Performance Analysis

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9337938766, 2462358

What is an Algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- As a computer professional, it is useful to know a standard set of important algorithms and to be able to design new algorithms as well as to analyze their efficiency.

What is an algorithm?

- Recipe, process, method, technique, procedure, routine,… with following requirements:
  1. Finiteness

     terminates after a finite number of steps
  2. Definiteness

     rigorously and unambiguously specified
  3. Input

     valid inputs are clearly specified
  4. Output

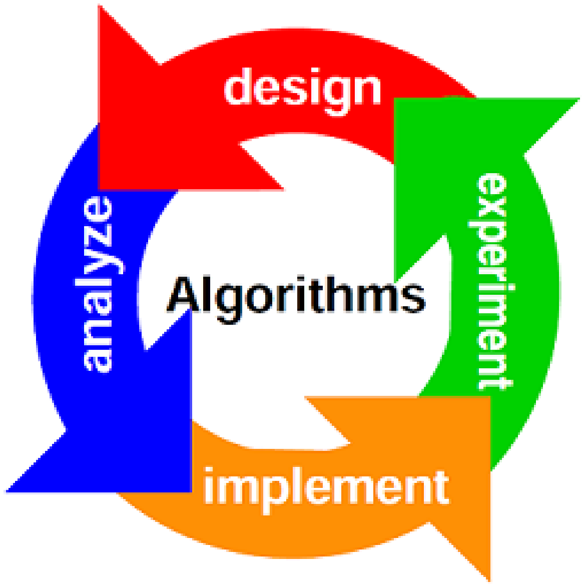
     can be proved to produce the correct output given a valid input
  5. Effectiveness

     steps are sufficiently simple and basic

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost



Problem & Algorithm

- An ***instance of a problem*** consists of all inputs needed to compute a solution to the problem.
- An algorithm is said to be ***correct*** if for every input instance, it halts with the correct output.
- A **correct algorithm** ***solves*** the given computational problem.
- An **incorrect algorithm** might not halt at all on some input instance, or it might halt with other than the desired answer.

Example: sorting

Statement of problem:

- Input:

  A sequence of n numbers

  $<a_1, \; a_2, \ldots, a_n>$

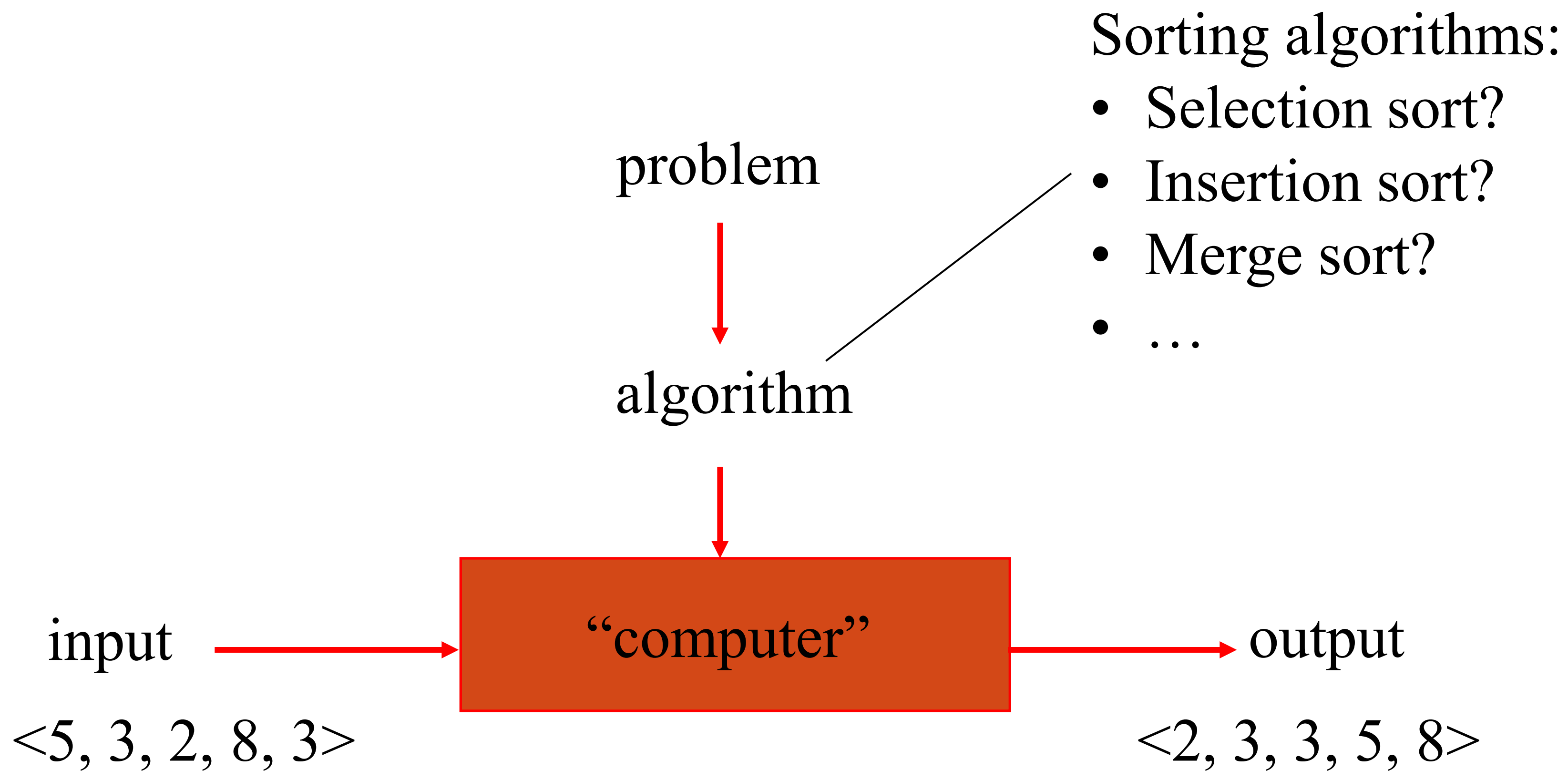
- Output:

  A reordering of the input sequence

  $<b_1, \; b_2, \ldots, b_n>$

  so that $b_i \leq b_j$ whenever $i < j$

Example: sorting



Sorting algorithms:

- Selection sort?
- Insertion sort?
- Merge sort?
- …

Selection sort

- Input:

  array a[0], …, a[n-1]

- Output:

  array a sorted in non-decreasing order

- Algorithm:

```
for i=0 to n-1
swap a[i] with smallest of a[i],…a[n-1]
```

Some well-known computational problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Basic issues related to algorithms

- How to design algorithms ?
- How to express algorithms ?
- Proving correctness of algorithms
- Efficiency
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound
- Space and time tradeoffs

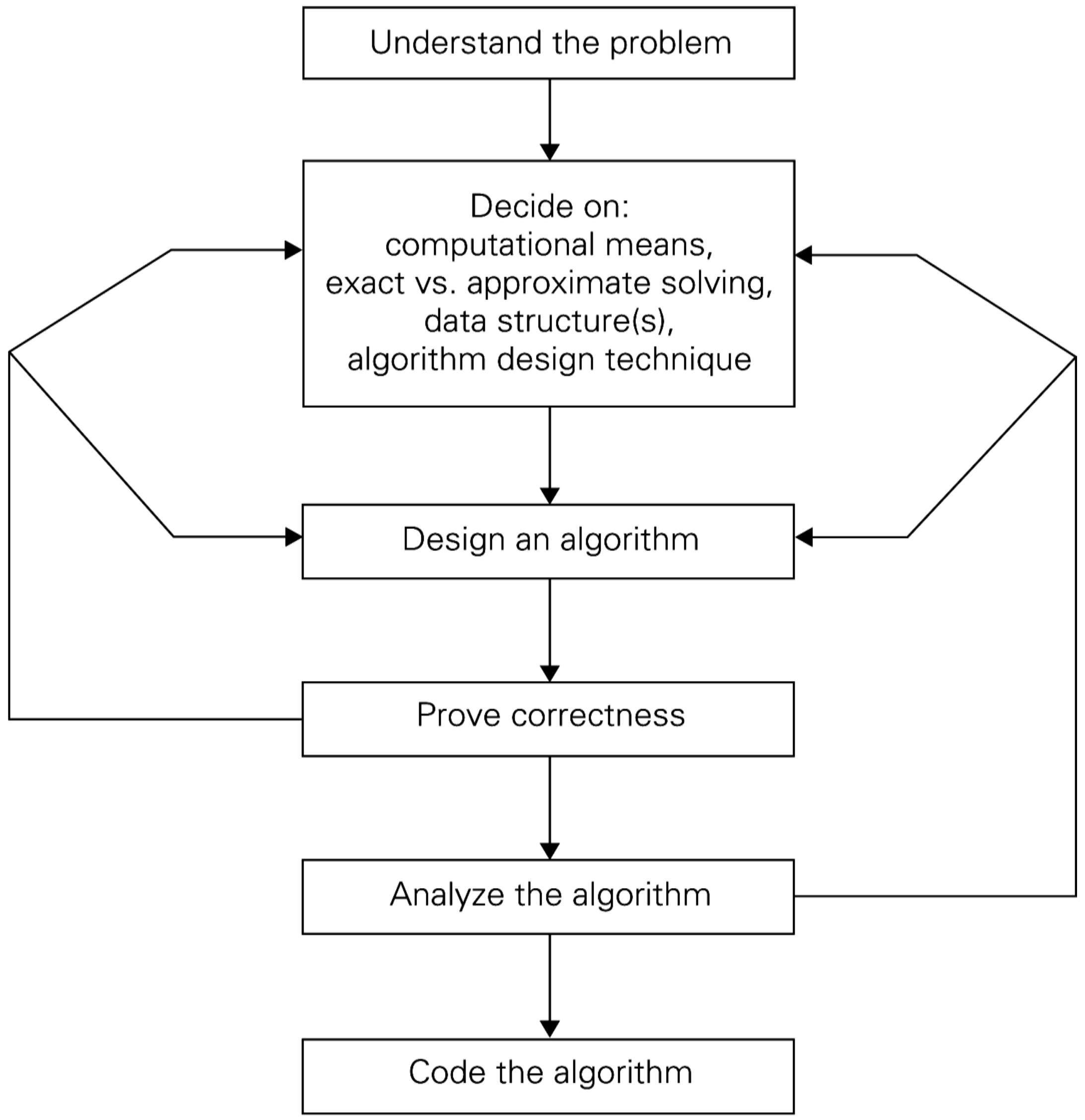
Analysis of algorithms

- **How good is the algorithm?**
  - Correctness
  - **Time efficiency**
  - **Space efficiency**
  - Simplicity
  - Generality
- **Does there exist a better algorithm?**
  - Lower bounds
  - Optimality

Why study algorithms and performance?

