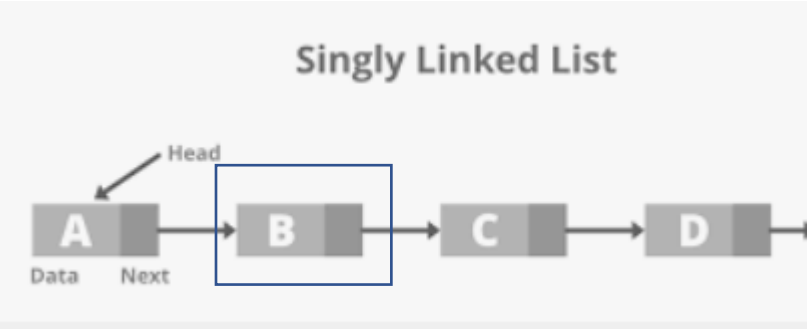
```
if(ptr == NULL) /* succeeds if p is not null */
If(ptr != NULL) /* succeeds if p is null */
```

Linked List

```
// A linked list node containing integers
struct Node {
        int data1;
                struct Node* next;
};
```

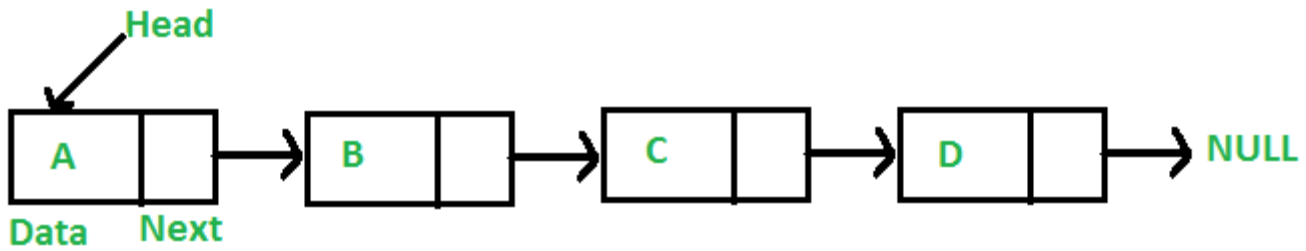
Linked List

```
// A linked list node containing integers and another structure.
struct Node {
        int data1;
        struct ind_address data2;
        struct Node* next;
};
```



- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location
- elements are linked using pointers
- Clearly you need to identify beginning and end of a linearly linked list!! How?

- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location
- elements are linked using pointers
- Clearly you need to identify beginning and end !! How?



Linked List Creation

- Define the node type
- Define a function that allocates memory for a single node (consistent with the “sizeof” the node and returns a pointer.

Sizeof is a much used operator in the C. It is a compile time unary operator which can be used to compute the size of its operand. sizeof can be applied to any data-type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes such as Structure, union etc.

```
#include <stdio.h>
int main()
{
        printf("%lu\n", sizeof(char));
        printf("%lu\n", sizeof(int));
        printf("%lu\n", sizeof(float));
        printf("%lu", sizeof(double));
        return 0;
}
```

Sizeof is a much used operator in the C. It is a compile time unary operator which can be used to compute the size of its operand. sizeof can be applied to any data-type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes such as Structure, union etc.

```
#include <stdio.h>
int main()
{
        printf("%lu\n", sizeof(char));
        printf("%lu\n", sizeof(int));
        printf("%lu\n", sizeof(float));
        printf("%lu", sizeof(double));
        return 0;
}
```

Output:

```
1
4
4
8
```

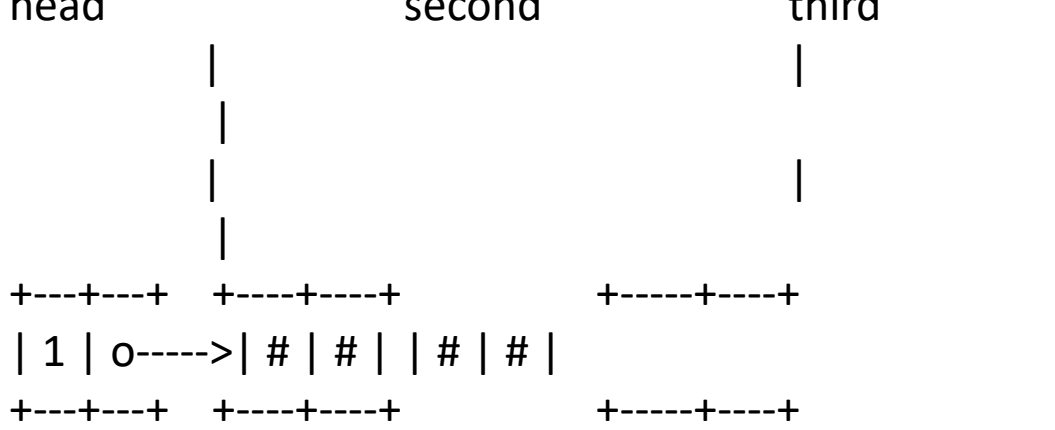
```
// A simple C program to introduce a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
        int data;
        struct Node* next;
};

// Program to create a simple linked  list with 3 nodes
int main()
{
        struct Node* head = NULL;
        struct Node* second = NULL;
        struct Node* third = NULL;

        // allocate 3 nodes in the heap
        head = (struct Node*)malloc(sizeof(struct Node));
        second = (struct Node*)malloc(sizeof(struct Node));
        third = (struct Node*)malloc(sizeof(struct Node));

        /* Three blocks have been allocated dynamically.
        We have pointers to these three blocks as head,
        second and third
        head                    second                  third
                        |                                       |
                         |
                        |                                       |
                         |
        +---+-----+     +----+----+     +----+----+
        | # | # |     | # | # |     | # | # |
        +---+-----+     +----+----+     +----+----+

# represents any random value.  Data is random because we haven't assigned anything yet */
```

```
// A simple C program to introduce a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
        int data;
        struct Node* next;
};
```

```
// A simple C program to introduce a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
        int data;
        struct Node* next;
};

// Program to create a simple linked  list with 3 nodes
int main()
{
        struct Node* head = NULL;
        struct Node* second = NULL;
        struct Node* third = NULL;

        // allocate 3 nodes in the heap
        head = (struct Node*)malloc(sizeof(struct Node));
        second = (struct Node*)malloc(sizeof(struct Node));
        third = (struct Node*)malloc(sizeof(struct Node));

        /* Three blocks have been allocated dynamically.
        We have pointers to these three blocks as head,
        second and third
        head                    second                  third
                        |                                       |
                         |
                        |                                       |
                         |
        +---+-----+     +----+----+     +----+----+
        | # | # |     | # | # |     | # | # |
        +---+-----+     +----+----+     +----+----+

# represents any random value.  Data is random because we haven't assigned anything yet */
```

```
// A simple C program to introduce a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
        int data;
        struct Node* next;
};

// Program to create a simple linked  list with 3 nodes
int main()
{
        struct Node* head = NULL;
        struct Node* second = NULL;
        struct Node* third = NULL;

        // allocate 3 nodes in the heap
        head = (struct Node*)malloc(sizeof(struct Node));
        second = (struct Node*)malloc(sizeof(struct Node));
        third = (struct Node*)malloc(sizeof(struct Node));

        /* Three blocks have been allocated dynamically.
        We have pointers to these three blocks as head,
        second and third
        head                    second                  third
                |                                       |
                 |                                      |
                |                                       |
                 |
        +---+-----+     +----+----+     +----+----+
        | # | # |     | # | # |     | # | # |
        +---+-----+     +----+----+     +----+----+                    */

# represents any random value.  Data is random because we haven't assigned anything yet */
```

```
head->data = 1; // assign data in first node
head->next = second; // Link first node with
// the second node

/* data has been assigned to the data part of the first
block (block pointed by the head). And next
pointer of first block points to second.
So they both are linked.
```



```
// assign data to second node
        second->data = 2;

        // Link second node with the third node
        second->next = third;

        /* data has been assigned to the data part of the second block (block pointed by second).
        And next pointer of the second block points to the third block. So all three blocks are linked.

        head                    second                  third
                |                                       |                                       |
                |                                       |                                       |
        +---+---+       +---+---+       +----+----+
        | 1 | o-----> | 2 | o-----> | # | # |
        +---+---+       +---+---+       +----+----+      */

        third->data = 3; // assign data to third node
        third->next = NULL;

        /* data has been assigned to data part of third  block (block pointed by third). And next   pointer  of the third block is made NULL to indicate
that the linked list is terminated here.

        We have the linked list ready.

                head
                        |
                        |
        +---+---+       +---+---+       +----+------+
        | 1 | o-----> | 2 | o-----> | 3 | NULL |
        +---+---+       +---+---+       +----+------+


        Note that only head is sufficient to represent  the whole list. We can traverse the complete  list by following next pointers. */

        return 0;
}
```

```
struct LinkedList{
    int data;
    struct LinkedList *next;
};
```

Creating a Node:

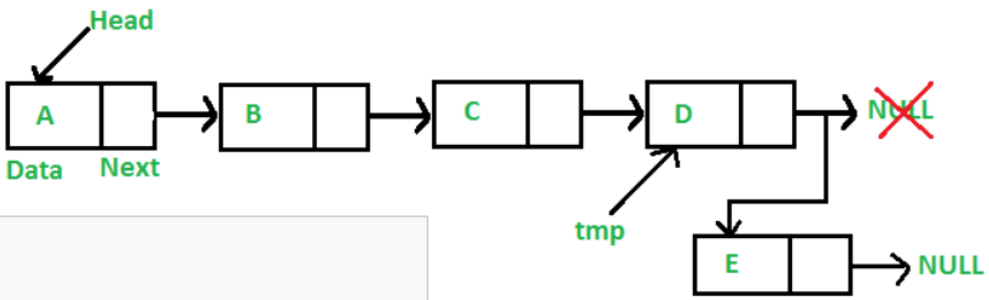
Let's define a data type of `struct LinkedList`to make code cleaner.

```
typedef struct LinkedList *node; //Define node as pointer of data type struct LinkedList

node createNode(){
     node temp; // declare a node
     temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory using malloc()
     temp->next = NULL;// make next point to NULL
     return temp;//return the new node
}
```

> **typedef** is used to define a data type in C.
>
> **malloc()** is used to dynamically allocate a single block of memory in C, it is available in the header file `stdlib.h`.
>
> **sizeof()** is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to `malloc`.

The above code will create a node with data as value and next pointing to NULL.

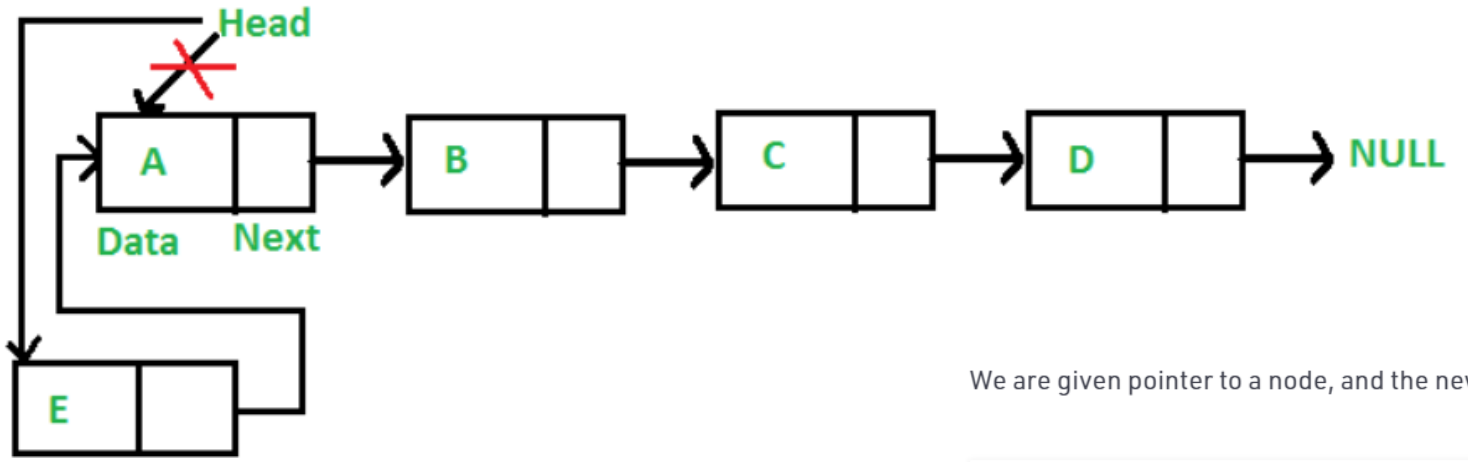
Append a node



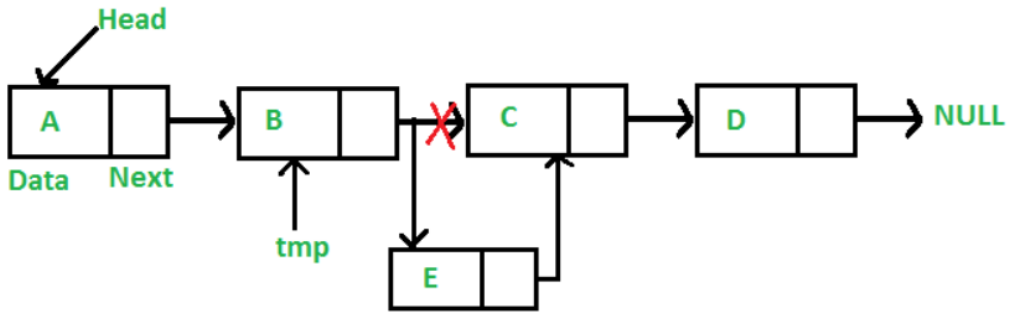

```
node addNode(node head, int value){
     node temp,p;// declare two nodes temp and p
     temp = createNode();//createNode will return a new node with data = value and next pointing to NULL.
     temp->data = value; // add element's value to data part of node
     if(head == NULL){
          head = temp;      //when linked list is empty
     }
     else{
          p  = head;//assign head to p
          while(p->next != NULL){
               p = p->next;//traverse the list until p is the last node.The last node always points to NULL.
          }
          p->next = temp;//Point the previous last node to the new node created.
     }
     return head;
}
```

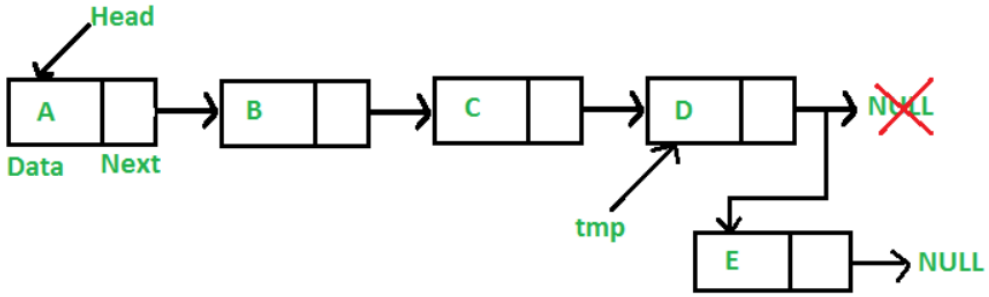
Here the new node will always be added after the last node. This is known as **inserting a node at the rear end**.

Inserting a node



We are given pointer to a node, and the new node is inserted after the given node.





Sorting an array of integers

- Bubble sort

Sorting an array of integers

- Bubble sort
  - **Bubble sort**, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Take an array of numbers " 5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required;

First Pass

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

- Write a C function to do a bubble sort using a C function to swap.