1. Algorithms help us to understand ***scalability.***
2. Performance often draws the line between what is feasible and what is impossible.
3. Algorithmic mathematics provides a ***language*** *for talking about program behavior.*
4. Performance is the ***currency*** *of computing.*
5. The lessons of program performance generalize to other computing resources.
6. Speed is fun!

Algorithm Design and Analysis Process



Performance Analysis

- The **efficiency** of an algorithm can be decided by measuring the **performance** of an algorithm.
- Performance of an algorithm is a process of making evaluative judgement about algorithms**.**
- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- The performance of an algorithm depends upon two factors.

❑ – 1) Amount of time required by an algorithm to execute (known as **time complexity**).

❑ – 2) Amount of storage required by an algorithm (known as **Space complexity**).

- **Performance analysis of an algorithm** is the process of calculating space and time required by that algorithm.

Performance of an algorithm

- Generally, the performance of an algorithm depends on the following elements…

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? etc.,

The goodness of an algorithm

- Time complexity (more important)
- Space complexity
- For a parallel algorithm :
  - time-processor product
- For a VLSI circuit :
  - area-time (AT, AT^2)

Need for analysis

1. To determine resource consumption
   - CPU time
   - Memory space
2. Compare different methods for solving the same problem before actually implementing them and running the programs.
3. To find an efficient algorithm

Complexity

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal* factors
  - *external* factors

Internal and external factors

internal factors

- The algorithm's efficiency, in terms of:
  - Time required to run
  - Space (memory storage)required to run

external factors

- Speed of the computer on which it is run
- Quality of the compiler
- Size of the input to the algorithm

Complexity measures the *internal* factors (usually more interested in time than space)

Two ways of finding complexity

- **Performance measurement** (Experimental study)

  Through Experiments or Empirical Approach

- **Performance Analysis** **(Theoretical Analysis)**

  Analytical Method or Theoretical Approach

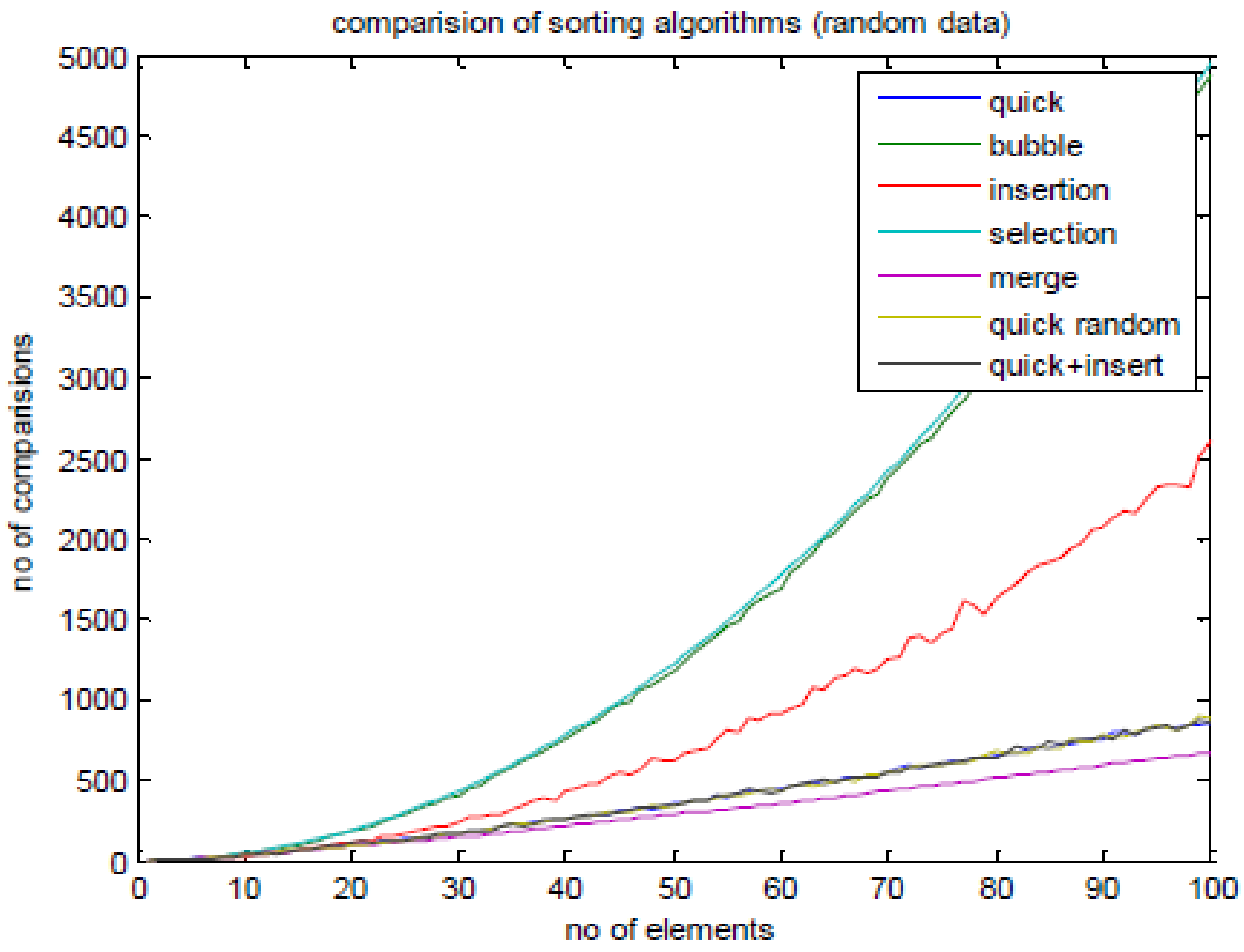
Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
- Use a method like System.currentTimeMillis()
- Plot the results

Java Code – Simple Program

```
import java.io.*;
class for1
{
 public static void main(String args[])
   throws Exception
  {
  int N,Sum;
  N=10000; // N value to be changed.
  Sum=0;
  long
   start=System.currentTimeMillis();
  int i,j;
for(i=0;i<N;i++)
  for(j=0;j<i;j++)
  Sum++;
  long
   end=System.currentTimeMillis()
   ;
  long time=end-start;
  System.out.println(" The start time
   is :  "+start);
  System.out.println(" The end time
   is :  "+end);
  System.out.println(" The time
   taken is :  "+time);
 }
}
```

Example plot



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space Complexity

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like function calls, jumping statements etc.,.

- Total amount of computer memory(primary memory) required by an algorithm to complete its execution is called as space complexity of that algorithm.

Space Complexity

```
Algorithm abc($a, b, c$)
{
    return $a + b + b * c + (a + b - c)/(a + b) + 4.0$;
}
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    for $i := 1$ to $n$ do
        $s := s + a[i]$;
    return $s$;
}
```

Algorithm 1.6 Iterative function for sum

Representation in computer

- **Computers** use a fixed number of bits to represent an **integer**. The commonly-used bit-lengths for **integers** are 8-bit, 16-bit, 32-bit or 64-bit ….
- Unsigned **Integers**: can represent zero and positive **integers**.
- Signed **Integers**: can represent zero, positive and negative **integers**.
- **Integers**. **Integers** are commonly **stored** using a word of **memory**, which is 4 bytes or 32 bits, so **integers** from 0 up to 4,294,967,295 (2^{32} - 1) can be **stored**.
- Below are the **integers** 1 to 5 **stored** as four-byte values (each row **represents** one **integer**).

```
0         :00000001  00000000  00000000  00000000  | 1
4         :00000010  00000000  00000000  00000000  | 2
8         :00000011 00000000   00000000  00000000  | 3
12        :00000100 00000000   00000000  00000000  | 4
16        :00000101 00000000   00000000  00000000  | 5
```

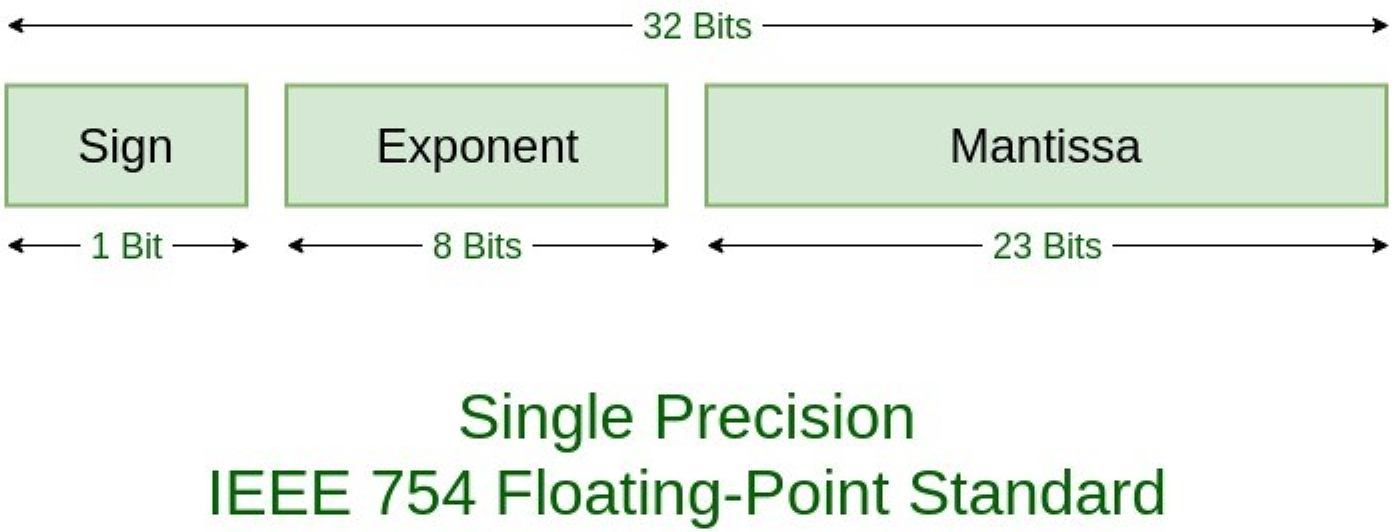
IEEE Standard 754 Floating Point Numbers

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**.
- The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

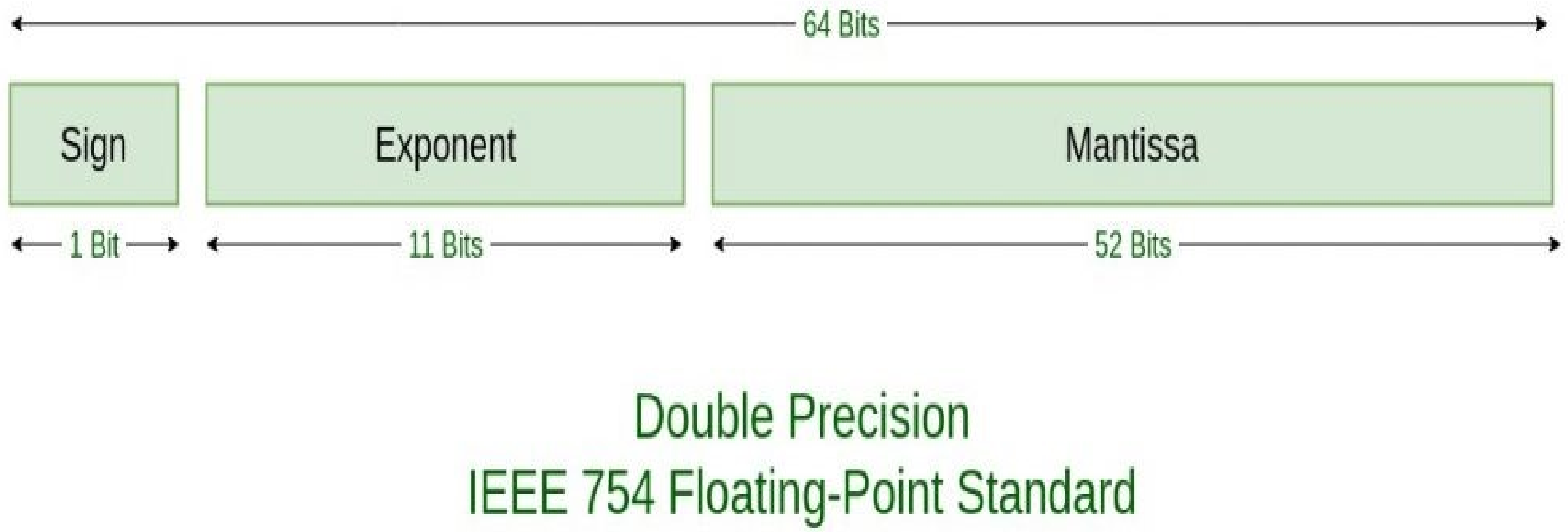
IEEE Standard 754 Floating Point Numbers

- IEEE 754 has 3 basic components:
- **The Sign of Mantissa –** This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
- **The Biased exponent –**The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
- **The Normalised Mantissa –** The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

IEEE Standard 754 Floating Point Numbers



IEEE Standard 754 Floating Point Numbers



Space Complexity

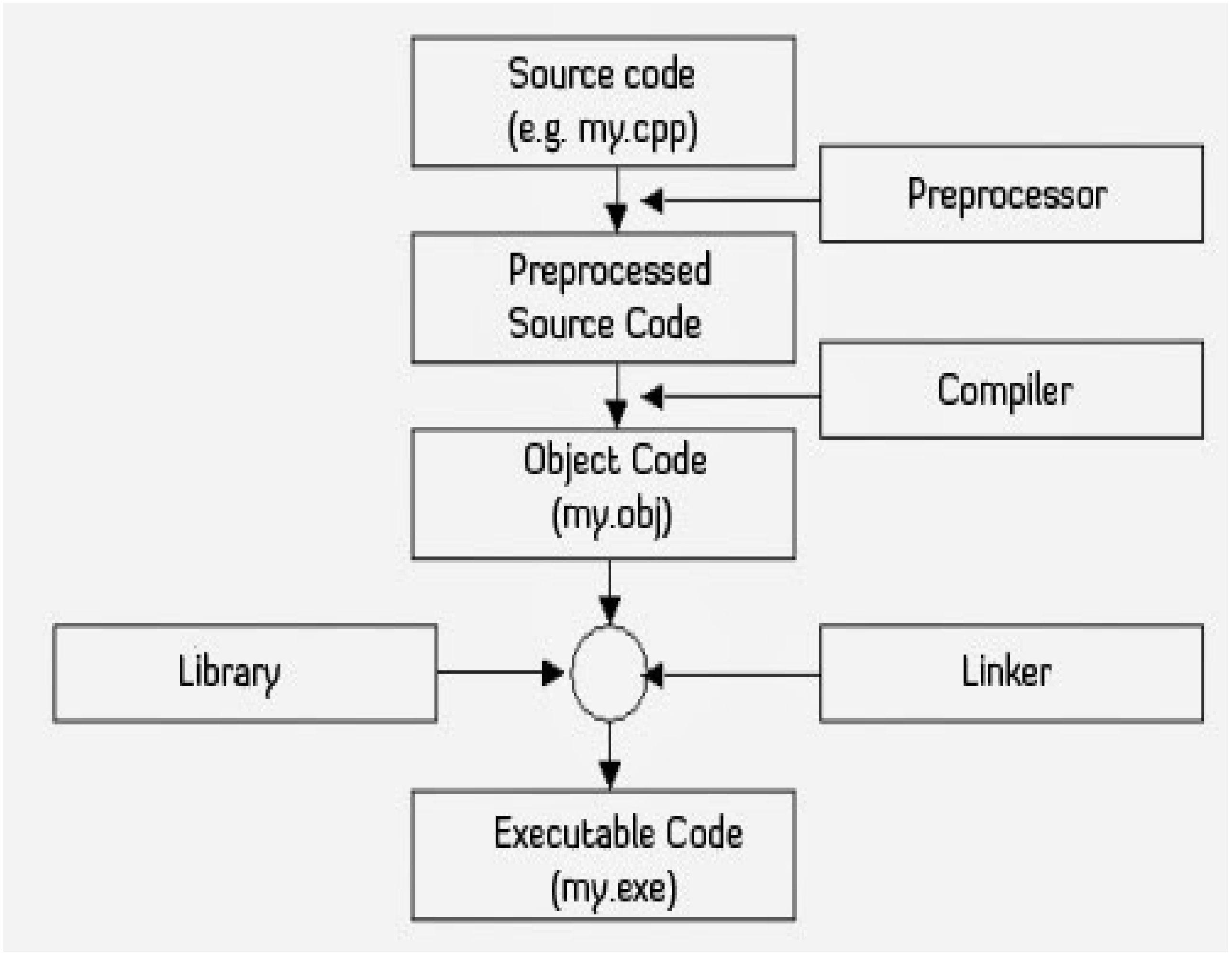
The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space requirements of an algorithm

- **The fixed part of an Algorithm (C)**
  - includes space for
    - Instructions
    - Simple variables
    - Fixed size component variables
    - Space for constants
    - Etc..
- **Variable Part of an Algorithm $S_P(I)$**

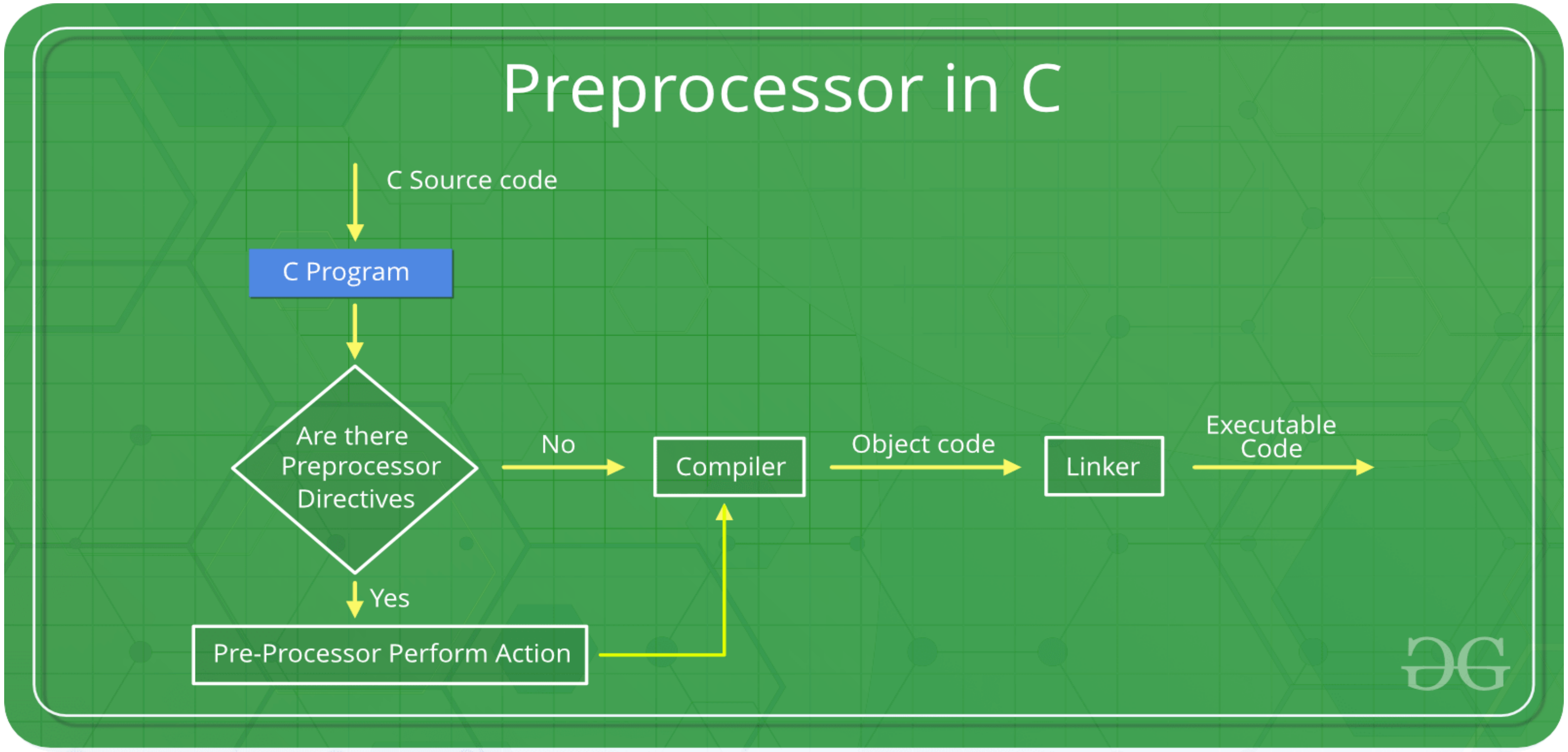
Steps of Conversion Source Code into Executable Code

- A compiler is a program which reads in a "program" in a source language and translates it into a program (often executable) in a target language.