- Write a bubble sort function that uses a swap function

```
//Improved Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
		int temp = *xp;
		*xp = *yp;
		*yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
int i, j;
//bool swapped;
for (i = 0; i < n-1; i++)
{
		//swapped = false;
		for (j = 0; j < n-i-1; j++)
		{
				if (arr[j] > arr[j+1])
				{
				swap(&arr[j], &arr[j+1]);
				//swapped = true;
				}
		}

		// IF no two elements were swapped by inner loop, then break
		//if (swapped == false)
				//break;
}
}
```

- Write a C function to do a bubble sort using a C function to swap.
- Write the bubblesort function

```
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

```
// Optimized implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
```

Homework

- Please understand the C program for bubble sort and optimized bubble sort. Understand how exactly it works.

15Nov2021

A reference to practice C programs (with strings etc.)

- There are many references on the net. You may look up any one of them. In particular take a look at

  https://www.w3schools.in/c-program/

- Take a look at many of these programs and practice.

Last time we began to discuss two distinct things

- how to define and work with a linear but non-contiguously placed data structure, namely a linearly linked list.

A GOOD REF on LINKED LIST: GO To:

http://cslibrary.stanford.edu/103/

- Search and sorting algorithms.

Today we will continue the discussion on different aspects of search and sort algorithm; how to determine correctness of an algorithm, how to measure complexities of algorithms etc.

Introductory Statements

- Typically we need to search for a data item.
- We can search for an item by going linearly across the data set. If the size of the data set is n, then it is plausible that we will take time proportional to n.
- Why do we need to sort then???
- Two reasons:

1. searching a sorted list is cheaper than searching an unsorted list.
2. Typically we need to search more than once !!

SEARCHING A SORTED LIST -- n is len(L)

- using **linear search**, search for an element is **O(n)**
- using **binary search**, can search for an element in **O(log n)**
  - assumes the **list is sorted**!
- when does it make sense to **sort first then search**?
  - SORT + O(log n) < O(n) → SORT < O(n) – O(log n)
  - when sorting is less than O(n)
- **NEVER TRUE!**
  - **to sort a collection of n elements must look at each one at least once!**

AMORTIZED COST -- n is len(L)

- why bother sorting first?
- in some cases, may **sort a list once** then do **many searches**
- **AMORTIZE cost** of the sort over many searches
- SORT + `K`*O(`log n`) < `K`*O(`n`)

→ for large `K`, **SORT time becomes irrelevant,** if cost of sorting is small enough

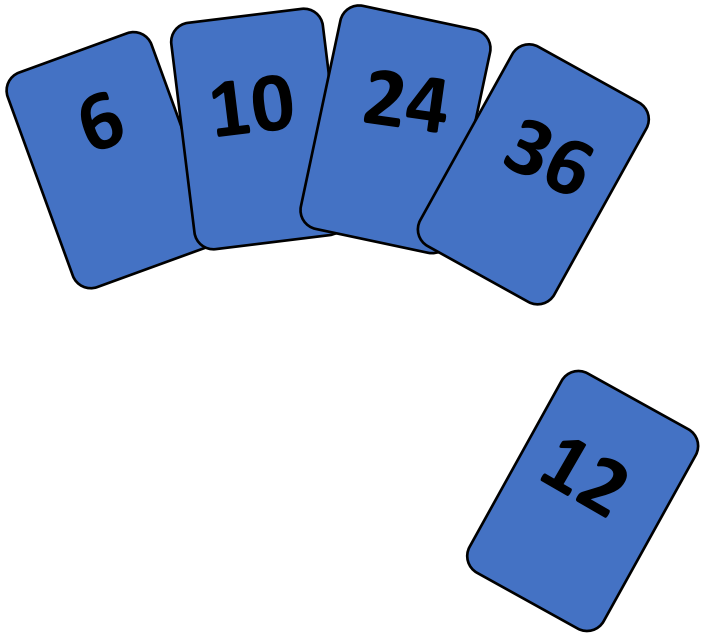
Sorting Algorithms

- Bubble sort
- Insertion Sort (will discuss today)
- Selection sort
- Merge Sort
- Heap sort

Insertion Sort

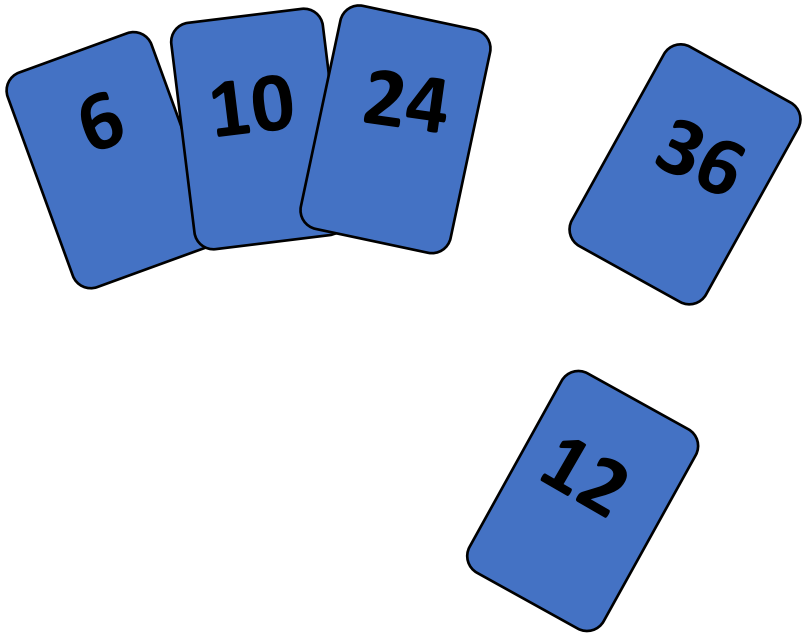
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

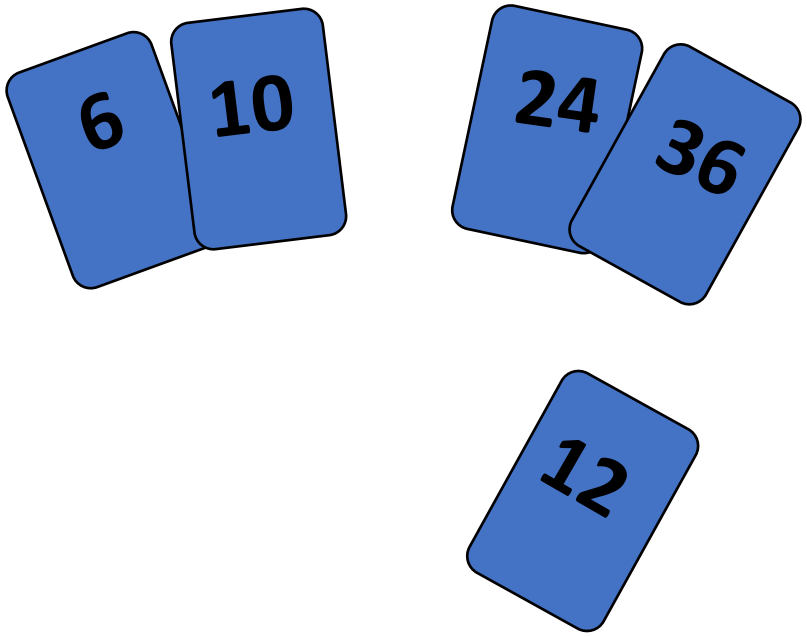


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort

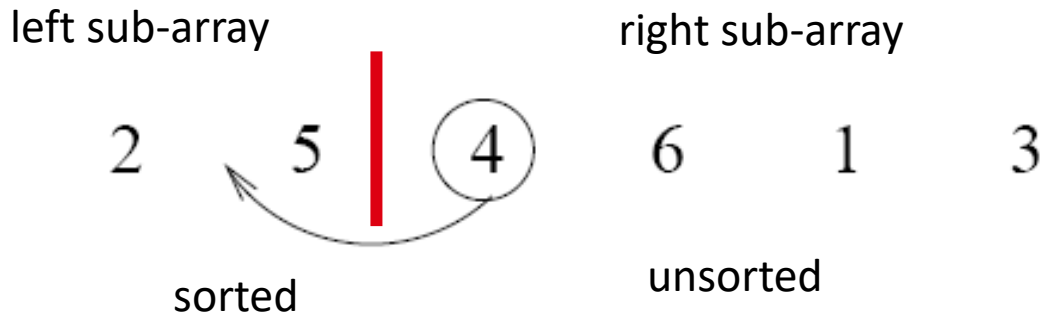


Insertion Sort

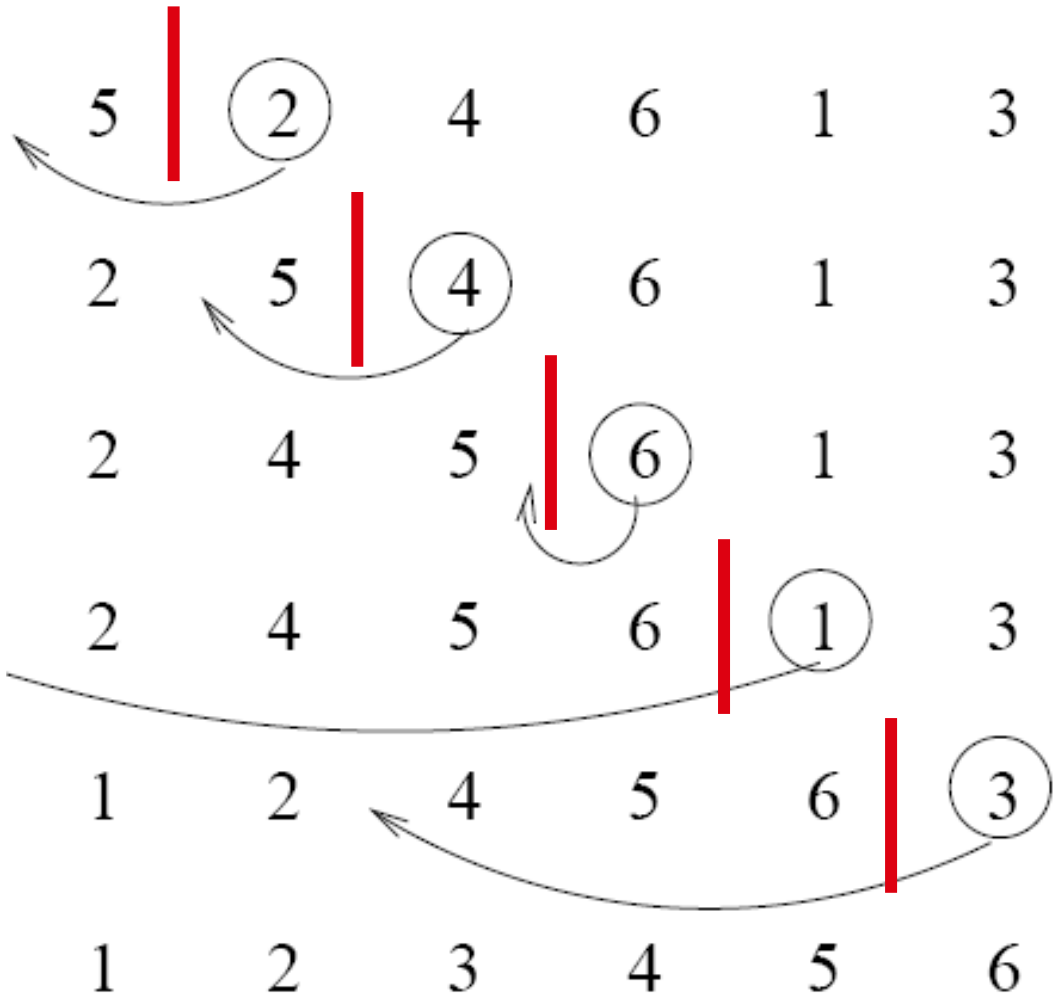
input array

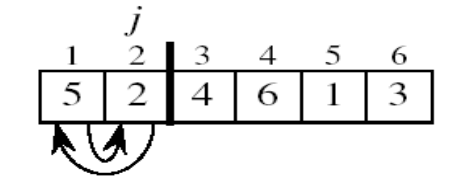
5 2 4 6 1 3

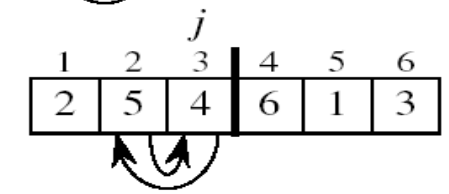
at each iteration, the array is divided in two sub-arrays:

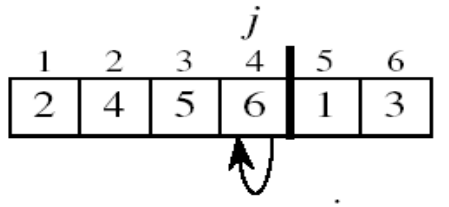


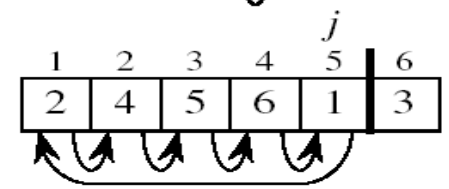
Insertion Sort

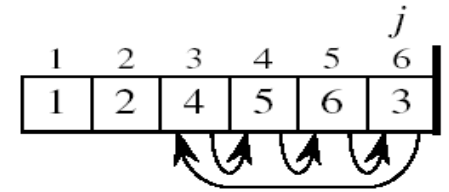






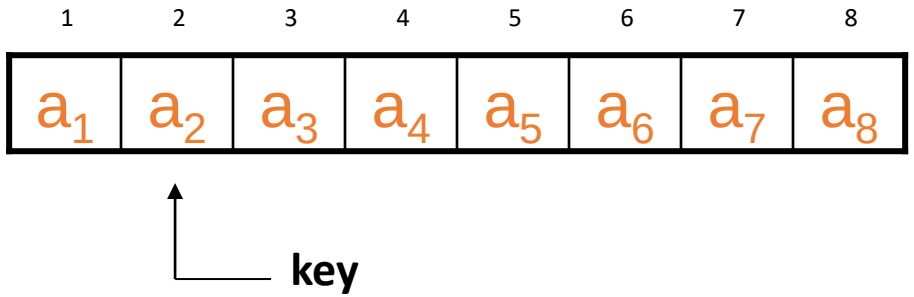






INSERTION-SORT

Alg.: INSERTION-SORT(*A*)



for j ← 2 **to** n

do key ← *A*[j]

▷ Insert *A*[j] into the sorted sequence *A*[1 . . j -1]

i ← j - 1

while i > 0 and *A*[i] > key

do *A*[i + 1] ← *A*[i]

i ← i - 1

A[i + 1] ← key

- Insertion sort – sorts the elements in place

How do we know an algorithm works or that it is correct?

- The idea is define and track a loop invariant.
- A loop invariant is something that remains constant throughout the loop.
- It ensures that the algorithm is correct and if it terminates, then it will give a correct result.

Proving Loop Invariants

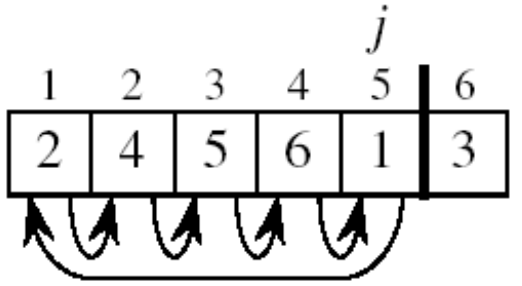
- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT*(A)*

for j ← 2 **to** n

do key ← A[j]



Insert A[j] into the sorted sequence A[1 . . j -1]

i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i - 1

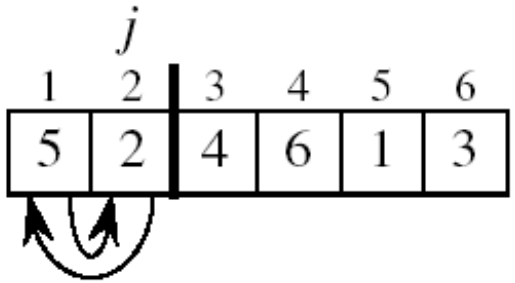
A[i + 1] ← key

Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Loop Invariant for Insertion Sort

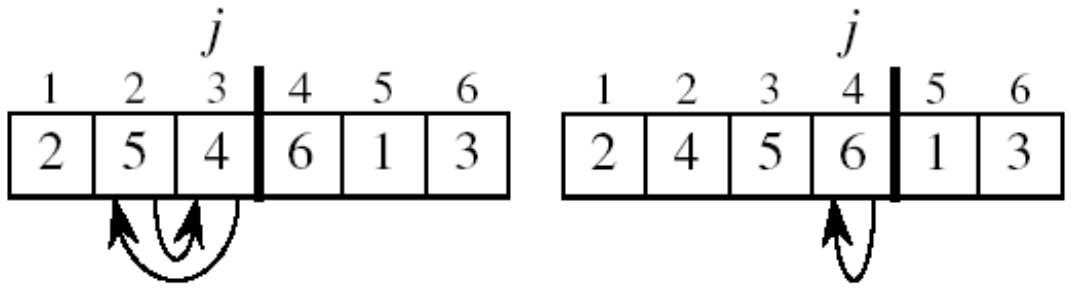
- **Initialization:**
  - Just before the first iteration, $j = 2$:

    the subarray $A[1 \ldots j-1] = A[1]$, (the element originally in $A[1]$) – is sorted



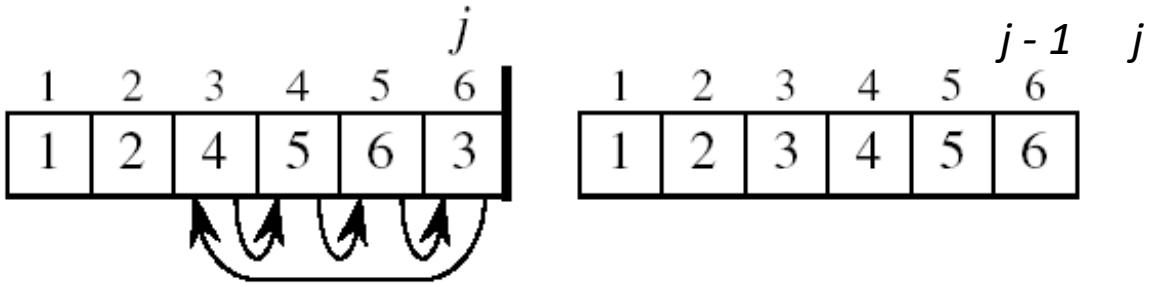
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for key (which has the value that started out in $A[j]$) is found
  - At that point, the value of key is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j = n + 1 \Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 \ldots n]$ consists of the elements originally in $A[1 \ldots n]$, but in sorted order



- The entire array is sorted!

Invariant: at the start of the **for** loop the elements in $A[1 \ldots j-1]$ are in sorted order

Insertion Sort - Summary

We will discuss these notations

- Advantages
  - Good running time for “almost sorted” arrays ($\Theta(n)$)
- Disadvantages

  $\approx n^2/2$ comparisons and exchanges ($\Theta(n^2)$ running time in worst and average case)

Insertion Sort - Summary

Time Complexity:

- **Best Case** Sorted array as input, [**O(N)**]. And **O(1)** swaps.
- **Worst Case**: Reversely sorted, and when inner loop makes maximum comparison, [$\mathbf{O(N^2)}$] . And $\mathbf{O(N^2)}$ swaps.
- **Average Case**: [$\mathbf{O(N^2)}$] . And $\mathbf{O(N^2)}$ swaps.

Space Complexity: [auxiliary, **O(1)**]. In-Place sort.

Advantage:

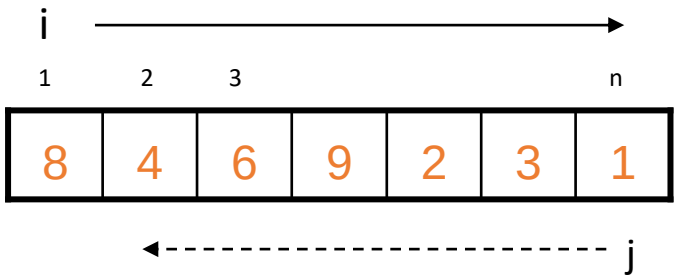
1. It can be easily computed.
2. Best case complexity is of **O(N)** while the array is already sorted.
3. Number of swaps reduced than bubble sort.
4. For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.
5. Stable sort.
6. Adaptive: total number of steps is reduced for partially sorted array.
7. In-Place sort.

Disadvantage:

1. It is generally used when the value of N is small. For **larger values of N**, it is **inefficient**.

Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower than Insertion sort

Time Complexity:

- **Best Case** Sorted array as input. Or almost all elements are in proper place. [**O(N)**]. **O(1)** swaps.
- **Worst Case**: Reversely sorted / Very few elements are in proper place. [**$O(N^2)$**] . **$O(N^2)$** swaps.
- **Average Case**: [**$O(N^2)$**] . **$O(N^2)$** swaps.

Space Complexity: A temporary variable is used in swapping [auxiliary, **O(1)**]. Hence it is In-Place sort.

Advantage:

1. It is the simplest sorting approach.
2. Best case complexity is of **O(N)** [for optimized approach] while the array is sorted.
3. Using **optimized approach**, it **can detect already sorted array in first pass** with time complexity of **O(N)**.
4. Stable sort: does not change the relative order of elements with equal keys.
5. In-Place sort.

Disadvantage:

1. Bubble sort is comparatively slower algorithm.

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Time Complexity:

- **Best Case** [$O(N^2)$]. And **O(1)** swaps.
- **Worst Case**: Reversely sorted, and when the inner loop makes a maximum comparison. [$O(N^2)$] . Also, **O(N)** swaps.
- **Average Case**: [$O(N^2)$] . Also **O(N)** swaps.

Space Complexity: [auxiliary, **O(1)**]. In-Place sort. (When elements are shifted instead of being swapped (i.e. temp=a[min], then shifting elements from ar[i] to ar[min-1] one place up and then putting a[i]=temp). If swapping is opted for, the algorithm is not In-place.)

What are the loop invariants in Bubble and Selection sort?

How do we evaluate algorithms?

EFFICIENCY OF PROGRAMS

- computers are fast and getting faster – so maybe efficient programs don't matter?
  - but data sets can be very large (e.g., in 2014, Google served 30,000,000,000,000 pages, covering 100,000,000 GB – how long to search brute force?)
  - thus, simple solutions may simply not scale with size in acceptable manner
- how can we decide which option for program is most efficient?

- separate **time and space efficiency** of a program
- tradeoff between them:
  - can sometimes pre-compute results are stored; then use "lookup" to retrieve (e.g., memoization for Fibonacci)
  - will focus on time efficiency

WANT TO UNDERSTAND EFFICIENCY OF PROGRAMS

Challenges in understanding efficiency of solution to a computational problem:

- a program can be **implemented in many different ways**
- you can solve a problem using only a handful of different **algorithms**
- would like to separate choices of implementation from choices of more abstract algorithm

HOW TO EVALUATE EFFICIENCY OF PROGRAMS

- measure with a **timer**
- **count** the operations
- abstract notion of **order of growth**

will argue that this is the most appropriate way of assessing the impact of choices of algorithm in solving a problem; and in measuring the inherent difficulty in solving a problem

TIMING PROGRAMS IS INCONSISTENT

- GOAL: to evaluate different algorithms
- running time **varies between algorithms** ✓
- running time **varies between implementations** ✗
- running time **varies between computers** ✗
- running time is **not predictable** based on small inputs ✗

- time varies for different inputs but cannot really express a relationship between inputs and time ✗

COUNTING OPERATIONS

- assume these steps take **constant time**:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32

def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

3 ops

1 op

loop x times

1 op

2 ops

mysum → 1+3x ops

COUNTING OPERATIONS IS BETTER, BUT STILL...

- GOAL: to evaluate different algorithms
- count **depends on algorithm** ✓
- count **depends on implementations** ✗
- count **independent of computers** ✓
- no clear definition of **which operations** to count ✗

- count varies for different inputs and can come up with a relationship between inputs and the count

STILL NEED A BETTER WAY

- timing and counting **evaluate implementations**
- timing **evaluates machines**

- want to **evaluate algorithm**
- want to **evaluate scalability**
- want to **evaluate in terms of input size**

STILL NEED A BETTER WAY

- Going to focus on idea of counting operations in an algorithm, but not worry about small variations in implementation (e.g., whether we take 3 or 4 primitive operations to execute the steps of a loop)
- Going to focus on how algorithm performs when size of problem gets arbitrarily large
- Want to relate time needed to complete a computation, measured this way, against the size of the input to the problem
- Need to decide what to measure, given that actual number of steps may depend on specifics of trial

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- a function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- when `e` is **first element** in the list → BEST CASE
- when `e` is **not in list** → WORST CASE
- when **look through about half** of the elements in list → AVERAGE CASE
- want to measure this behavior in a general way

BEST, AVERAGE, WORST CASES

- suppose you are given a list `L` of some length `len(L)`
- **best case**: minimum running time over all possible inputs of a given size, `len(L)`
  - constant for `search_for_elmt`
  - first element in any list
- **average case**: average running time over all possible inputs of a given size, `len(L)`
  - practical measure

generally will focus on this case

- **worst case**: maximum running time over all possible inputs of a given size, `len(L)`
  - linear in length of list for `search_for_elmt`
  - must search entire list and not find it

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

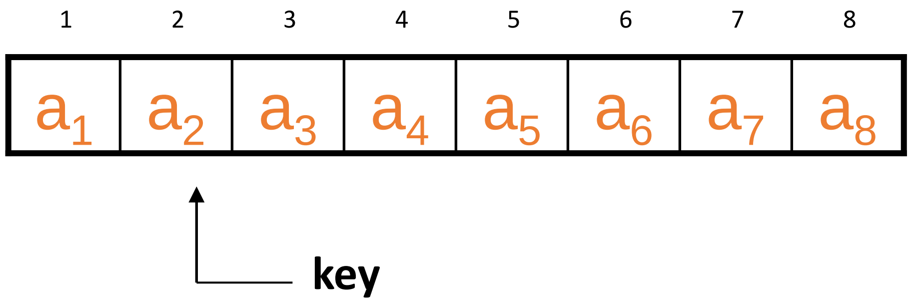
Bottom line idea on growth of algorithmic time

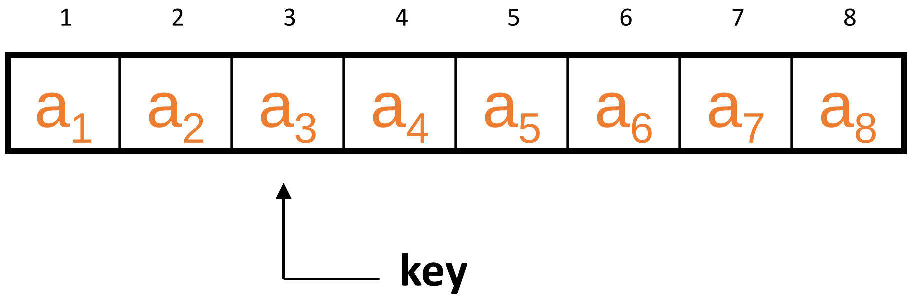
- A+B ln (n) < A+B ln(n) + C n < …. + D(n^2) for large enough n (B, C, D are positive constants, depends on machine etc.

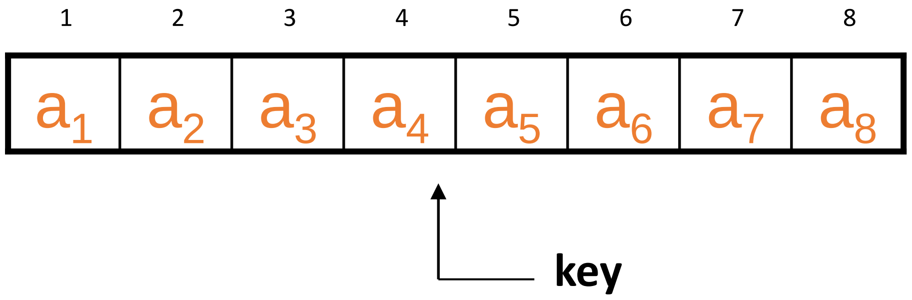
Some References

- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec10.pdf
- https://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt

Detailed Analysis of Insertion Sort







Analysis of Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted "**while** i > 0 and A[i] > key"
  - $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$

  $(c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always $A[i] > key$ in **while** loop test
  - Have to compare key with all elements to the left of the j-th position
    $\Rightarrow$ compare with j-1 elements $\Rightarrow t_j = j$

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$$= an^2 + bn + c$$ a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

≈ $n^2/2$ comparisons

≈ $n^2/2$ exchanges

Summary

- When you add all these terms, the terms proportional to n-square will win !

Growth of Functions. Given functions f and g, we wish to show how to quantify the statement : "g grows as fast as f".

The growth of functions is directly related to the complexity of algorithms. We are guided by the following principles.

- We only care about the behavior for "large" problems.
- We may ignore implementation details such as loop counter incrementation.

Let f and g be functions from the natural numbers to the real numbers. Then g **asymptotically dominates** *f, or*

f is big-O of g

if there are positive constants C and k such that

$$|f(x)| \leq C|g(x)| \text{ for } x \geq k.$$

Usually we will deal with functions that are manifestly positive at least for "large values of x".

If f is big-O of g, then we write

$f(x)$ is $O(g(x))$ or $f \in O(g)$.

Example:

Show that $x^2 + 10$ *is* $O(x^2)$.

Let $C = 3$ and $k = 3$.
Then, if $x \geq 3, 3x^2 = x^2 + 2x^2 \geq x^2 + 2 \cdot 3^2 \geq x^2 + 10$

Or,

Let $C = 2$ and $k = 4$. Then, if $x \geq 4$,

$2x^2 = x^2 + x^2 \geq x^2 + 4^2 \geq x^2 + 10$.

So the values of C and k are flexible.

Also note that $3x^2 = O(x^2)$

Clearly some general theorems are useful.

THEOREMS: (Without Proof).

$$If \lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = L,\ where\ L \geq 0,\ then\ f \in O(g).$$

$$If \lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = \infty,\ then\ f\ is\ \mathbf{not}\ O(g)\ (f \notin O(g)).$$

$$\lim_{x\to\infty} \frac{x^2+10}{x^2} = \lim_{x\to\infty}\left(1+\frac{10}{x^2}\right) = 1+0 = 1.$$

$x^2 + 10 \in O(x^2)$.

How do you interpret the statement $f \text{ is not } O(g)$? That is, how do you negate the definition? The definition says:

> $f \in O(g)$ if and only if there exist constants C and k such that, for all x, if $x \geq k$, then $|f(x)| \leq C|g(x)|$.

The negation would then read:

> f *is not* $O(g)$ if and only if for all constants C and k, there exist x such that $x \geq k$ and $|f(x)| > C|g(x)|$.

Show that x^2 *is not* $O(x)$*. (Prove it)*

Show that $2x^3 + x^2 - 3x + 2$ *is* $O(x^3)$.

In general,

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant.

Then

If k>=d, p(n) is $O(x^k)$

A polynomial of degree n is $O(x^n)$.

How about log(x) vs x ?

We already know that x is NOT O(log x); easy to show that log x = O(x)

Definition of small o(): If f and g are such that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

then we say f *is* **little-***o* *of* g*, written*

$$f \in o(g).$$

Theorem: *If* f *is* $o(g)$*, then* f *is* $O(g)$.

Useful results:

If f_1 *is* $O(g_1)$ *and* f_2 *is* $O(g_2)$*, then* (f_1+f_2) *is* $O(max\{|g_1|, |g_2|\})$.

If f_1 *and* f_2 *are both* $O(g)$*, then* $(f_1 + f_2)$ *is* $O(g)$.

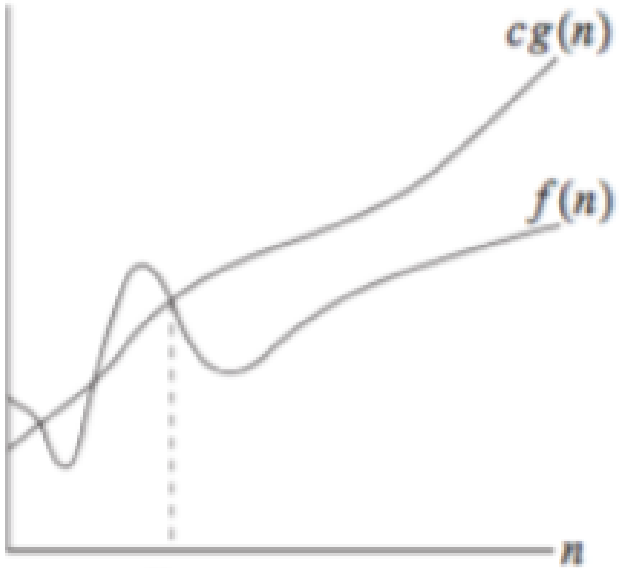
If f_1 *is* $O(g_1)$ *and* f_2 *is* $O(g_2)$*, then* (f_1f_2) *is* $O(g_1g_2)$.

If f_1 *is* $O(f_2)$ *and* f_2 *is* $O(f_3)$*, then* f_1 *is* $O(f_3)$.

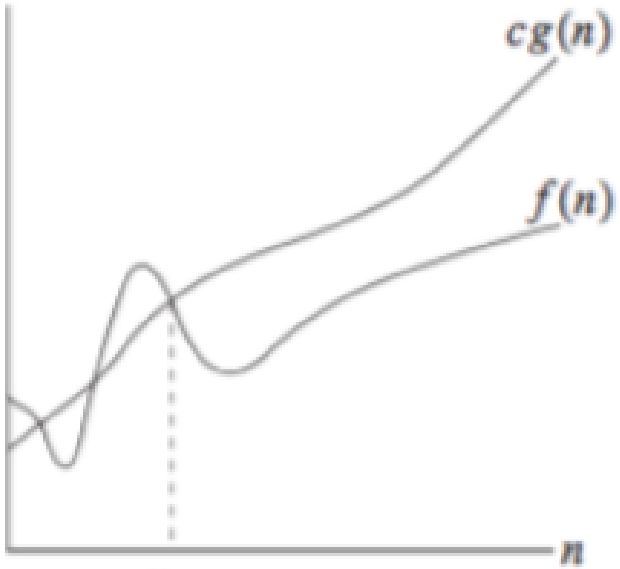
If f *is* $O(g)$*, then* (af) *is* $O(g)$ *for any constant* a.

22Nov2021

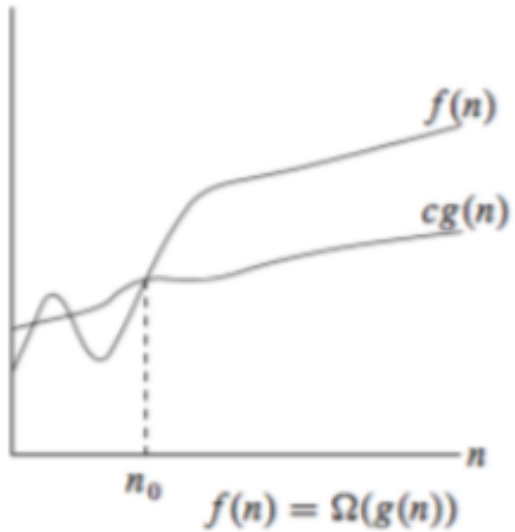
- Growth of Functions
- Divide and Conquer algorithms
- Merge sort
- Recurrence relation for Merge sort.
- Solving recurrence relations.



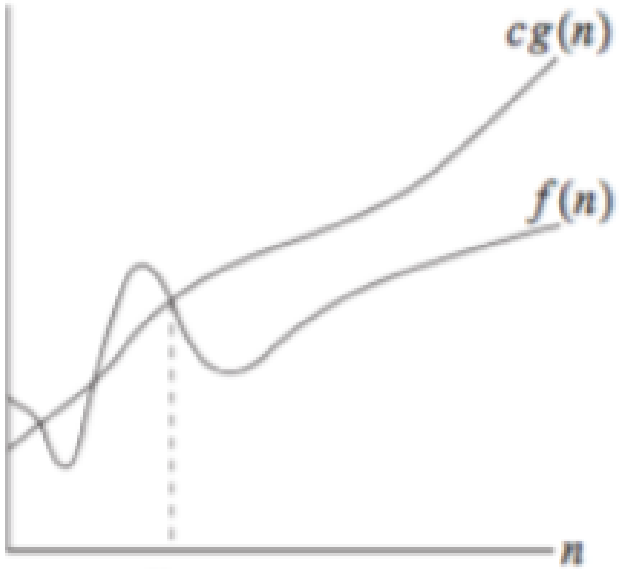
$f(n) = O(g(n))$



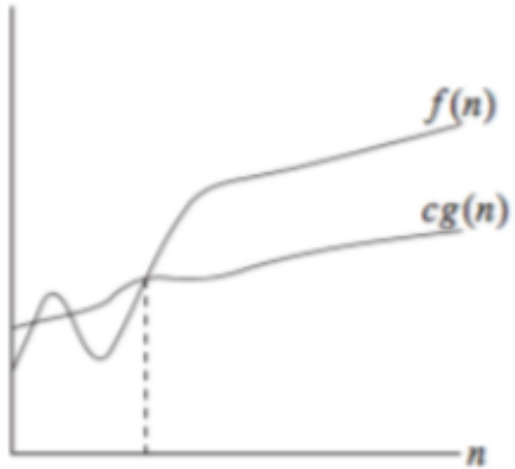
$f(n) = O(g(n))$



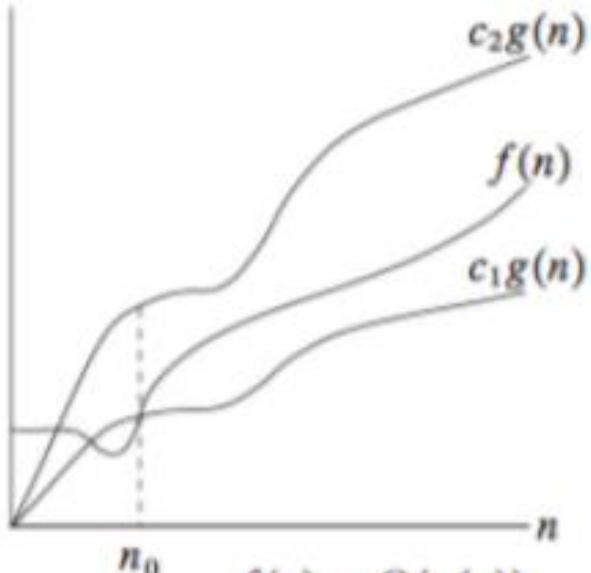
$f(n) = \Omega(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

Let's say we want to solve a problem P. For e.g. P could be the problem of sorting an array, or finding the smallest element in an array. Divide-and-conquer is an approach that can be applied to any P and goes like this:

Divide-and-Conquer

To Solve P:

1. *Divide* P into two smaller problems P_1, P_2.
2. *Conquer* by solving the (smaller) subproblems recursively.
3. *Combine* solutions to P_1, P_2 into solution for P.

The simplest way is to divide into *two* subproblems. Can be extended to divide into k subproblems.

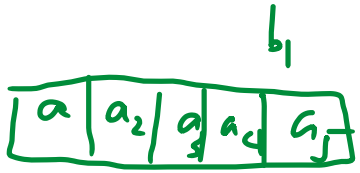
Analysis of divide-and-conquer algorithms and in general of recursive algorithms leads to recurrences.

MERGE SORT

A divide-and-conquer solution for sorting an array gives an algorithm known as mergesort:

- Mergesort:
  - Divide: Divide an array of n elements into two arrays of $n/2$ elements each.
  - Conquer: Sort the two arrays recursively.
  - Combine: Merge the two sorted arrays.
- Assume we have procedure Merge(A, p, q, r) which merges sorted A[p..q] with sorted A[q+1....r]
- We can sort A[p...r] as follows (initially p=0 and r=n-1):