STEP 1: PREPROCESSOR

- The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.



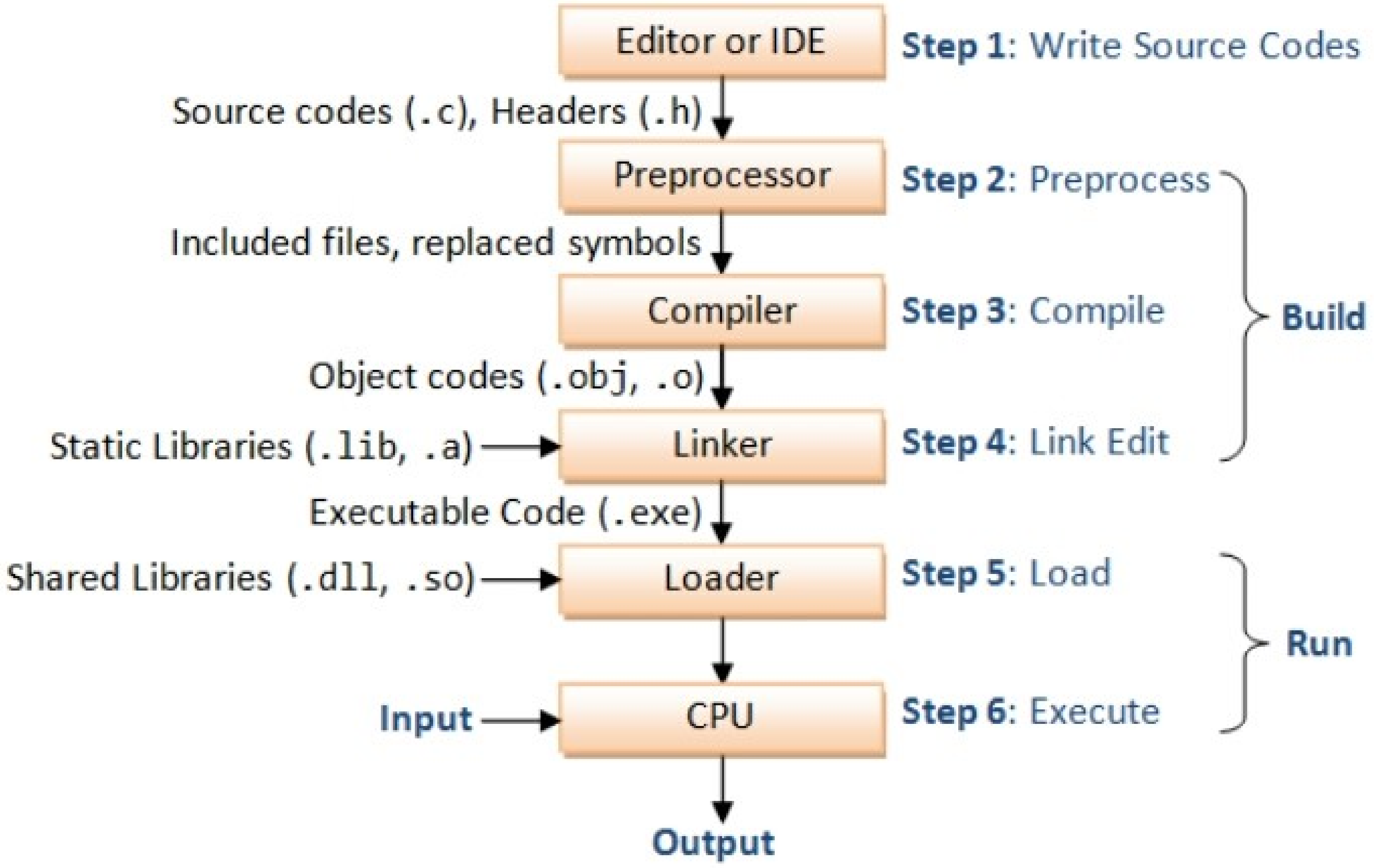
STEP 1: PREPROCESSOR

- The **C preprocessor** is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.
- A **preprocessor** is a program that processes its input data to produce output that is used as input to another program.
- The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers.
- The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of full-fledged programming languages.

The C preprocessor provides four separate facilities

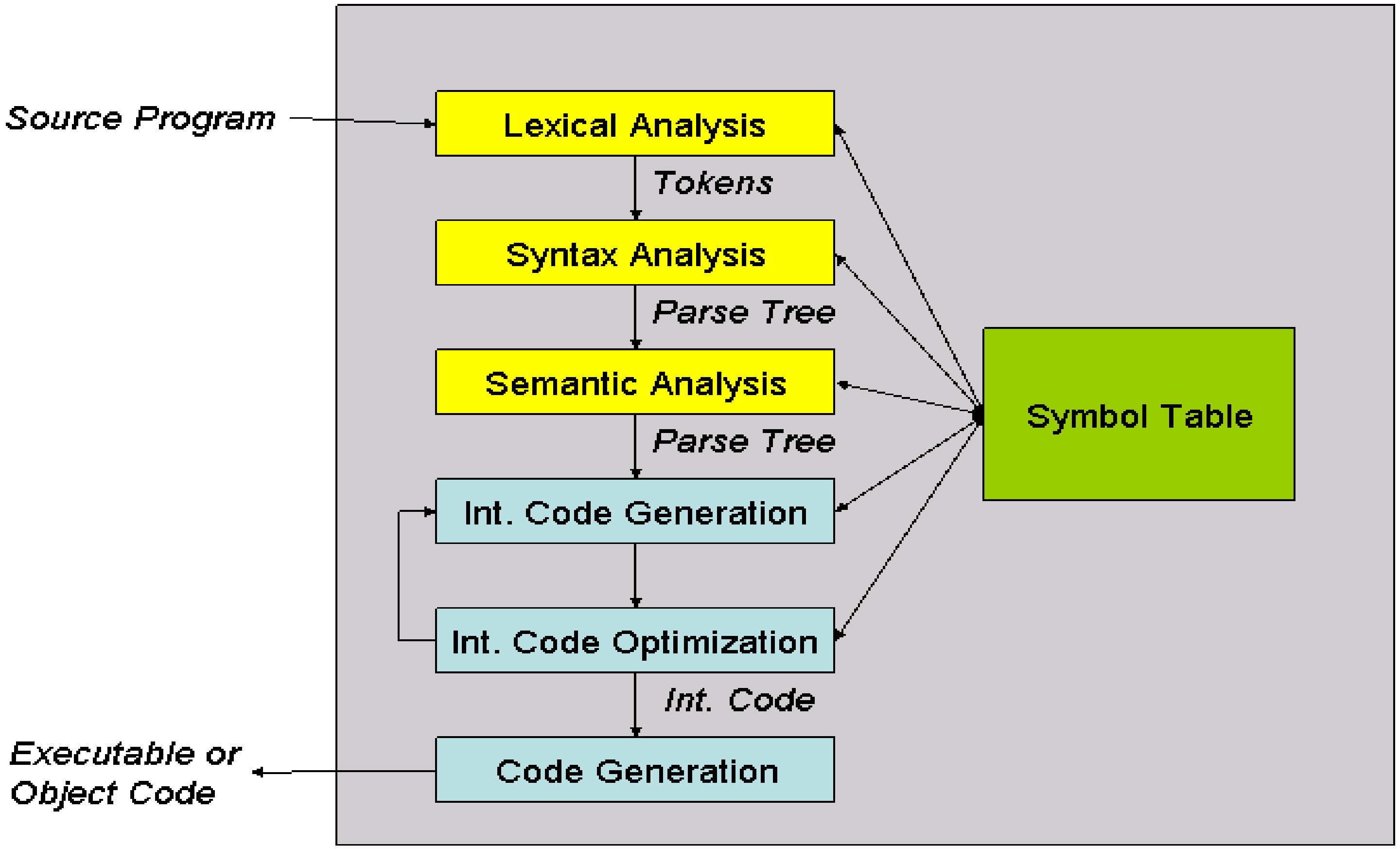
- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

how c preprocessor works



STEP 2: COMPILER [Compiling Stages]

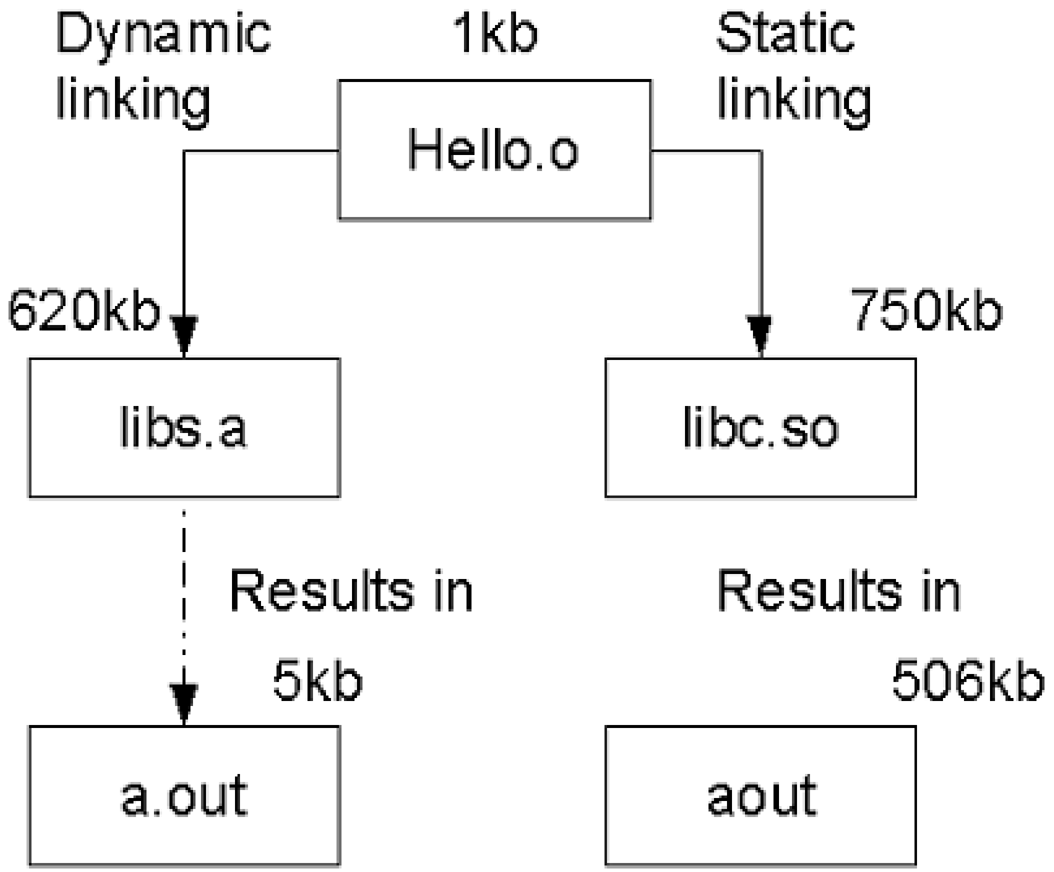
- The internal representation of the code generated by the compiler is called an intermediate language (IL).