```
Merge Sort(A,p,r)
    If p < r then
        q = ⌊(p + r)/2⌋
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)
```



a_1 a_2 a_3 a_4 5_1 5

n operations

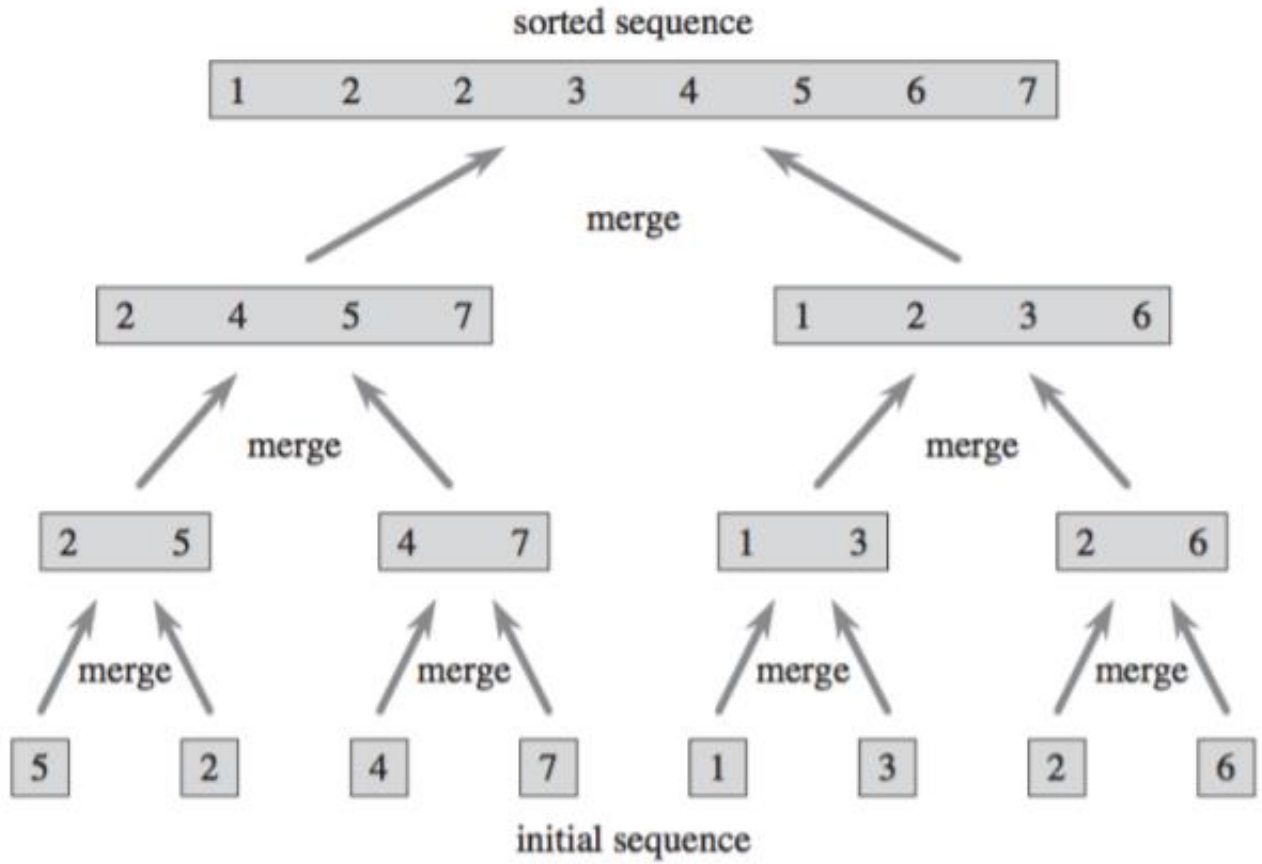
Suppose you are sorting A = [5, 2, 4, 7, 1, 3, 2, 6].
We divide this into the arrays [5, 2, 4, 7] and [1, 3, 2, 6], and MergeSort each of those arrays.
To sort [5, 2, 4, 7], we divide it into the arrays [5, 2] and [4, 7], and MergeSort each of those arrays.
To sort [5, 2], we divide it into the arrays [5] and [2].

At this point we have reached the base case, so MergeSorting the array [5] just returns the array [5], and MergeSorting the array [2] just returns the array [2].

But now we need to merge together [5] and [2], which gives us [2, 5].

Merging together [2, 5] and [4, 7] gives us [2, 4, 5, 7].

Finally, merging together [2, 4, 5, 7] and [1, 2, 3, 6] gives us [1, 2, 2, 3, 4, 5, 6, 7].

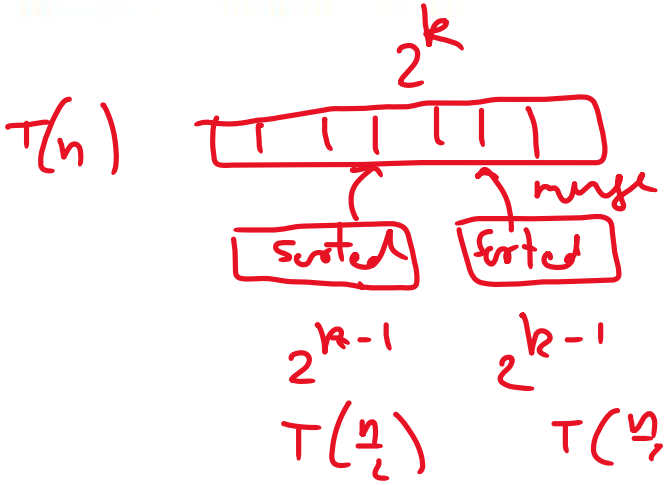


The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Mergesort Analysis

- To simplify things, let us assume that n is a power of 2, i.e $n = 2^k$ for some k.
- Running time of a recursive algorithm can be analyzed using a **recurrence relation**. Each "divide" step yields two sub-problems of size $n/2$.
- Let $T(n)$ denote the worst-case running time of mergesort on an array of n elements. We have:

$$\begin{aligned} T(n) &= c_1 + T(n/2) + T(n/2) + c_2 n \\ &= 2T(n/2) + (c_1 + c_2 n) \end{aligned}$$

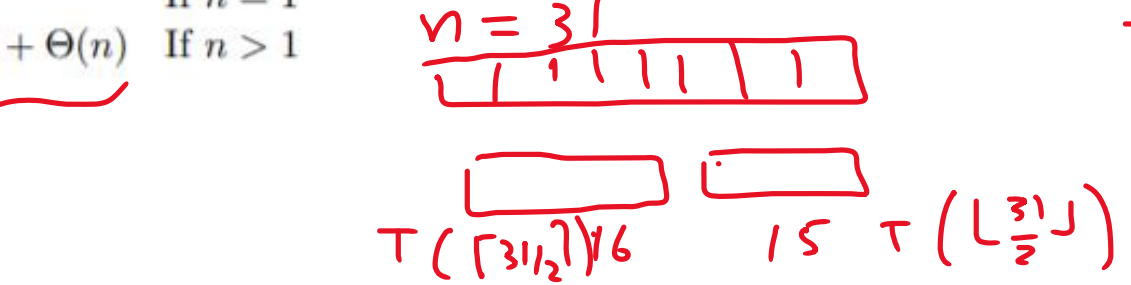


$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \text{cost of merging}$$

$$\text{cost of merging} = c_2 n + c_1$$

- Simplified, $T(n) = 2T(n/2) + \Theta(n)$
- Note: If $n \neq 2^k$ the recurrence gets more complicated.

$$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{If } n > 1 \end{cases}$$



Substitution method of Solving Recurrence Relations:.

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works.

Lets try to solve a recurrence relations using this.

$$T(1) = 1 \text{ and } T(n) = 2T(\lfloor n/2 \rfloor) + n \text{ for } n > 1.$$

$T(n) < cn \log n$ for some c when $n > n_0$.

Substitution method of Solving Recurrence Relations:.

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works.

Lets try to solve a recurrence relations using this.

$$T(1) = 1 \text{ and } T(n) = 2T(\lfloor n/2 \rfloor) + n \text{ for } n > 1.$$

We guess that the solution is $T(n) = O(n \log n)$. So we must prove that $T(n) \leq cn \log n$ for some constant c. (We will get to n_0 later, but for now let's try to prove the statement for all $n \geq 1$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{for } c \geq 1) \end{aligned}$$

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{for } c \geq 1) \end{aligned}$$

Now we need to show the base case. This is tricky, because if $T(n) \leq cn \log n$, then $T(1) \leq 0$, which is not a thing. So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{for } c \geq 1) \end{aligned}$$

Now we need to show the base case. This is tricky, because if $T(n) \leq cn \log n$, then $T(1) \leq 0$, which is not a thing. So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.)

We choose $n = 2$ and $n = 3$ for our base cases because when we expand the recurrence formula, we will always go through either $n = 2$ or $n = 3$ before we hit the case where $n = 1$.

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{for } c \geq 1) \end{aligned}$$

So proving the inductive step as above, plus proving the bound works for $n = 2$ and $n = 3$, suffices for our proof that the bound works for all $n > 1$.

Plugging the numbers into the recurrence formula, we get $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. So now we just need to choose a c that satisfies those constraints on $T(2)$ and $T(3)$. We can choose $c = 2$, because $4 \leq 2 \cdot 2 \log 2$ and $5 \leq 2 \cdot 3 \log 3$.

Therefore, we have shown that $T(n) \leq 2n \log n$ for all $n \geq 2$, so $T(n) = O(n \log n)$.

Master Method

- It is possible to come up with a formula for recurrences of the form $T(n) = aT(n/b) + n^c$ ($T(1) = 1$). This is called the *master method*.
  - Merge-sort $\Rightarrow T(n) = 2T(n/2) + n$ ($a = 2, b = 2$, and $c = 1$).

$$T(n) = aT\left(\frac{n}{b}\right) + n^c \quad a \geq 1, b \geq 1, c > 0$$

$$\Downarrow$$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^c \\ \Theta(n^c \log_b n) & a = b^c \\ \Theta(n^c) & a < b^c \end{cases}$$

Other recurrences

Some important/typical bounds on recurrences not covered by master method:

- Logarithmic: $\Theta(\log n)$
  - Recurrence: $T(n) = 1 + T(n/2)$
  - Typical example: Recurse on half the input (and throw half away)
  - Variations: $T(n) = 1 + T(99n/100)$
- Linear: $\Theta(N)$
  - Recurrence: $T(n) = 1 + T(n-1)$
  - Typical example: Single loop
  - Variations: $T(n) = 1 + 2T(n/2), T(n) = n + T(n/2), T(n) = T(n/5) + T(7n/10 + 6) + n$
- Quadratic: $\Theta(n^2)$
  - Recurrence: $T(n) = n + T(n-1)$
  - Typical example: Nested loops
- Exponential: $\Theta(2^n)$
  - Recurrence: $T(n) = 2T(n-1)$

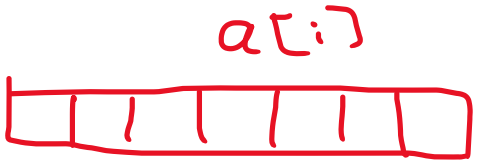
Ref

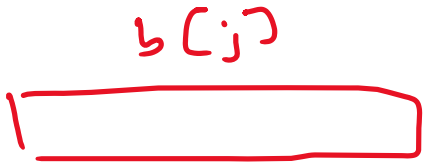
- https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture1.pdf
- https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture2.pdf
- https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture3.pdf
- https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture2.pdf
- https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/fall14/Lectures/02-recurrences/recurrences.pdf

24Nov2021

- Last time we discussed the time complexity of merge – sort.
- The complexity is O(n log n).
- We did this assuming that the time complexity of merging two sorted arrays of length m and n into a sorted array of length m+n is O(m+n)
- We will complete this today.

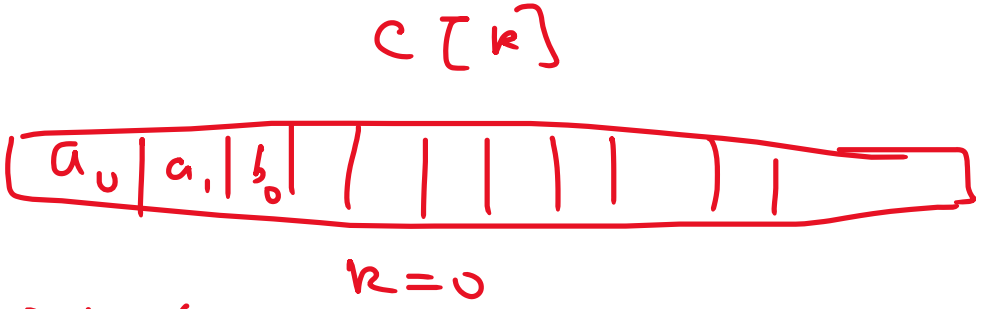
- Input: sorted array a of length m, and sorted array b of length n
- Create an empty array c of length m+n, set index_a and index_b to 0
- While (index_a < length of a) and (index_b < length of b)
    a. Add the smaller of a[index_a] and b[index_b] to the end of c.
    b. Increment the index of c.
    c. Increment the index of the list with the smaller element
- If any elements are left over in a or b, add them to the end of c, in order
- Return c.





$0 \le i \le m-1$

$0 \le j \le n-1$



$0 \le k \le m+n-1$

$i=0, j=0$

if $a_0 < b_0$ $\quad c[0] = a[0] \quad k=1$.

$i=1, j=0, k=1,$ if $a_1 < b_0$ $\quad c[1] = a_1$

$k=2$

$i=2, j=0$

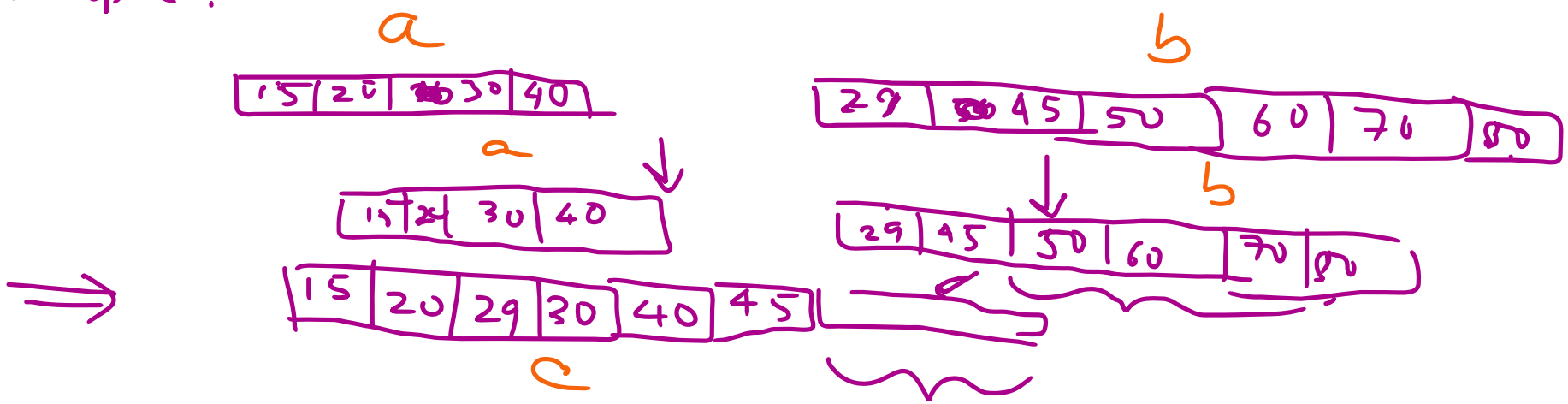
$i=2, j=0$ if $b_0 < a_2$ $\quad c[2] = b_0 \quad [k = i+j]$

$i=2, j=1$

$k=3$

So, ultimately $a[i]$ or $b[j]$ will get exhausted. Then $c[k]$ will just copy the rest.

Example.


a
15 20 30 40
a
15 30 40
15 20 29 30 40 45
c
b
29 45 50 60 70 80
b
29 45 50 60 70 80

Timing Analysis:

Let $m = n$

WLOG. Let a_0 be lenThan b_0.

Let $T_{ab}(m,n)$ be the time complexity for m, n arrays

Then obviously $T(n,n) = T(n-1,n) + K$

where K is time taken to compare two elements & write to C

$$T(n,n) = T(n-i,n-j) + K(i+j)$$

$$= \begin{cases} T(0, n-l) \\ \text{or } T(n-l, 0) \end{cases} + k(n+l)$$

Similarly for $T(m,n)$ $\Rightarrow T(n,n) = c(m+n) = O(n)$

Looking at it another way

$T(m,n) =$ either $T(m-1,n) +$ constant

or $T(m,n-1) +$ constant

Assume $T(m,n) \leq c(m+n)$ for $m \geq n_0$ & some c
$n \geq n_0$
& upto $m-1$, $n-1$

is $O(m+n)$

Then for

$T(m,n) = T(m-1,n) + K$ (WLOG)

$\leq c(m-1+n)$

$\leq c(m+n)$

$\Rightarrow T(m,n) = O(m+n)$.