STEP 3: LIBRARY

- A **library** is a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked.
- As long as a higher level program uses a library to make system calls, it does not need to be re-written to implement those system calls over and over again.
- In addition, the behavior is provided for reuse by multiple independent programs.
- A program invokes the library-provided behavior via a mechanism of the language.

STEP 3: LIBRARY (Linking Library)



Types of Library

- **1. Static Library :** Static libraries are collection of object files. As the linker processes a library it simply takes a copy of the object files it needs as if you had just compiled them. At the end of linking the executable (i.e. the program you will finally run) contains all your code and all the required library code so can be quite large.

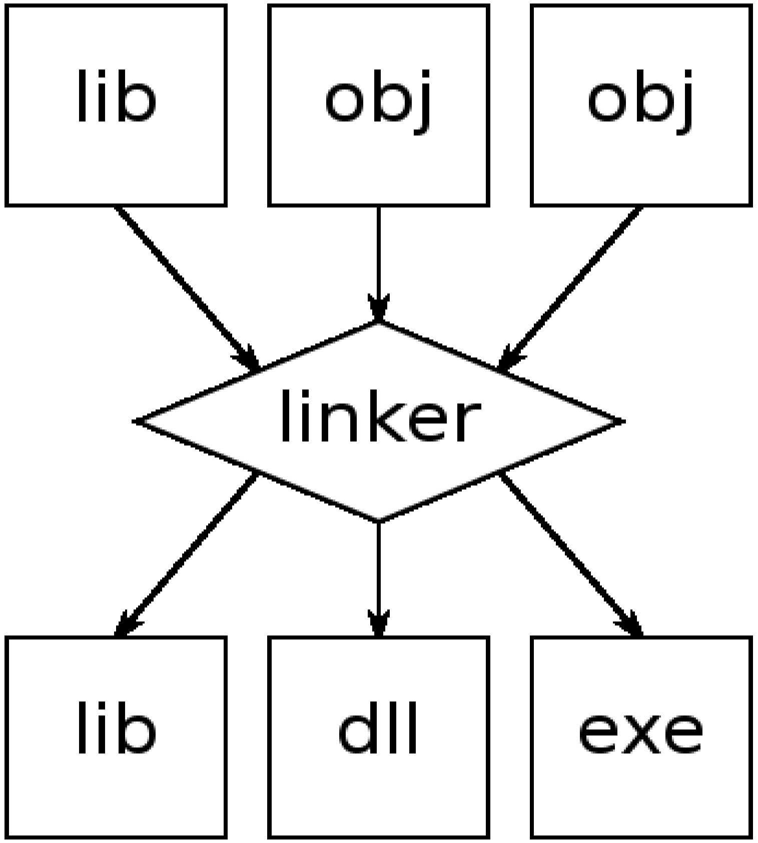
Extension Used : a.a suffix e.g. libZoo.a

- **2. Dynamic Library / Shareable :** The input to this type of a library is the same as for static libraries i.e. object files, it is the way the linker handles them that is different. In this case it records the information necessary to load it into memory but does not add the code to the executable. So at the end of linking the executable is typically very small. During execution, if the code is called then it is loaded dynamically into shared memory that can be used by other programs.

Extension Used : a .so suffix e.g. libZoo.so

STEP 4: LINKER

- A **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.



STEP 4: LINKER

- A **linker** or **link editor** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.
- A simpler version that wrote its output directly to memory was called the *loader*, though loading is typically considered a separate process
- Typically, an object file can contain three kinds of symbols:

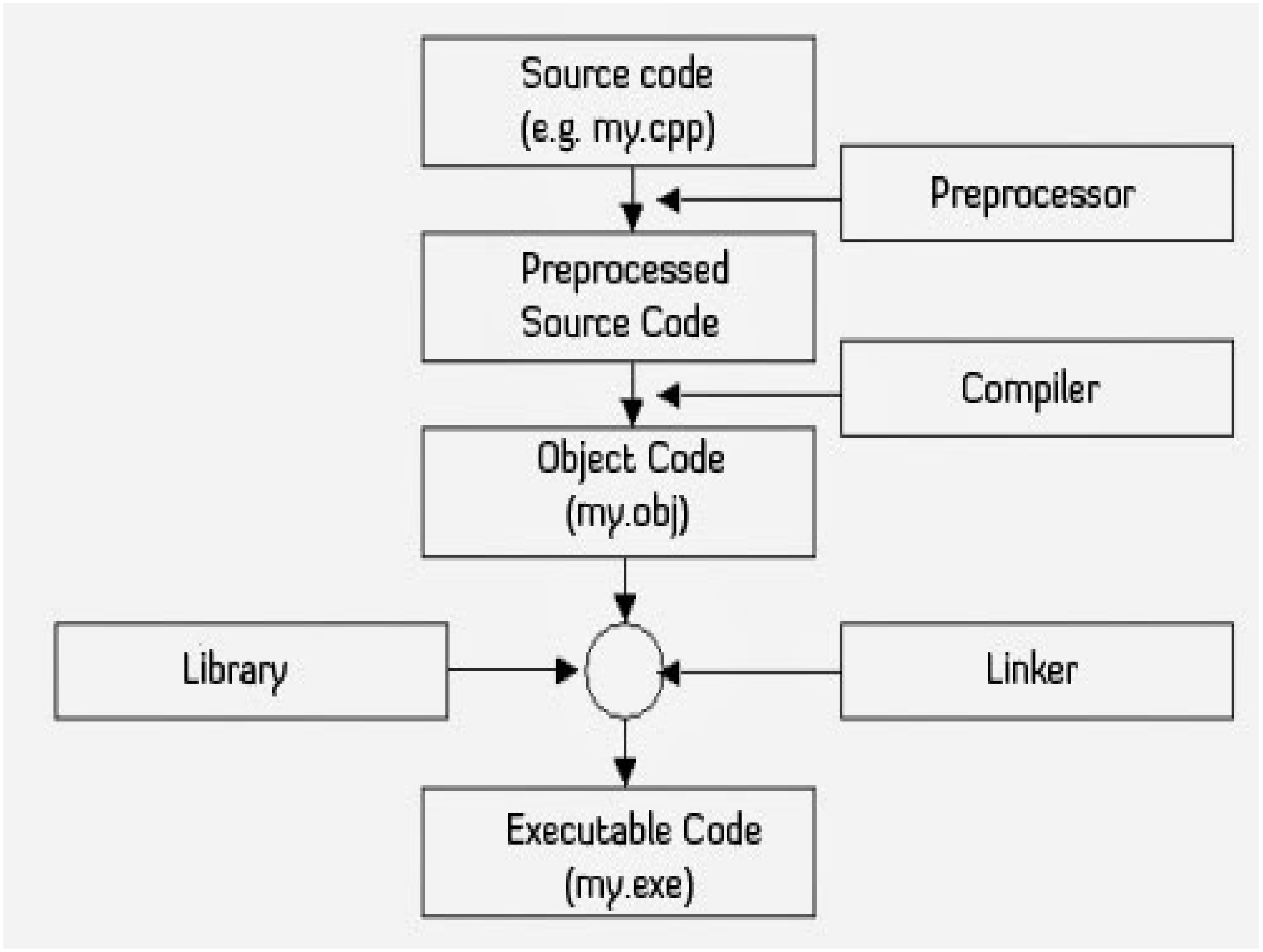
❑ defined symbols, which allow it to be called by other modules,

❑ undefined symbols, which call the other modules where these symbols are defined, and

❑ local symbols, used internally within the object file to facilitate **relocation**.

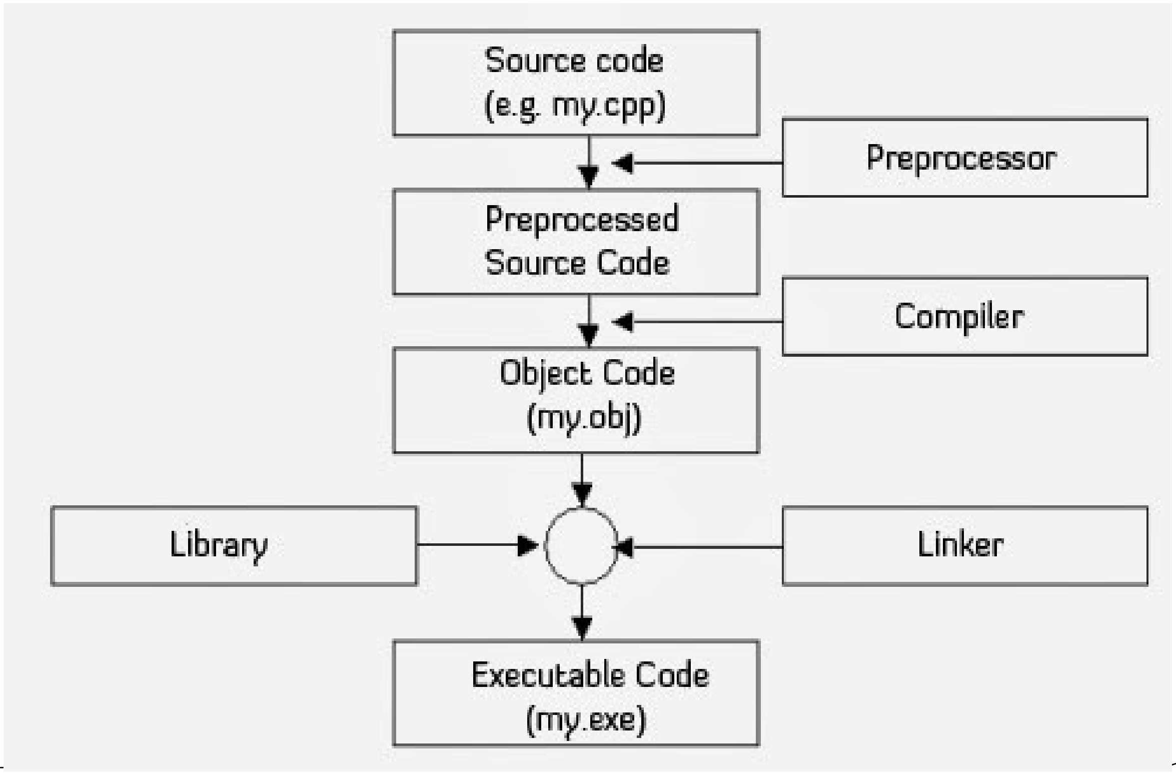
EXE FILE(Executable File)

- A file in a format that the computer can directly execute. Unlike source files, executable files cannot be read by humans.
- **EXE** is a file extension for an executable file format.

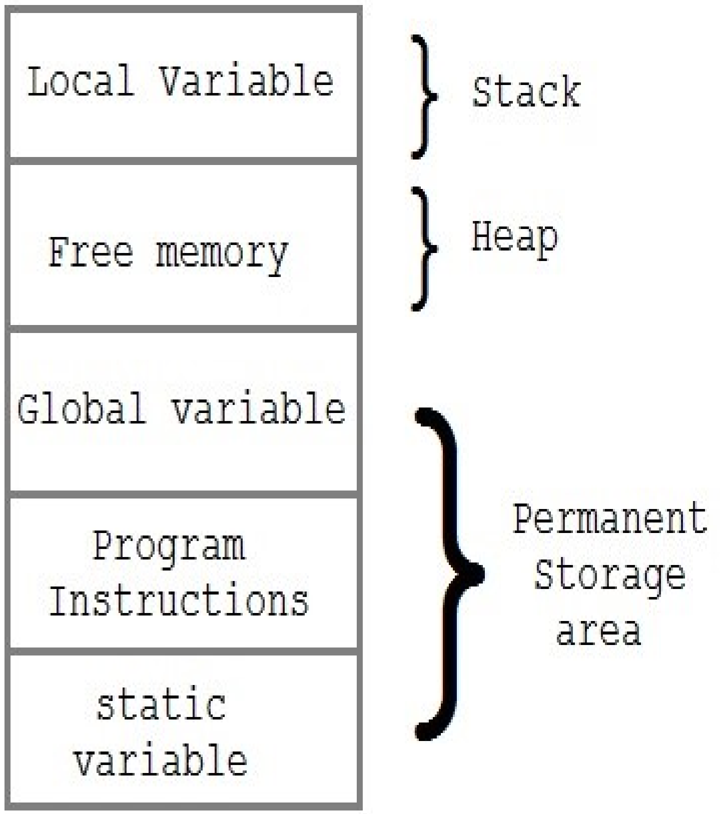


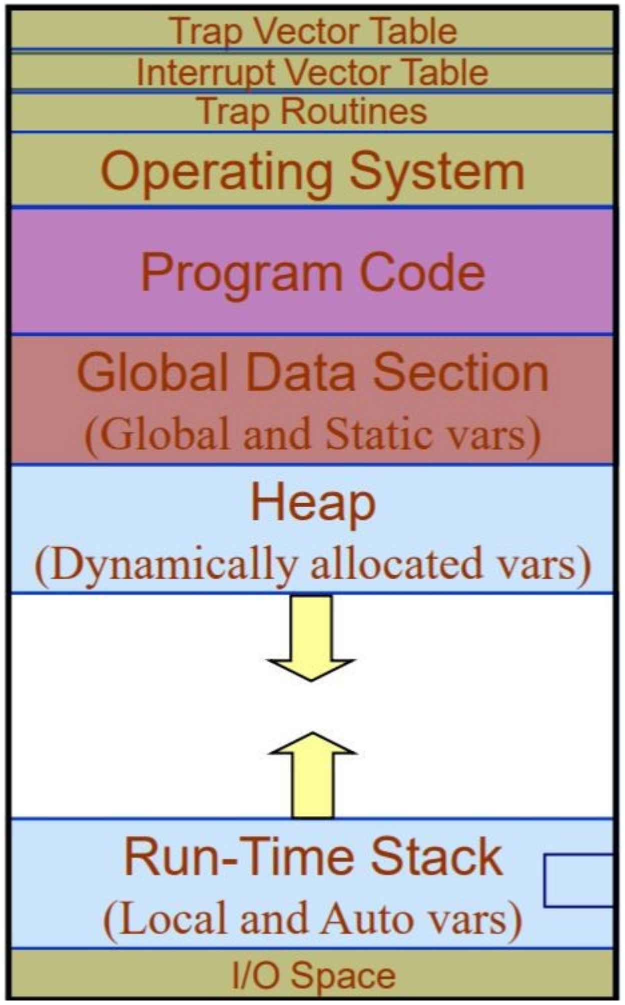
EXE FILE(Executable File)

- An executable is a file that contains a program - that is, a particular kind of file that is capable of being executed or run as a program in the computer.
- An executable file can be run by a program in Microsoft DOS or Windows through a command or a double click.
- **COM** and **BAT** are other types of executable file types in Windows.



Static and dynamic memory allocation in C





Memory layout of C Program

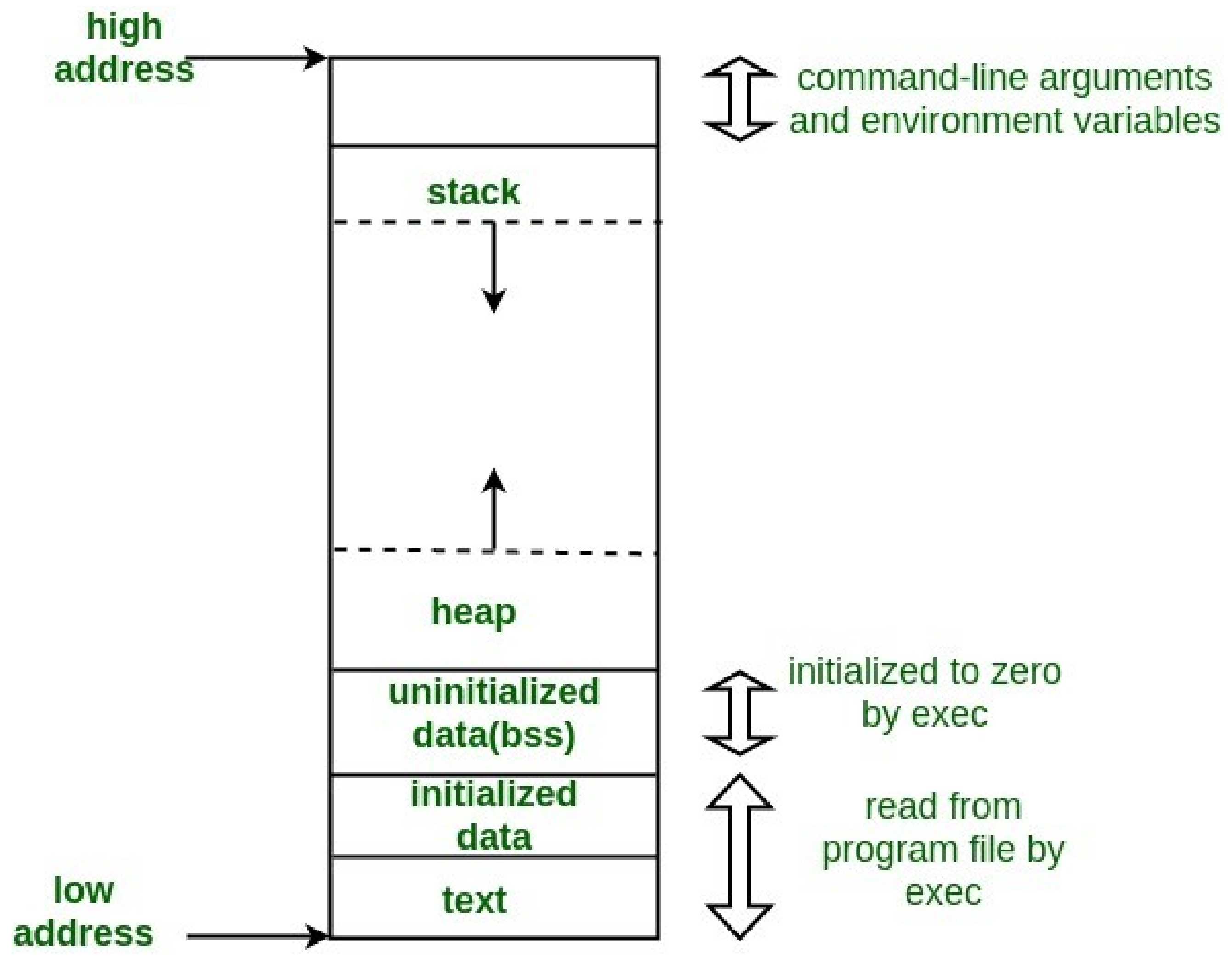
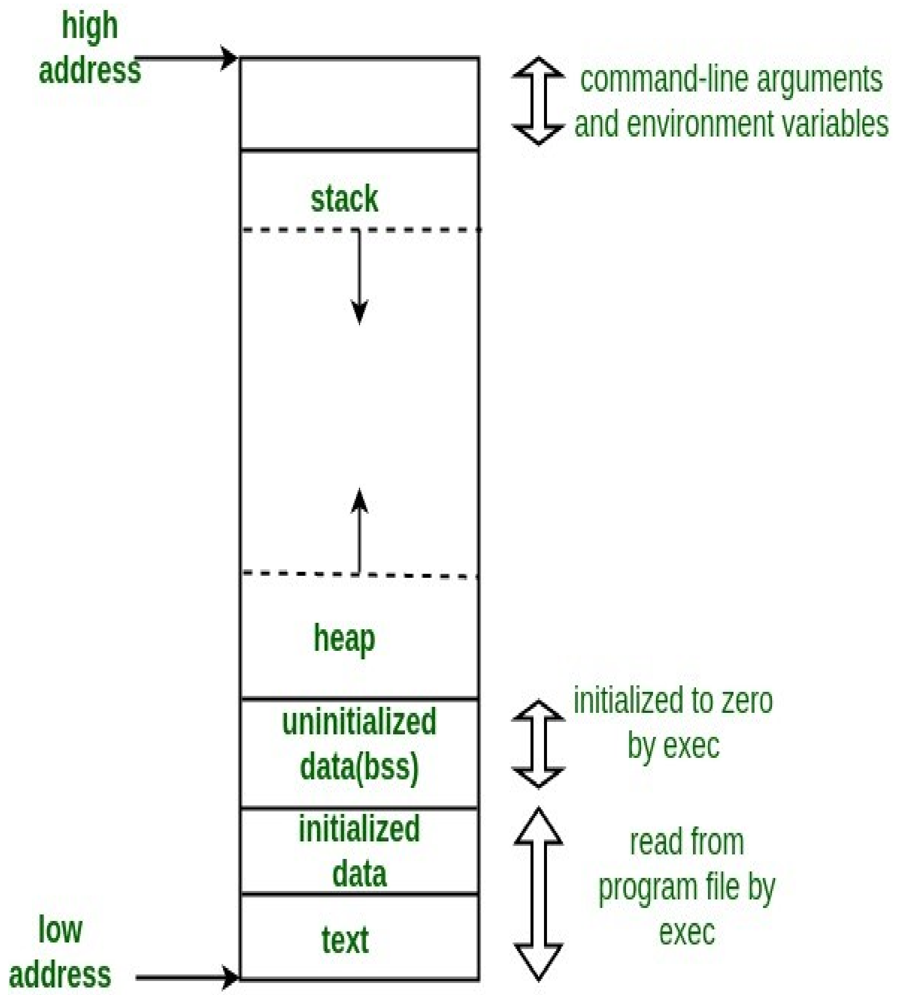

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 1

Memory layout of C Program

A typical memory representation of a C program consists of the following sections.