```
#include <stdio.h>
#include <stdlib.h>
int merge_two_sorted_arrays(int arr1[], int arr2[], int arr3[], int
m, int n)
{
int i,j,k;
i = j = k = 0;
for(i=0;i < m && j < n;)
{
if(arr1[i] < arr2[j])
{
arr3[k] = arr1[i];
k++;
i++;
}
else
{
arr3[k] = arr2[j];
k++;
j++;
}
}
while(i < m)
{
arr3[k] = arr1[i];
k++;
i++;
}
while(j < n)
{
arr3[k] = arr2[j];
k++;
j++;
}
}
```

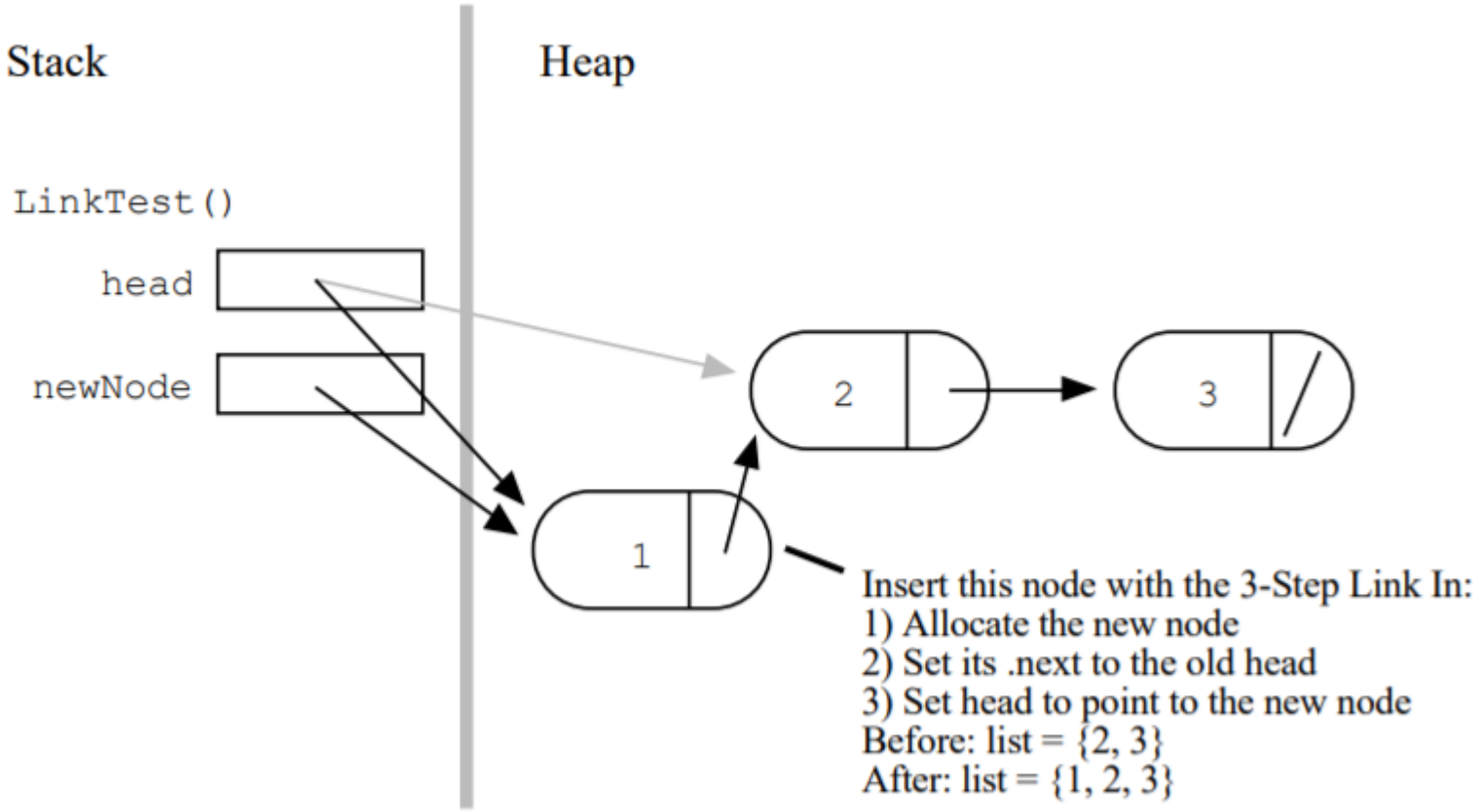
29NOv2021

- Back to Important Structures
  - Link List
  - Stack
  - Binary Trees
  - Etc.

- Adding a node at the header of a link list.

(Ref. http://cslibrary.stanford.edu/103/LinkedListBasics.pdf)

Adding a node at the beginning of the code



Adding a link to the head

- Wrong code. Why?

```
struct node {
   int            data;
   struct node*   next;
};
```

```
void WrongPush(struct node* head, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = head;
   head = newNode;      // NO this line does not work!
}
```

```
void WrongPushTest() {
   List head = BuildTwoThree();

   WrongPush(head, 1);  // try to push a 1 on front -- doesn't work
}
```

Adding a node to the head of a linked list

```
void WrongPush(struct node* head, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = head;
   head = newNode;       // NO this line does not work!
}


void WrongPushTest() {
   List head = BuildTwoThree();

   WrongPush(head, 1);  // try to push a 1 on front -- doesn't work
}
```

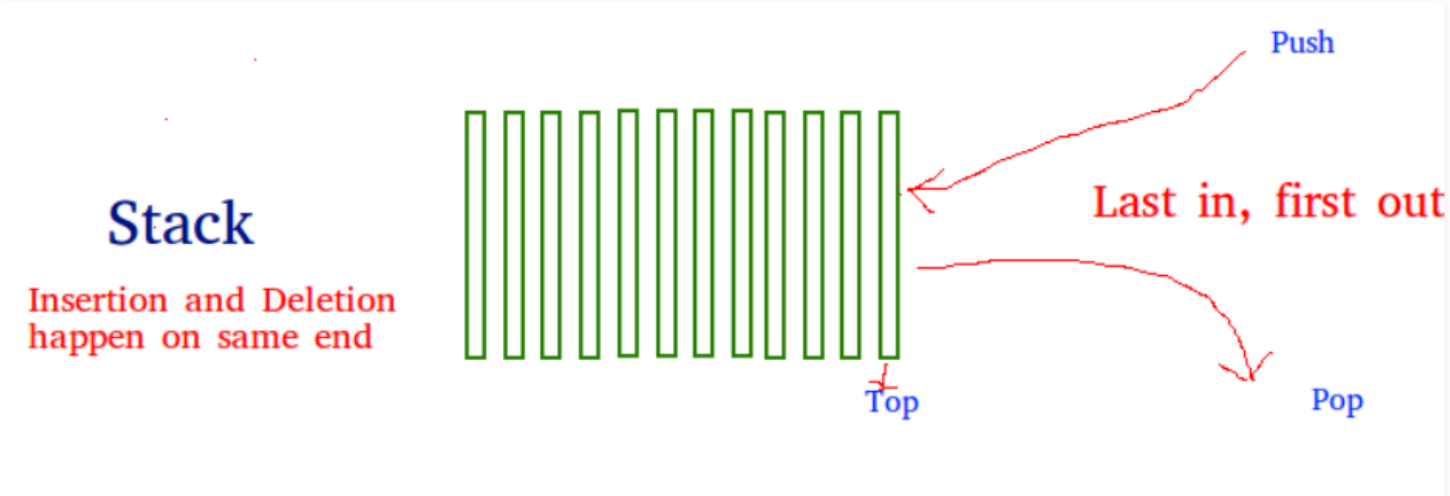
```
/*
 Takes a list and a data value.
 Creates a new link with the given data and pushes
 it onto the front of the list.
 The list is not passed in by its head pointer.
 Instead the list is passed in as a "reference" pointer
 to the head pointer -- this allows us
 to modify the caller's memory.
*/
void Push(struct node** headRef, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = *headRef;  // The '*' to dereferences back to the real head
   *headRef = newNode;        // ditto
}


void PushTest() {
   struct node* head = BuildTwoThree();// suppose this returns the list {2, 3}

   Push(&head, 1);      // note the &
   Push(&head, 13);

   // head is now the list {13, 1, 2, 3}
}
```

Stack as abstract data type

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

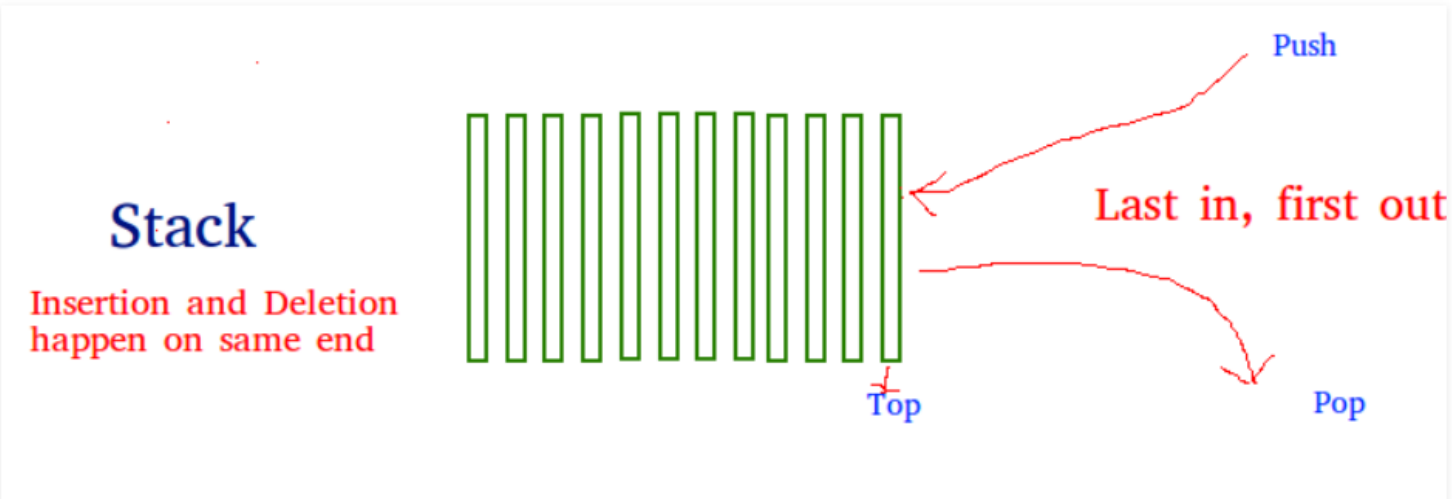


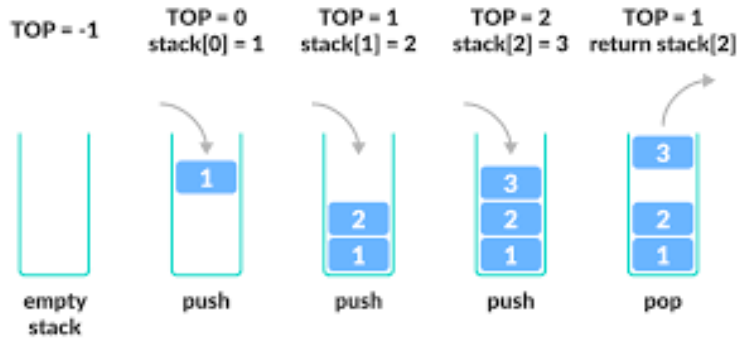
Stack as abstract data type

Stack is a linear data structure which follows a particular order in which the operations are performed.
The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.



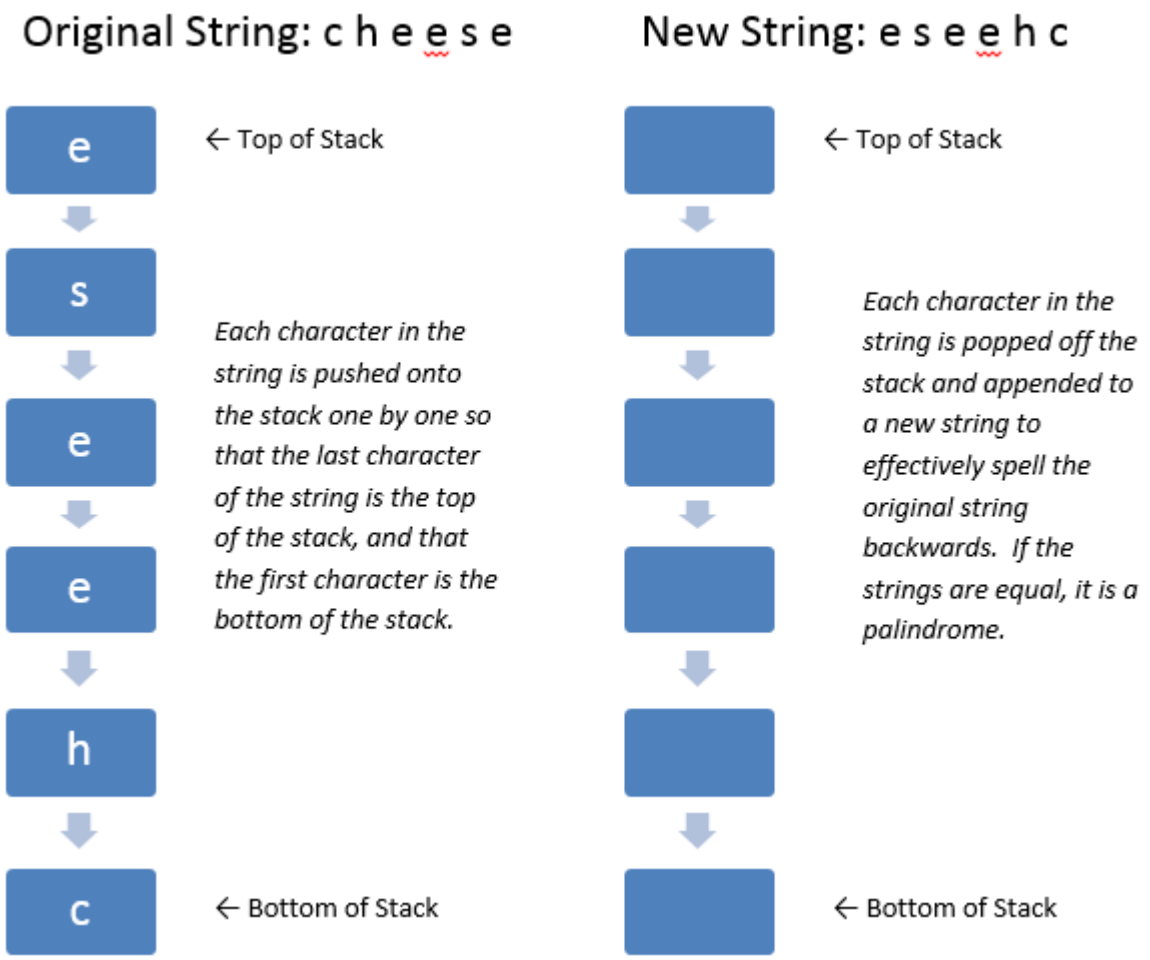


How to implement a Stack

- Can be implemented as array
  - Needs maximum size of the stack to create an empty stack, then all the functions.
- Can be implemented as a linked list
  - Create a single node.
  - Push is attaching a node to the top
  - Pop is reading value at the top and then deleting it from the top (head)
- How will you write Is_full and Is_empty?

Reversing a string using stack and checking if it is a palindrome

1) Create an empty stack.
2) One by one push all characters of string to stack.
3) One by one pop all characters from stack and put them back to string.



Stack implemented as an array

- A bounded stack can be implemented as array
- We have to know the maximum size
- We have to have an indication when the stack is empty (or full)
- We need to store the data in the stack.

So how to we define a stack structure consistent with these?

Stack structure implemented as array