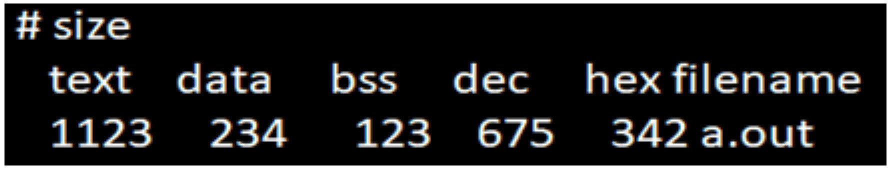
1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Memory layout of C Program

- The size command in Linux is a very important command that will **allow to list the section size and the total size of the object files** or the archived files in its argument list.
- When we do not specify the object file in the parameter list, then by default, 'a. out' file is used.

Example:



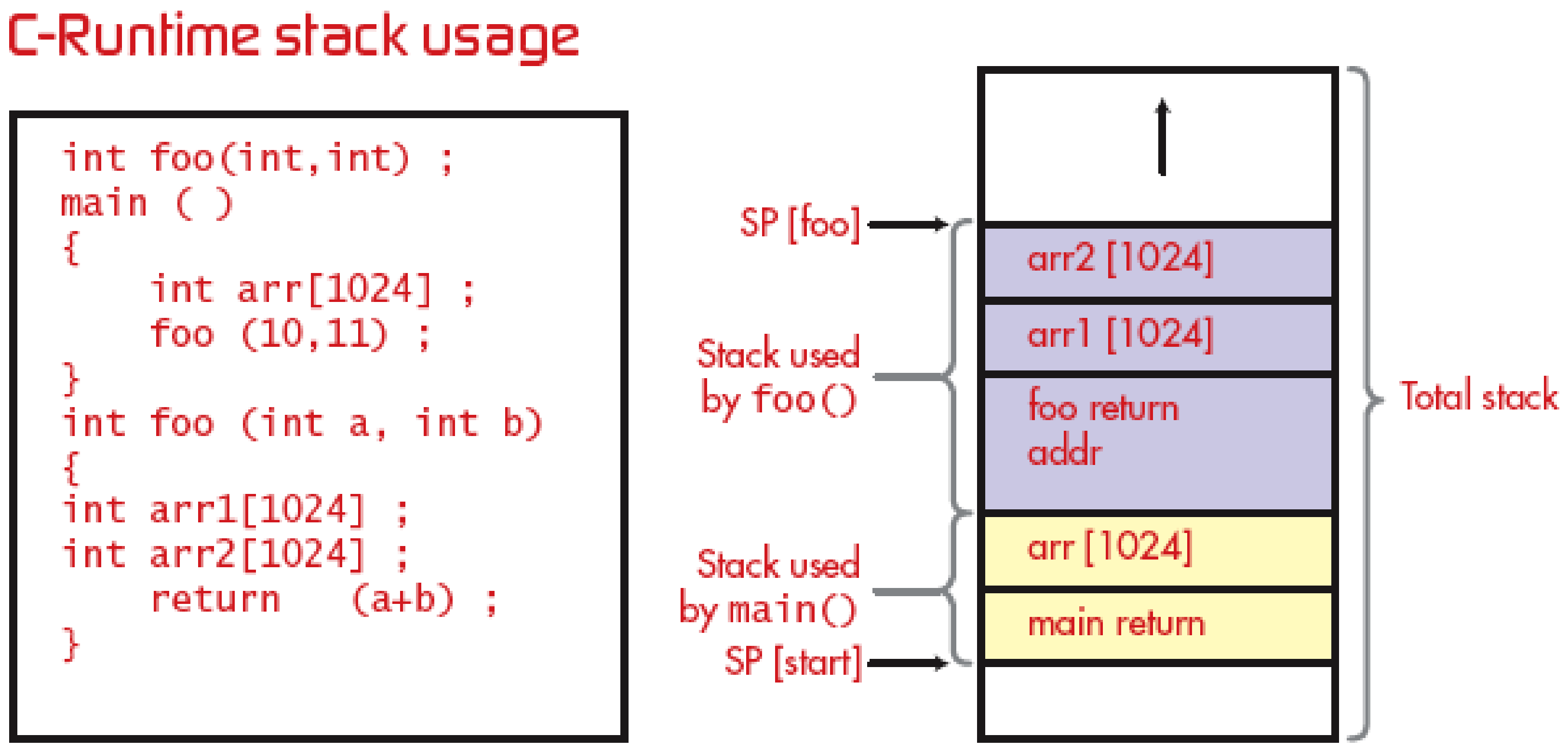
```
# size
  text   data    bss    dec     hex filename
  1123    234    123    675     342 a.out
```

$ gcc my.c -o my

$ size my

- The first three entries are for text, data, and bss sections, with their corresponding sizes.
- Then comes the total in decimal and hexadecimal formats. And finally, the last entry is for the filename.

Run time stack



C-Runtime stack usage

```
int foo(int,int) ;
main ( )
{
    int arr[1024] ;
    foo (10,11) ;
}
int foo (int a, int b)
{
int arr1[1024] ;
int arr2[1024] ;
    return   (a+b) ;
}
```

Figure 2

Why Space Complexity

- When memory was expensive we focused on making programs as **space** efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)
- The space complexity is used to estimate the size of the largest problem a program can solve.
- If a program is to run on a multi-user computer system, then it is required to specify the amount of memory to be allocated to the program.

Space Complexity (cont'd) $S(P)=C+S_P(I)$

The space needed by an algorithm is the sum of a fixed part and a variable part

Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:

- - e.g. name of the data collection
- same size for classifying 2GB or 1MB of texts

Variable part: Space needed by variables, whose size is dependent on the size of the problem:

- - e.g. actual text
- - load 2GB of text VS. load 1MB of text

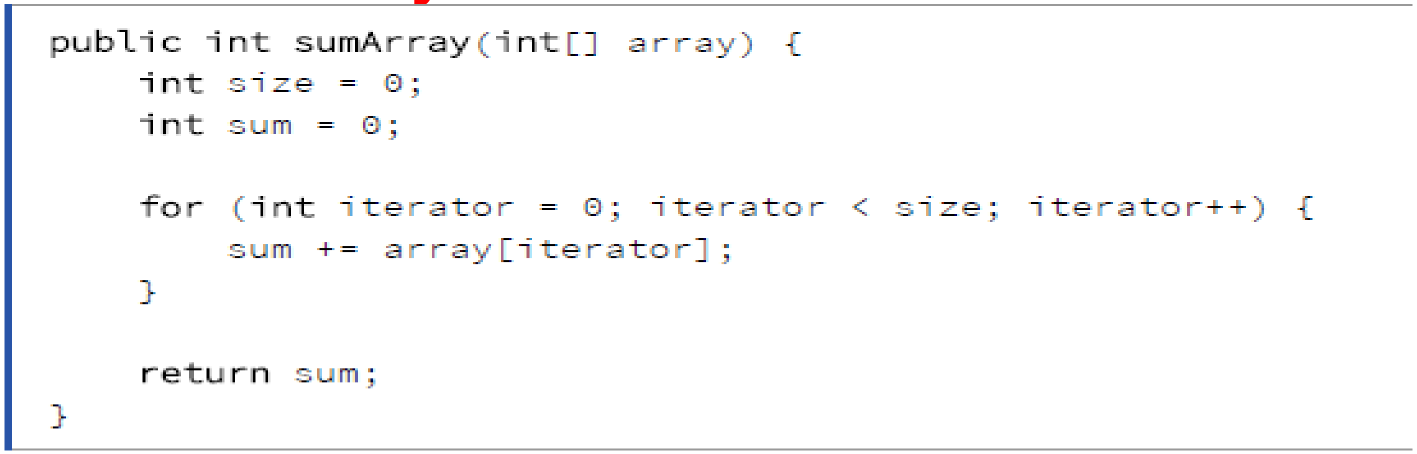
Space Complexity $S(P)=C + S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with **I**
  - **recursive stack space**, formal parameters, local variables, return address

Space Complexity Cont…

- **A variable part of an algorithm is consists of the space needed by component variable whose size depends on the particular problem instance being solved**
- The variable part includes space for
  - Component variables whose size is dependant on the particular problem instance being solved
  - Recursion stack space
  - Etc..