```
// Data structure for stack
struct stack
{
   int maxsize;    // define max capacity of stack
   int top;    //set top to -1 for an empty stack
   int *items; // can also write items[]?? items[maxsize]??
};
```

Functions related to stack implemented as array

```
// Utility function to initialize stack
struct stack* newStack(int capacity)
{
    struct stack *pt = (struct stack*)malloc(sizeof(struct stack));

    pt->maxsize = capacity;
    pt->top = -1;
    pt->items = (int*)malloc(sizeof(int) * capacity);

    return pt;
}
```

Functions related to stack implemented as array

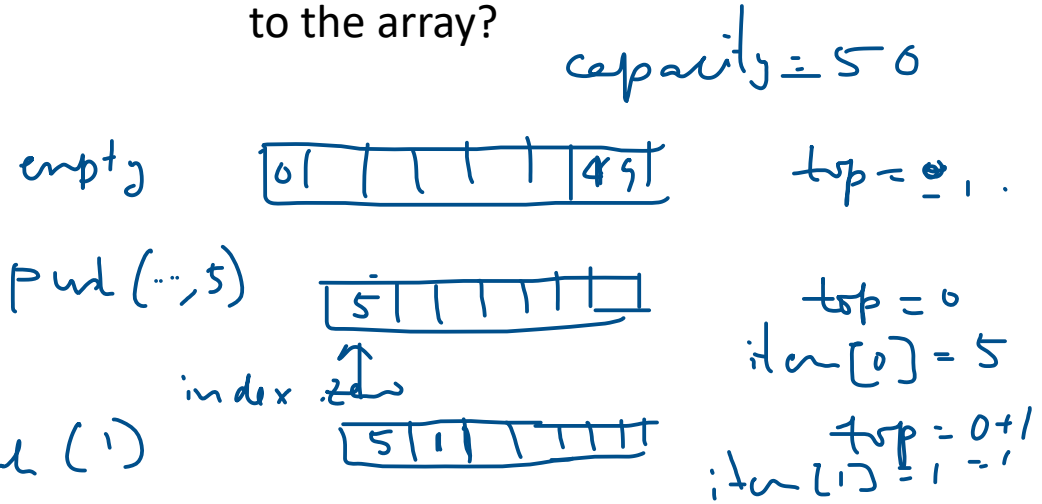
```
// Utility function to initialize stack
struct stack* newStack(int capacity)
{
    struct stack *pt = (struct
stack*)malloc(sizeof(struct stack));

    pt->maxsize = capacity;
    pt->top = -1;
    pt->items = (int*)malloc(sizeof(int) * capacity);

    return pt;
}
```

```
// Utility function to check if the stack is empty or not
int isEmpty(struct stack *pt)
{
    return pt->top == -1;    // or return size(pt) == 0;
}

// Utility function to check if the stack is full or not
int isFull(struct stack *pt)
{
    return pt->top == pt->maxsize - 1;
// or return size(pt) == pt->maxsize;
}
```

Push and Pop an element

```
// Utility  PUSH function to add an element x in the stack
void push(struct stack *pt, int x)
{
   // check if the stack is already full. Then inserting an element
would
   // lead to stack overflow
   if (isFull(pt))
   {
      printf("OverFlow\nProgram Terminated\n");
      exit(EXIT_FAILURE);
   }

   // add an element and increments the top index
   pt->items[++pt->top] = x;
}
```

Explain how the last line works.
How is it adding to the array?
Pushed to which location or index to the array?



Push and Pop an element

```
// Utility  PUSH function to add an element x in the stack
void push(struct stack *pt, int x)
{
   // check if the stack is already full. Then inserting an element
would
   // lead to stack overflow
   if (isFull(pt))
   {
      printf("OverFlow\nProgram Terminated\n");
      exit(EXIT_FAILURE);
   }

   // add an element and increments the top index
   pt->items[++pt->top] = x;
}
```

```
// Utility function to pop top element from the stack
int pop(struct stack *pt)
{
   // check for stack underflow
   if (isEmpty(pt))
   {
      printf("UnderFlow\nProgram Terminated\n");
      exit(EXIT_FAILURE);
   }

   // decrement stack size by 1 and (optionally) return the
popped element
   return pt->items[pt->top--];
}
```

Peek at the top of the stack

```
// Utility function to return top element in a stack
int peek(struct stack *pt)
{
   // check for empty stack
   if (!isEmpty(pt))
      return pt->items[pt->top];
   else
      exit(EXIT_FAILURE);
}
```

Sample Program illustrating a stack using all these functions

```
int main()
{
   // create a stack of capacity 5
   struct stack *pt = newStack(5);

   push(pt, 1);
   push(pt, 2);
   push(pt, 3);

   printf("Top element is %d\n", peek(pt));
   printf("Stack size is %d\n", size(pt));

   pop(pt);
   pop(pt);
   pop(pt);

   if (isEmpty(pt))
      printf("Stack is empty");
   else
      printf("Stack is not empty");

   return 0;
}
```

Output:

```
Inserting 1
Inserting 2
Inserting 3
Top element is 3
Stack size is 3
Removing 3
Removing 2
Removing 1
Stack is empty
```

Using Header files to define structure as per

A Stack Implemented as an array in C

Header file for a list

```
//-------------------------------------------------------------
// File: Code120_Stack.h
// Purpose: Header file for a demonstration of a stack implemented
//              as an array.  Data type: Character
// Programming Language: C
// Author: Dr. Rick Coleman
//-------------------------------------------------------------
#ifndef CODE120_STACK_H
#define CODE120_STACK_H

#include <stdio.h>

#define MAX_SIZE 50           // Define maximum length of the stack

// List Function Prototypes
void InitStack();             // Initialize the stack
void ClearStack();            // Remove all items from the stack
int Push(char ch);            // Push an item onto the stack
char Pop();                   // Pop an item from the stack
int isEmpty();                // Return true if stack is empty
int isFull();                 // Return true if stack is full

// Define TRUE and FALSE if they have not already been defined
#ifndef FALSE
#define FALSE (0)
#endif
#ifndef TRUE
#define TRUE (!FALSE)
#endif

#endif // End of stack header
```

Using Header files to define structure as per

A Stack Implemented as an array in C

Header file for a list

```
//-------------------------------------------------------------
// File: Code120_Stack.h
// Purpose: Header file for a demonstration of a stack implemented
//              as an array.  Data type: Character
// Programming Language: C
// Author: Dr. Rick Coleman
//-------------------------------------------------------------
#ifndef CODE120_STACK_H
#define CODE120_STACK_H

#include <stdio.h>

#define MAX_SIZE 50           // Define maximum length of the stack

// List Function Prototypes
void InitStack();             // Initialize the stack
void ClearStack();            // Remove all items from the stack
int Push(char ch);            // Push an item onto the stack
char Pop();                   // Pop an item from the stack
int isEmpty();                // Return true if stack is empty
int isFull();                 // Return true if stack is full

// Define TRUE and FALSE if they have not already been defined
#ifndef FALSE
#define FALSE (0)
#endif
#ifndef TRUE
#define TRUE (!FALSE)
#endif

#endif // End of stack header
```

```
#include "Code120_Stack.h"

// Declare these as static so no code outside of this source
// can access them.
static int top; // Declare global index to top of the stack
static char theStack[MAX_SIZE]; // The stack
```

https://www.cs.uah.edu/~rcoleman/Common/CodeVault/Code/Code120_Stack.html

Application of Stack

- Checking parenthesis
- Expression handling (in fix to Pre fix or post fix)
- What else?

Check for balanced expressions

- Identify what you want to see balanced and what rules apply

e.g. Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in exp

Algorithm:

- Declare a character stack S.
- Now traverse the expression string exp.
  1. If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
  2. If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.

- After complete traversal, if there is some starting bracket left in stack then “not balanced”

Initially :

Opening bracket. Push into stack

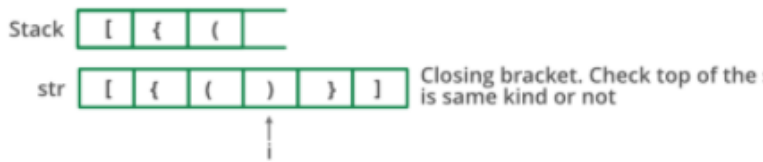
Step 1:

Opening bracket. Push into stack

Step 2:

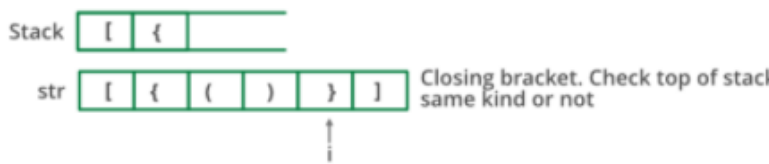
Opening bracket. Push into stack

Step 3:



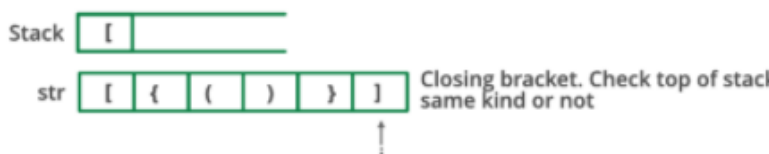
Closing bracket. Check top of the stack is same kind or not

Step 4:

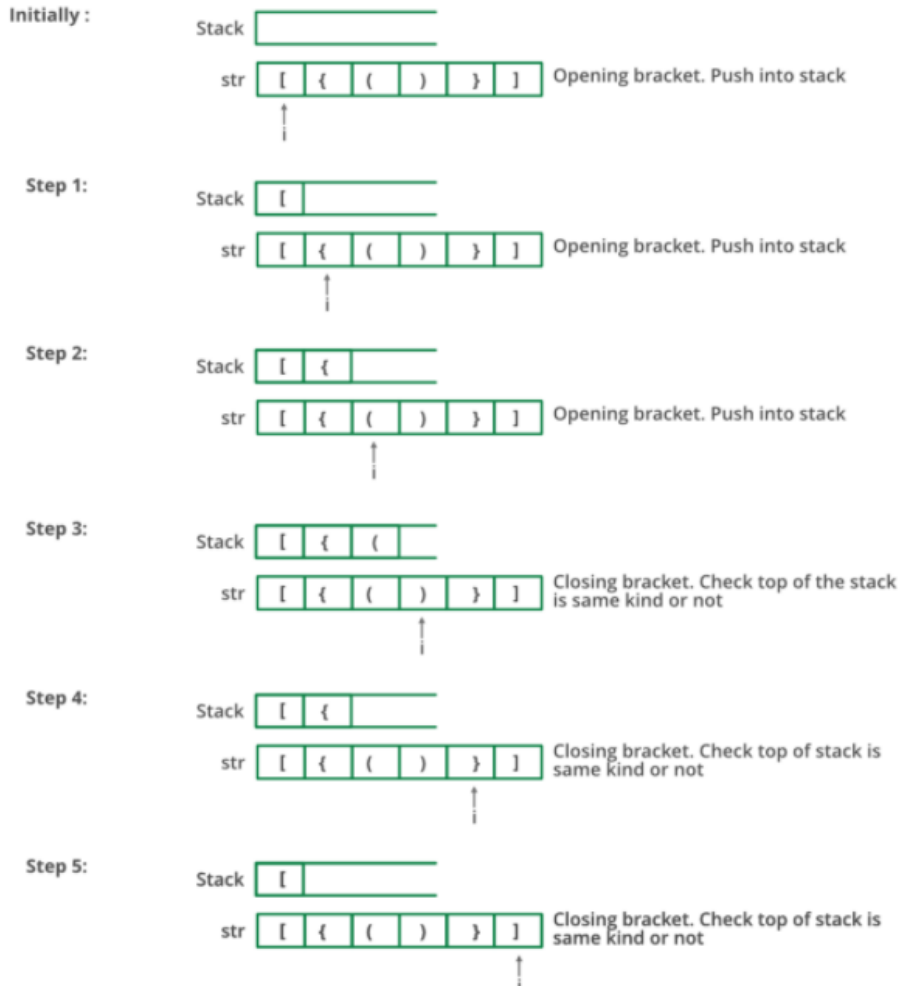


Closing bracket. Check top of stack is same kind or not

Step 5:



Closing bracket. Check top of stack is same kind or not




```
#include <stdio.h>
#include <stdlib.h>
#define bool int

// structure of a stack node
struct sNode {
                char data;
                struct sNode* next;
};

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data);

// Function to pop an item from stack
int pop(struct sNode** top_ref);

// Returns 1 if character1 and character2 are matching left
// and right Brackets
bool isMatchingPair(char character1, char character2)
{
                if (character1 == '(' && character2 == ')')
                                return 1;
                else if (character1 == '{' && character2 == '}')
                                return 1;
                else if (character1 == '[' && character2 == ']')
                                return 1;
                else
                                return 0;
}

// Return 1 if expression has balanced Brackets
bool areBracketsBalanced(char exp[])
{
                int i = 0;

                // Declare an empty character stack
                struct sNode* stack = NULL;

                // Traverse the given expression to check matching
                // brackets
                while (exp[i])
                {
```

01Dec2021

Stack structure implemented as array

```
// Data structure for stack
struct stack
{
   int maxsize;    // define max capacity of stack
   int top;    //set top to -1 for an empty stack
   int *items; // can also write items[]?? items[maxsize]??
};
```

Last time there was a discussion as to whether int *items is correct, how it knows that it is an array , and whether int items[], int items[maxsize] would work as well.

A simpler question.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *elements;
elements = malloc(50*sizeof(int));    // what does this do???
return 0;
}
```

A simpler question.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *elements;
elements = malloc(50*sizeof(int));    // what does this do???
elements[19] = 5;
printf("%d",elements[19]);
return 0;
}
```

```c
struct mystructure {
    int maxsize;
    int top;
    int *elements;

};
```

DECLARATION ONLY; NO ITILIALIZATION

```c
struct mystructure* createmyarrayStack(int size)

{
struct mystructure *newarray =  (struct mystructure*) (malloc(sizeof(struct mystructure)));

    newarray -> maxsize = size;
    newarray -> top = -1;
    newarray -> elements = malloc(newarray -> maxsize * sizeof(int));
    return newarray;
}
```

```c
int main()
{
struct mystructure* myStack;
int myStacksize = 100;
myarray = createmyarrayStack(myStacksize);
return 0;
}
```

```c
struct mystructure {
    int maxsize;
    int top;
    int *elements;

};
```

```c
struct mystructure* createmyarrayStack(int size)

{
struct mystructure *newarray =  (struct mystructure*) (malloc(sizeof(struct mystructure)));

    newarray -> maxsize = size;
    newarray -> top = -1;
    newarray -> elements = malloc(newarray -> maxsize * sizeof(int));
    return newarray;
}
```

```c
int main()
{
struct mystructure* myStack;
int myStacksize = 100;
myarray = createmyarrayStack(myStacksize);

myStack->elements[10]=7;
printf("%d", myStack->elements[10]);
return 0;
}
```

Compare stack implementations

ARRAY

```c
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

Compare stack implementations

ARRAY

```c
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

LINKED LIST

```c
// A structure to represent a stack
struct StackNode {
    int data;
    struct StackNode* next;
};

struct StackNode* newNode(int data)
{
    struct StackNode* stackNode =
      (struct StackNode*)
      malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}
```

File handling in C

- Suppose you have a “text file” in C and you want to append a line at the end.
- For example, you have a list of integers and you want to append another integer
- Or you want to sort these integers and save the sorted one in another list.
- All this calls for reading from a file. Basic Operations are opening a file for a specific purpose (read, write, append) and closing a file.

- When working with files, we need to declare a pointer of type file. This declaration is needed for communication between the file and the program.
- In other words, the pointer will store the address required to access the file

- When working with files, we need to declare a pointer of type file. This declaration is needed for communication between the file and the program.
- In other words, the pointer will store the address required to access the file
- Declaration:

```
FILE *filepointervariablename;
```

Opening a file

- The C function to open a file is fopen

Opening a file

- The C function to open a file is fopen

The **fopen() method** in C is a library function that is used to open a file to perform various operations which include reading, writing etc. along with various modes. If the file exists then the particular file is opened else a new file is created.

Syntax:

```
FILE *fopen(const char *file_name, const char *mode_of_operation);
```

Parameters: The method accepts two parameters of character type:

1. **file_name:** This is of C string type and accepts the name of the file that is needed to be opened.
2. **mode_of_operation:** This is also of C string type and refers to the mode of the file access. Below are the file access modes for C:
   i. **"r" –** Searches file. If the file is opened successfully fopen() loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen() returns NULL.
   ii. **"w" –** Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
   iii. **"a" –** Searches file. If the file is opened successfully fopen() loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
   iv. **"r+" –** Searches file. If opened successfully, fopen() loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
   v. **"w+" –** Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
   vi. **"a+" –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.

Return Value: The function is used to return a pointer to FILE if the execution succeeds else NULL is returned.

Useful to check if the file could be opened

EXAMPLE:

```
FILE *fptr;
fptr = fopen("trial1.txt","a");

  if(fptr == NULL)
  {
     printf("Error!");
     exit(1);
  }
```

Exercise adding an integer to a file containing integers

HW

- Study the following functions

  fscanf, fgets, fgetc, fputc ….

- fgetc() function is a file handling function in C programming language which is used to read a character from a file. It reads single character at a time and moves the file pointer position to the next address/location to read the next character.

HW:

Sort the numbers in a file containing integers separated by a blank or a newline and save it into a sorted file.

8Dec

HW: Has anyone tried this??

Sort the numbers in a file containing integers separated by a blank or a newline and save it into a sorted file.

HW: Has anyone tried this??

Sort the numbers in a file containing integers separated by a blank or a newline and save it into a sorted file.

- Open the File to read:
- Store the numbers in an array or a linked list.
- Sort the array or the linked list as you find new numbers – What sort program will you use here?
- End with End of File.
- WRITE THIS PROGRAM – not so easy, neither too difficult

Two Dimensional Arrays.

- A way to store a m x n matrix

```
data_type array_name[x][y];
```

`Example: int x[10][20];`

Can store a 10x20 matrix whose elements are integers.

Is there another way to store matrices for practical purposes.

- Many applications deal with very very large matrices most of whose elements are zero.

Matrix Operations for Large but Sparse Matrix

- Sparse matrix is a matrix (typically A VERY LARGE matrix) most of whose elements are zero.
- The idea is to represent such matrices with a data structure that retains only the non zero elements.
- The challenge is to find such a representation and develop time efficient functions such that the “sparse representation” or sparsity is maintained through out.

Unstructured Sparse Matrices

Airline flight matrix.

- airports are numbered 1 through n
- flight(i,j) = list of nonstop flights from airport i to airport j
- n = 1000 (say)
- n x n array of list references => 4 million bytes
- total number of flights = 20,000 (say)
- need at most 20,000 list references => at most 80,000 bytes

Unstructured Sparse Matrices

Web page matrix.

- web pages are numbered 1 through n
- web(i,j) = number of links from page i to page j

Web analysis.

- authority page … page that has many links to it
- hub page … links to many authority pages

Web Page Matrix

- n = 2 billion (and growing by 1 million a day)
- n x n array of ints => $16 * 10^{18}$ bytes ($16 * 10^{9}$ GB)
- each page links to 10 (say) other pages on average
- on average there are 10 nonzero entries per row
- space needed for nonzero elements is approximately 20 billion x 4 bytes = 80 billion bytes (80 GB)

Unstructured Sparse Matrices

A typical representation

- Define a structure that has a tuplet array
  - Each array element is a tuplet[N] (row, column, value)
  - We can use the zero element of the array to retain some key information about the original matrix
  - E.g. tuplet[0] = {no. of rows, no. of columns, no. of nonzero elements Are we saving any space? Only if sparsity <1/3
  - p = number of non zero elements, p <= (mn)/3 for a m x n matrix

Representation Of Unstructured Sparse Matrices

Single linear list in row-major order.

- scan the nonzero elements of the sparse matrix in row-major order
- each nonzero element is represented by a triple

  (row, column, value)
- the list of triples may be an array list or a linked list (chain)

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	-15
row 1	0	11	3	0	0	0
row 2	0	0	0	-6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

a[0].row: number of rows of the matrix

a[0].col: number of columns of the matrix

a[0].value: number of nonzero entries

The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

a[0].row: number of rows of the matrix

a[0].col: number of columns of the matrix

a[0].value: number of nonzero entries

The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Retrieving a row information vs Retrieving a column

How many non zero element are in a row or a column?

To do that for a row is simple. Complexity?

To do that that a column is not so simple Complexity?

a[0].row: number of rows of the matrix

a[0].col: number of columns of the matrix

a[0].value: number of nonzero entries

The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Retrieving a row vs Retrieving a column

How many non zero elements are there in a specific row or a specific column?

To do that for a row is simple. '=
Complexity O(p) = O(mn)

To do that that a column is not so simple
Complexity O(p*n) = O(mn^2), But we can devise an algorithm to convert it to O(mn) { we will not prove it here}

a[0].row: number of rows of the matrix

a[0].col: number of columns of the matrix

a[0].value: number of nonzero entries

The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

- Any important problem in dealing with sparse matrix is how to transpose a sparse matrix by keeping its properties intact. We will not discuss this here except to note that there are algorithms to do this efficiently.

Multiplying Polynomials.

- How do we represent polynomials?
- How do we add? What is the time complexity?
- How do we multiply polynomials?

The Polynomial Multiplication Algorithm

The Polynomial Multiplication Algorithm

Problem:

Given two polynomials of degree n

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_n x^n, \end{aligned}$$

compute the product $A(x)B(x)$.

The Polynomial Multiplication Algorithm

Problem:

Given two polynomials of degree n

$$A(x) = a_0 + a_1 x + \cdots + a_n x^n$$
$$B(x) = b_0 + b_1 x + \cdots + b_n x^n,$$

compute the product $A(x)B(x)$.

Implement each polynomial as an array of the coefficients

- What is the complexity in terms of the degree of the polynomial ?

- What is the complexity in terms of the degree of the polynomial ?

The Direct (Brute Force) Approach

Let $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$.

Set $C(x) = \sum_{k=0}^{2n} c_i x^i = A(x)B(x)$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

for all $0 \leq k \leq 2n$.

- What is the complexity in terms of the degree of the polynomial ?

The Direct (Brute Force) Approach

Let $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$.

Set $C(x) = \sum_{k=0}^{2n} c_i x^i = A(x)B(x)$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

for all $0 \leq k \leq 2n$.

The direct approach is to compute all c_k using the formula above. The total number of multiplications and additions needed are $\Theta(n^2)$ and $\Theta(n^2)$ respectively. Hence the complexity is $\Theta(n^2)$.

Divide and Conquer – First Try

The Divide Step: Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1},$$

$$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}.$$

Then $A(x) = A_0(x) + A_1(x) x^{\lfloor \frac{n}{2} \rfloor}$.

Similarly we define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\lfloor \frac{n}{2} \rfloor}.$$

Divide and Conquer – First Try

The Divide Step: Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1},$$

$$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}.$$

Then $A(x) = A_0(x) + A_1(x) x^{\lfloor \frac{n}{2} \rfloor}$.

Similarly we define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\lfloor \frac{n}{2} \rfloor}.$$

Then

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}.$$

Remark: The original problem of size n is divided into 4 problems of input size $\frac{n}{2}$.

Divide and Conquer – First Try

The Divide Step: Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1},$$

$$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}.$$

Then $A(x) = A_0(x) + A_1(x) x^{\lfloor \frac{n}{2} \rfloor}$.

Similarly we define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\lfloor \frac{n}{2} \rfloor}.$$

Then

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}.$$

Remark: The original problem of size n is divided into 4 problems of input size $\frac{n}{2}$.

$T(n) \le 4T\left(\frac{n}{2}\right) + cn$ for some c & $n > n_0$

A_0B_0, A_0B_1, A_1B_0, A_1B_1 $\Big\}$ $4T\left(\frac{n}{2}\right)$

A_0B_0 $\Big\{$ $A_0B_1 x^{n/2} \to k_1 n$, $A_1B_0 x^{n/2} \to k_2 n$, $A_1B_1 x^{2n/2} \to k_3 n$ $\Big\}$ $4T\left(\frac{n}{2}\right) + cn$

$T(n) = 4T\left(\frac{n}{2}\right) + c \cdot O(n)$

Example:

$$\begin{aligned} A(x) &= 2+5x+3x^2+x^3-x^4 \\ B(x) &= 1+2x+2x^2+3x^3+6x^4 \\ A(x)B(x) &= 2+9x+17x^2+23x^3+34x^4+39x^5 \\ &\quad +19x^6+3x^7-6x^8 \end{aligned}$$

$$\begin{aligned} A_0(x) &= 2+5x, & A_1(x) &= 3+x-x^2, \\ & & A(x) &= A_0(x)+A_1(x)x^2 \\ B_0(x) &= 1+2x, & B_1(x) &= 2+3x+6x^2, \\ & & B(x) &= B_0(x)+B_1(x)x^2 \end{aligned}$$

$$\begin{aligned} A_0(x)B_0(x) &= 2+9x+10x^2 \\ A_1(x)B_1(x) &= 6+11x+19x^2+3x^3-6x^4 \\ A_0(x)B_1(x) &= 4+16x+27x^2+30x^3 \\ A_1(x)B_0(x) &= 3+7x+x^2-2x^3 \\ A_0(x)B_1(x)+A_1(x)B_0(x) &= 7+23x+28x^2+28x^3 \end{aligned}$$

$$\begin{aligned} &A_0(x)B_0(x)+(A_0(x)B_1(x)+A_1(x)B_0(x))x^2+A_1(x)B_1(x)x^4 \\ &= 2+9x+17x^2+23x^3+34x^4+39x^5+19x^6+3x^7-6x^8 \end{aligned}$$

The Conquer Step: Solve the four subproblems, i.e., computing

$$A_0(x)B_0(x), \quad A_0(x)B_1(x),$$
$$A_1(x)B_0(x), \quad A_1(x)B_1(x)$$

by recursively calling the algorithm 4 times.

The Conquer Step: Solve the four subproblems, i.e., computing

$$A_0(x)B_0(x), \quad A_0(x)B_1(x),$$
$$A_1(x)B_0(x), \quad A_1(x)B_1(x)$$

by recursively calling the algorithm 4 times.

The Combining Step: Adding the following four polynomials

$$\begin{aligned} & A_0(x)B_0(x) \\ & + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} \\ & + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} \\ & + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}. \end{aligned}$$

takes $\Theta(n)$ operations. Why?

PolyMulti1$(A(x), B(x))$

{

$A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}$;

$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}$;

$B_0(x) = b_0 + b_1 x + \cdots + b_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}$;

$B_1(x) = b_{\lfloor \frac{n}{2} \rfloor} + b_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + b_n x^{n - \lfloor \frac{n}{2} \rfloor}$;

$U(x) = PolyMulti1(A_0(x), B_0(x))$;
$V(x) = PolyMulti1(A_0(x), B_1(x))$;
$W(x) = PolyMulti1(A_1(x), B_0(x))$;
$Z(x) = PolyMulti1(A_1(x), B_1(x))$;

return $\left(U(x) + [V(x) + W(x)] x^{\lfloor \frac{n}{2} \rfloor} + Z(x) x^{2 \lfloor \frac{n}{2} \rfloor} \right)$

}

Running Time of the Algorithm

Assume n is a power of 2, $n = 2^h$. By substitution (expansion),

$$
\begin{aligned}
T(n) &= 4\,T\left(\frac{n}{2}\right) + cn \\
&= 4\left[4\,T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right] + cn \\
&= 4^2\,T\left(\frac{n}{2^2}\right) + (1+2)cn \\
&= 4^2\left[4\,T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right] + (1+2)cn \\
&= 4^3 T\left(\frac{n}{2^3}\right) + (1+2+2^2)cn \\
&\;\;\vdots \\
&= 4^i T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{i-1} 2^j cn \quad \text{(induction)} \\
&\;\;\vdots \\
&= 4^h T\left(\frac{n}{2^h}\right) + \sum_{j=0}^{h-1} 2^j cn \\
&= n^2\,T(1) + cn(n-1) \\
&\quad \text{(since } n = 2^h \text{ and } \sum_{j=0}^{h-1} 2^j = 2^h - 1 = n-1\text{)} \\
&= \Theta(n^2).
\end{aligned}
$$

The same order as the brute force approach!

This proof is by direct computation for the special case when $n = 2^h$

$T(n) = 4T\left(\frac{n}{2}\right) + c(n)$

n need not be 2^h

Assume $T(n) = O(n^2)$

Substitute & by induct $T(n/2)$ is $O(n^2)$ $T(n/2)$ bec

$O(n^2)$

Lesson

- Divide and Conquer may not be the silver bullet
- You have to think deeper.

What can we do? How can we reduce steps?

Then

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}.$$

Remark: The original problem of size n is divided into 4 problems of input size $\frac{n}{2}$.

Note that these middle two terms can be combined because they have the same coefficients!!!
So we need to only compute

What can we do? How can we reduce steps?

Then

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_0(x)x^{\lfloor \frac{n}{2} \rfloor} + A_1(x)B_1(x)x^{2\lfloor \frac{n}{2} \rfloor}.$$

$A_0 B_0$ } $2T(\frac{n}{2})$
$A_1 B_1$ }

Remark: The original problem of size n is divided into 4 problems of input size $\frac{n}{2}$.

$A_0B_1 + B_1A_0 = CD - A_0B_0$
$T(\frac{n}{2}) + O(n) - A_1B_1$

Note that these middle two terms can be combined because they have the same coefficients!!!
So we need to only compute

$$A_0(x)B_1(x) + A_1(x)B_0(x)$$

$A \neq A_0 + A_1 = C$ $\quad B \neq B_0 + B_1$ $\longrightarrow$ $O(\frac{n}{2})$ $O(n)$

$CD = (A_0 + A_1) * (B_0 + B_1)$
$T(\frac{n}{2}) = (A_0B_0) + (A_1B_1) + (A_0B_1 + A_1B_0)$

Define

$$\begin{aligned} Y(x) &= (A_0(x) + A_1(x)) \times (B_0(x) + B_1(x)) \\ U(x) &= A_0(x)B_0(x) \\ Z(x) &= A_1(x)B_1(x) \end{aligned}$$

Then

$$Y(x) - U(x) - Z(x) = A_0(x)B_1(x) + A_1(x)B_0(x).$$

Hence $A(x)B(x)$ is equal to

$$U(x) + [Y(x) - U(x) - Z(x)]x^{\lfloor \frac{n}{2} \rfloor} + Z(x) \times x^{2\lfloor \frac{n}{2} \rfloor}$$

Conclusion: You need to call the multiplication procedure 3, rather than 4 times.

PolyMulti2$(A(x), B(x))$
{

$A_0(x) = a_0 + a_1 x + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}$;

$A_1(x) = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + a_n x^{n - \lfloor \frac{n}{2} \rfloor}$;

$B_0(x) = b_0 + b_1 x + \cdots + b_{\lfloor \frac{n}{2} \rfloor - 1} x^{\lfloor \frac{n}{2} \rfloor - 1}$;

$B_1(x) = b_{\lfloor \frac{n}{2} \rfloor} + b_{\lfloor \frac{n}{2} \rfloor + 1} x + \cdots + b_n x^{n - \lfloor \frac{n}{2} \rfloor}$;

$Y(x) = PolyMulti2(A_0(x) + A_1(x), B_0(x) + B_1(x))$

$U(x) = PolyMulti2(A_0(x), B_0(x))$;

$Z(x) = PolyMulti2(A_1(x), B_1(x))$;

return $\left(U(x) + [Y(x) - U(x) - Z(x)] x^{\lfloor \frac{n}{2} \rfloor} + Z(x) x^{2 \lfloor \frac{n}{2} \rfloor} \right)$;

}

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

Assume $n = 2^h$. Let $\lg x$ denote $\log_2 x$.
By the substitution method,

$$
\begin{aligned}
T(n) &= 3\,T\left(\frac{n}{2}\right) + c\,n \\
&= 3\left[3\,T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right] + c\,n \\
&= 3^2\,T\left(\frac{n}{2^2}\right) + \left(1 + \frac{3}{2}\right) c\,n \\
&= 3^2\left[3\,T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right] + \left(1 + \frac{3}{2}\right) c\,n \\
&= 3^3 T\left(\frac{n}{2^3}\right) + \left(1 + \frac{3}{2} + \left[\frac{3}{2}\right]^2\right) c\,n \\
&\;\;\vdots \\
&= 3^h T\left(\frac{n}{2^h}\right) + \sum_{j=0}^{h-1} \left[\frac{3}{2}\right]^j c\,n.
\end{aligned}
$$

Assume $n = 2^h$. Let $\lg x$ denote $\log_2 x$.
By the substitution method,

$$
\begin{aligned}
T(n) &= 3\,T\left(\frac{n}{2}\right) + c\,n \\
&= 3\left[3\,T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right] + c\,n \\
&= 3^2\,T\left(\frac{n}{2^2}\right) + \left(1 + \frac{3}{2}\right) c\,n \\
&= 3^2\left[3\,T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right] + \left(1 + \frac{3}{2}\right) c\,n \\
&= 3^3 T\left(\frac{n}{2^3}\right) + \left(1 + \frac{3}{2} + \left[\frac{3}{2}\right]^2\right) c\,n \\
&\;\;\vdots \\
&= 3^h T\left(\frac{n}{2^h}\right) + \sum_{j=0}^{h-1} \left[\frac{3}{2}\right]^j c\,n
\end{aligned}
$$

$$T(n) = \Theta(n^{\lg 3} T(1) + 2\,c\,n^{\lg 3}) = \Theta(n^{\lg 3}).$$

13 Dec. 2021

- Quick Sort.

Quicksort is a sorting algorithm based on the divide and conquer approach where

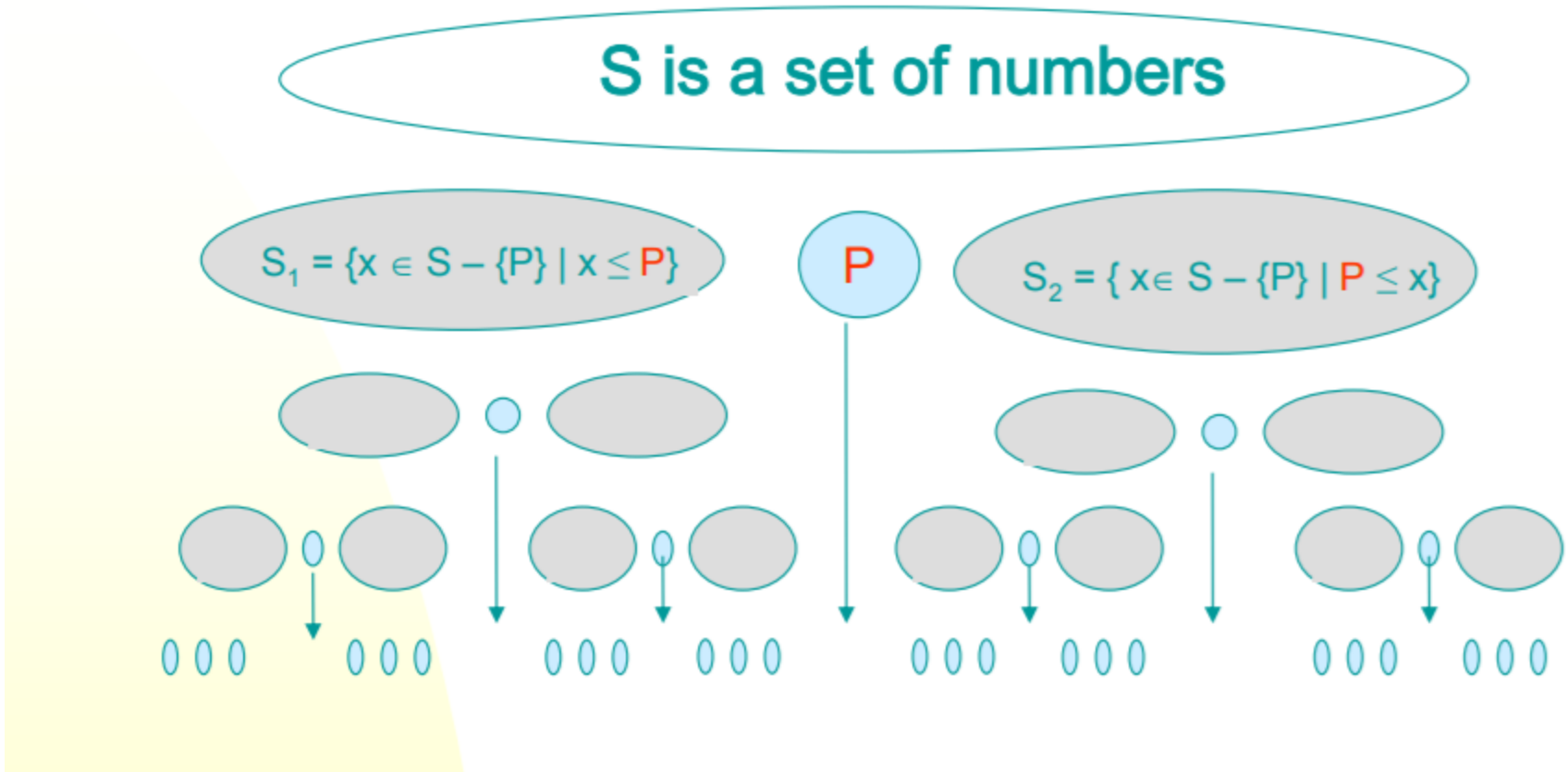
1. An array is divided into subarrays by selecting a pivot element (element selected from the array).
2. While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
3. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
4. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

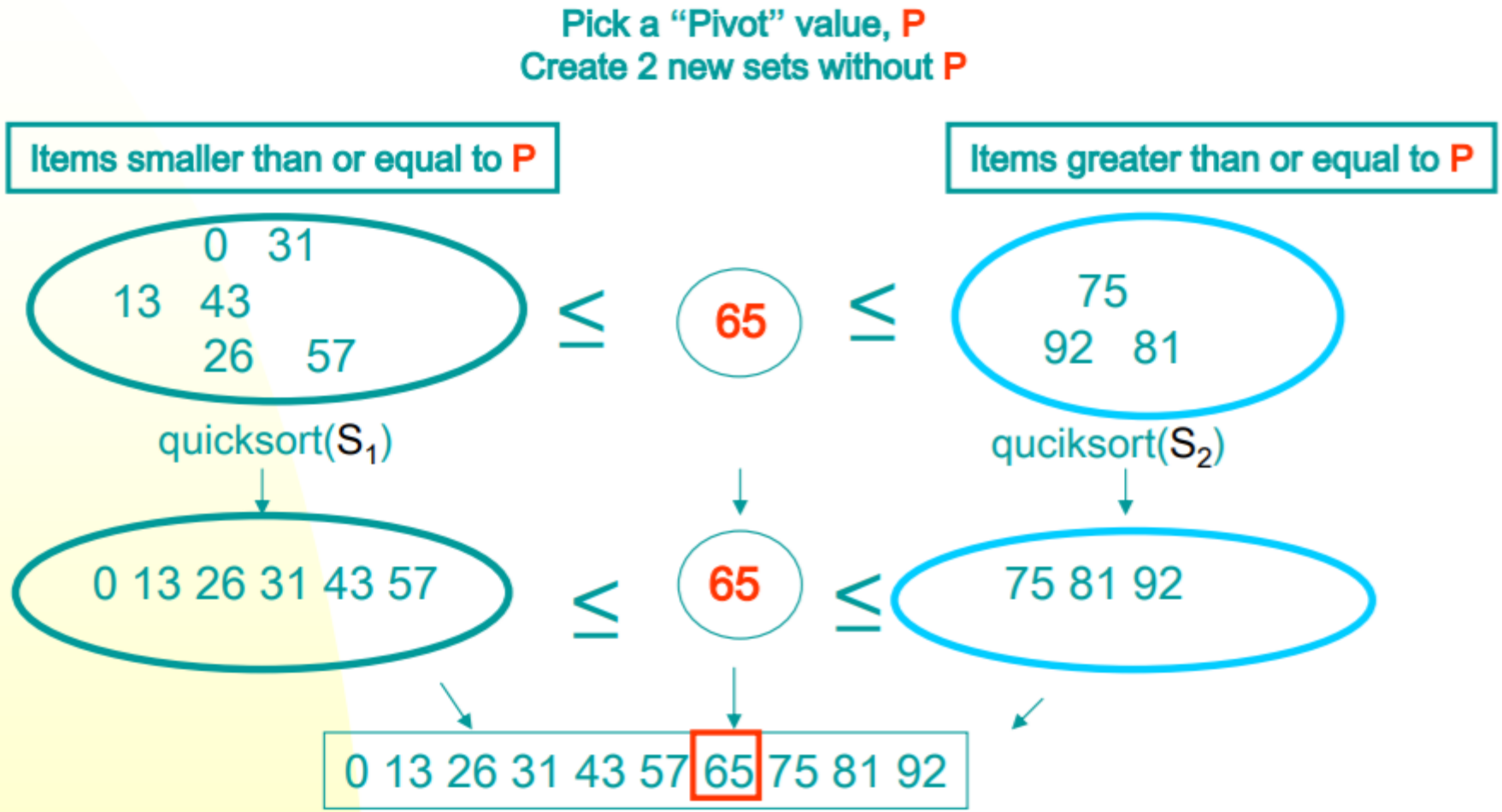
Basic Ideas

(Another divide-and-conquer algorithm)

- Pick an element, say ***P*** (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to (≤) ***P*** (the **left-**block S_1)
  2. ***P*** (the only element in the **middle**-block)
  3. those greater than or equal to (≥) ***P*** (the **right-**block S_2)
- Repeat the process **recursively** for the **left-** and **right-** sub-blocks. Return {quicksort(S_1), ***P***, quicksort(S_2)}. (That is the results of quicksort(S_1), followed by ***P***, followed by the results of quicksort(S_2))

Basic Ideas




Pick a "Pivot" value, P
Create 2 new sets without P
Items smaller than or equal to P
Items greater than or equal to P
0 31
13 43
26 57
≤
65
≤
75
92 81
quicksort(S_1)
quciksort(S_2)
0 13 26 31 43 57
≤
65
≤
75 81 92
0 13 26 31 43 57 65 75 81 92

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot
3. Pick a random element as pivot.
4. Pick median as pivot.

Note:

- The main idea is to find the “right” position for the pivot element ***P***.
- After each **“pass”**, the pivot element, ***P***, should be **“in place”.**
- Eventually, the elements are sorted since each pass puts at least one element (i.e., ***P***) into its final position.

Issues:

- How to choose the pivot ***P*** ?
- How to partition the block into sub-blocks?

Implementation

Algorithm I:

```
int partition(int A[ ], int left, int right);

// sort A[left..right]
void quicksort(int A[ ], int left, int right)
{   int q ;
    if (right > left )
    {
     q = partition(A, left, right);
     // after 'partition'
     //→ A[left..q-1] ≤ A[q] ≤ A[q+1..right]
     quicksort(A, left, q-1);
     quicksort(A, q+1, right);
    }
}
```

Implementation

```
// select A[left] be the pivot element)
int partition(int A[], int left, int right);
{   P = A[left];
    i = left;
    j = right + 1;
    for(;;) //infinite for-loop, break to exit
    { while (A[++i] < P) if (i >= right) break;
      // Now, A[i] ≥ P
      while (A[--j] > P) if (j <= left) break;
      // Now, A[j] ≤ P
      if (i >= j ) break; // break the for-loop
      else swap(A[i], A[j]);
    }
    if (j == left) return j ;
    swap(A[left], A[j]);
    return j;
}
```

Example

Input:	65	70	75	80	85	60	55	50	45	
P: 65	*i*									
Pass 1	**65**	70	75	80	85	60	55	50	45	
(i)		*i*							*j*	← **swap (A[i], A[j])**
	65	**45**	75	80	85	60	55	50	**70**	
(ii)			*i*					*j*		← **swap (A[i], A[j])**
	65	**45**	**50**	80	85	60	55	**75**	**70**	
(iii)				*i*			*j*			← **swap (A[i], A[j])**
	65	**45**	**50**	**55**	85	60	**80**	**75**	**70**	
(iv)					*i*	*j*				← **swap (A[i], A[j])**
	65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**	
(v)					*j*	*i*				**if `(i>=j) break`**
	60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**	**swap (A[left], A[j])**

Items smaller than or equal to 65

Items greater than or equal to 65

Example

Result of Pass 1: 3 sub-blocks:

60 45 50 55 **65** 85 80 75 70

Pass 2a (left sub-block):

60 45 50 55 (P = 60)

i *j*

60 45 50 55

j i if `(i>=j) break`

55 45 50 60 swap (A[left], A[j])

Pass 2b (right sub-block):

85 80 75 70 (P = 85)

i *j*

85 80 75 70

j i if `(i>=j) break`

70 80 75 85 swap (A[left], A[j])

Quick sort

The Worst-case (?)

The Best-case (?)

The Average-case (?)

Running time analysis

Best case:

- The pivot is in the middle (median) (at each partition step), i.e. after each partitioning, on a block of size ***n,*** the result
  - yields two sub-blocks of approximately equal size and the pivot element in the "middle" position
  - takes ***n*** data comparisons.
- Recurrence Equation becomes

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n/2) + cn \end{cases}$$

Solution: $\theta(n \log n)$

Comparable to Mergesort!!

Running time analysis

Worst-Case (Data is sorted already)

- When the pivot is the smallest (or largest) element at partitioning on a block of size ***n***, the result
  - yields one empty sub-block, one element (pivot) in the "correct" place and one sub-block of size ***(n-1)***
  - takes $\theta(n)$ times.
- Recurrence Equation:

$$\begin{cases} T(1) = 1 \\ T(n) = T(n-1) + cn \end{cases}$$

Solution: $\theta(n^2)$

Worse than Mergesort!!!

Running time analysis

Average case:

It turns out the average case running time also is $\theta(n \log n)$.

20Dec2021

Miscellaneous and Closing

```c
int main()
/* Explain the code – what is it doing?*/
{
    FILE *fp;
    int no_lines = 0;
    char filename[40]; //what is this line doing?
   char sample_chr;

    printf("Enter file name: ");
    scanf("%s", filename);
    fp = fopen(filename, "r"); //what is this line doing?

    sample_chr = getc(fp);
```

```c
int main()
/* Explain the code – what is it doing?*/
{
    FILE *fp;
    int no = 0;
    char filename[40], //what is this line doing?
   char sample_chr;

    printf("Enter file name: ");
    scanf("%s", filename);
    fp = fopen(filename, "r"); //what is this line doing?

    sample_chr = getc(fp);//what dies this line do?
```

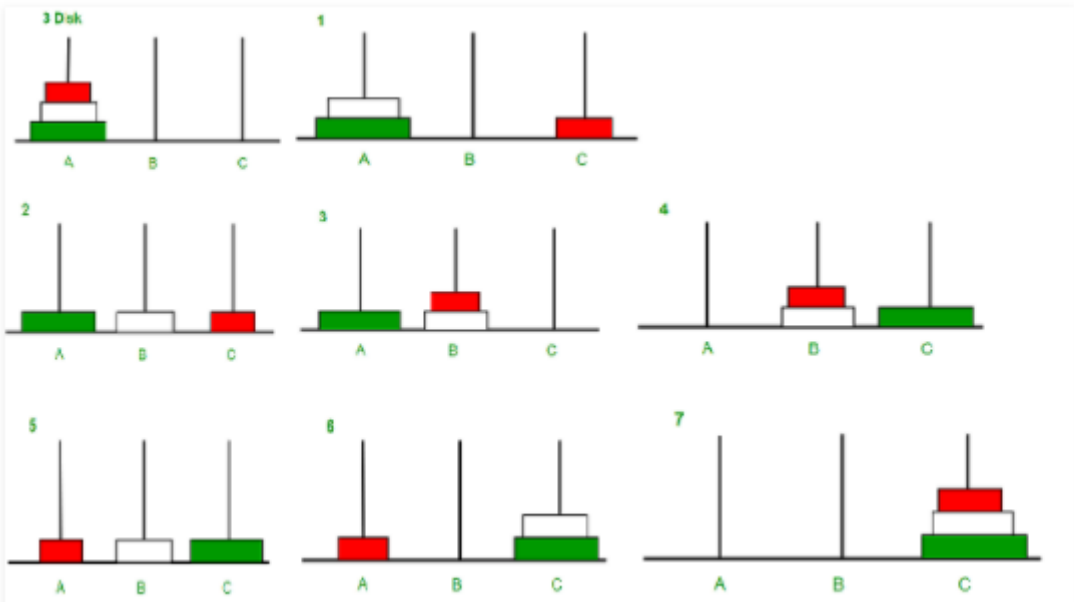
```c
while (sample_chr != EOF) {
        if (sample_chr == '/n')
        {
            //increment variable 'no' by 1
            no=no+1;
        }
        //take next character from file.
        sample_chr = getc(fp);
    }

// WHAT IS THE WHILE LOOP DOING?
    fclose(fp); //close file.
    printf(" %d %s ", no, filename);
    return 0;
}
```

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

```
Step 1 : Shift first disk from 'A' to 'B'.
Step 2 : Shift second disk from 'A' to 'C'.
Step 3 : Shift first disk from 'B' to 'C'.

The pattern here is :
Shift 'n-1' disks from 'A' to 'B'.
Shift last disk from 'A' to 'C'.
Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks :
```



```
#include <stdio.h>
void TOH(int n,char x,char y,char z)
{   if(n>0)

{     TOH(n-1,x,z,y);
            //printf("\n%c to %c",x,y);

            TOH(1,x,y,z);

            TOH(n-1,z,y,x);
}
}

int main()
{
int n=5;

TOH(n,'A','B','C');
return 0;}
```

```
#include <stdio.h>
void TOH(int n,char x,char y,char z)
{   if(n>0)

{     TOH(n-1,x,z,y);
            //printf("\n%c to %c",x,y);

            TOH(n-1,z,y,x);
}
}

int main()
{
int n=5;

TOH(n,'A','B','C');
return 0;}
```

How many time printf executed?

How will you modify the program to count it?

```
#include <stdio.h>
void TOH(int n,char x,char y,char z)
{   if(n>0)

{     TOH(n-1,x,z,y);
            printf("\n%c to %c",x,y);

            TOH(n-1,z,y,x);
}
}

int main()
{
int n=5;

TOH(n,'A','B','C');
return 0;}
```

$T(n) = T(n-1) + c + T(n-1)$

$T(n) - 2\ T(n-1) + c$

What the time complexity of this algorithm?

The End