Space complexity of a program that sums all integer elements in an array:



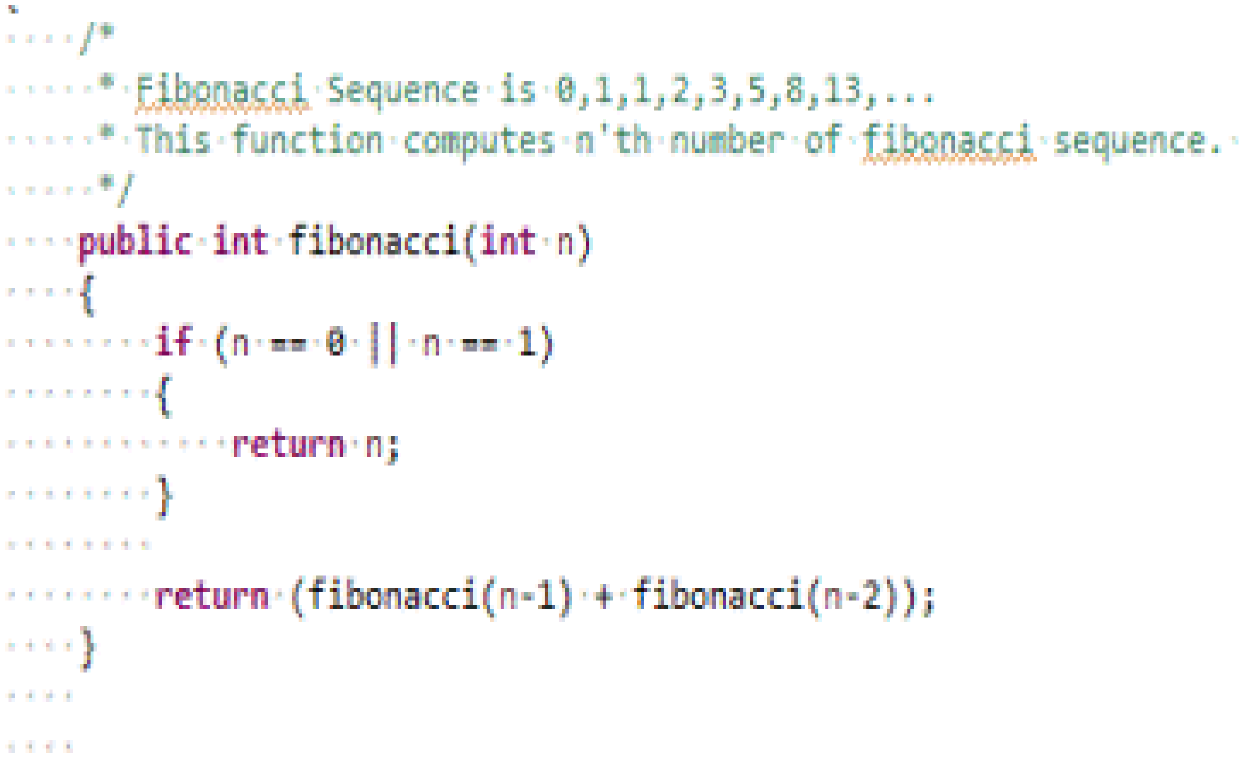
```
public int sumArray(int[] array) {
    int size = 0;
    int sum = 0;

    for (int iterator = 0; iterator < size; iterator++) {
        sum += array[iterator];
    }

    return sum;
}
```

Space by all variables present in the above code:

- *array* – the function's only argument – the space taken by the array is equal **4*n* bytes** where *n* is the length of the array
- *size* – a 4-byte integer
- *sum* – a 4-byte integer
- *iterator* – a 4-byte integer
- The total space needed for this algorithm to complete is **4*n*** **+ 4 + 4 + 4** (bytes). The highest order of *n* in this equation is just *n*. Thus, the space complexity of that operation is ***O(n)***.

Space complexity of recursive algorithm

- The function 'fibonacci(int n)' computes n'th number of fibonacci sequence.
- In general, if f(n) denotes n'th number of fibonacci sequence then f(n) = f(n-1) + f(n-2).
- For this recurrence relation, f(0) = 0 and f(1) = 1 are terminating conditions.



```
    /*
     * Fibonacci Sequence is 0,1,1,2,3,5,8,13,...
     * This function computes n'th number of fibonacci sequence.
     */
    public int fibonacci(int n)
    {
        if (n == 0 || n == 1)
        {
            return n;
        }

        return (fibonacci(n-1) + fibonacci(n-2));
    }
```

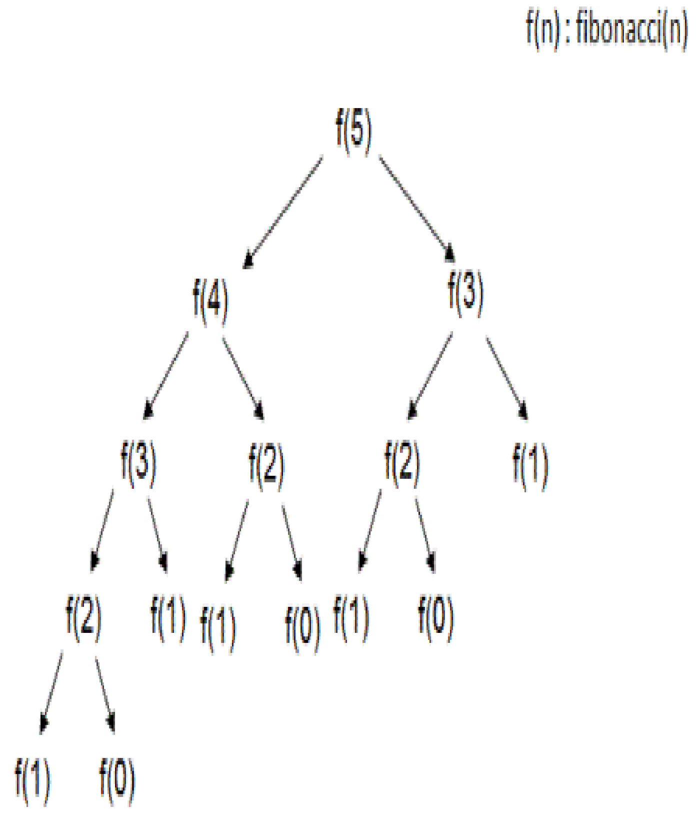
Time complexity of recursive algorithm

In this recursion tree, each state(except f(0) and f(1)) generates two additional states and total number of states generated are 15.

In general, total number of states are approximately equal to 2^n for computing n'th fibonacci number(f(n)).

Each state denotes a function call to 'fibonacci()' which does nothing but to make another recursive call.

The total time taken to compute nth number of fibonacci sequence is $O(2^n)$.

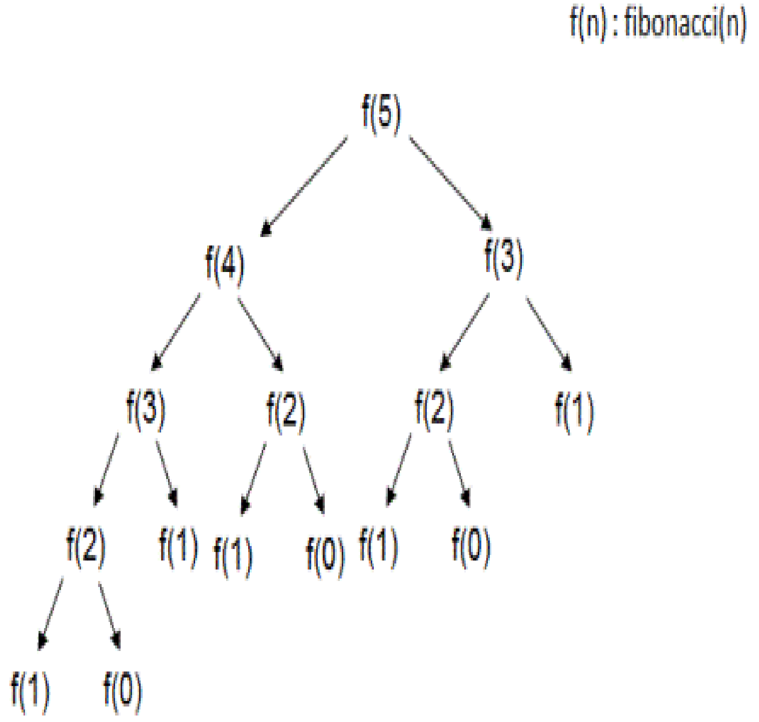


Recursion tree generated for computing 5th number of fibonacci sequence

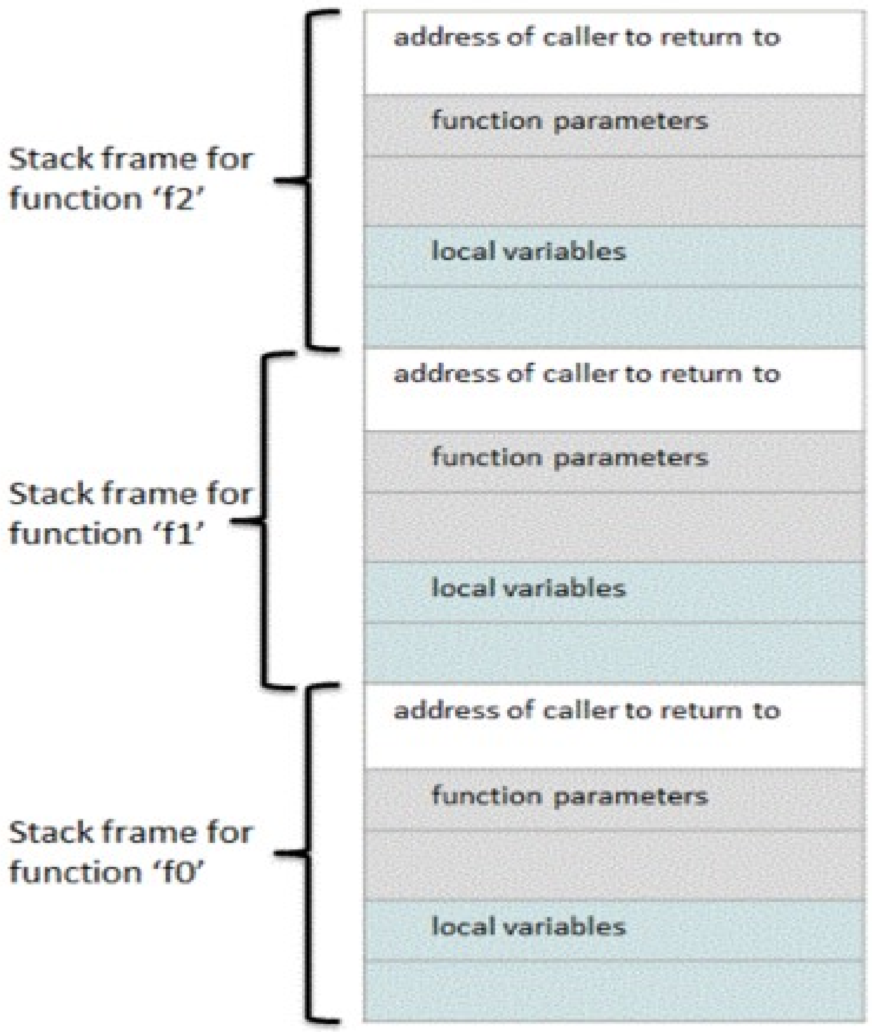
Space complexity of recursive algorithm

- Let's try to understand in general when the stack frames are generated and for how long are they kept in the memory?
- When a function 'f1' is called from function 'f0', stack frame corresponding to this function 'f1' is created.
- This stack frame is kept in the memory until call to function 'f1' is not terminated.
- This stack frame is responsible for saving the parameters of function 'f1', local variables in the function 'f1' and return address of caller function(function 'f0').
- Now when this function 'f1' calls another function 'f2', stack frame corresponding to 'f2' is also generated and is kept in the memory until call to 'f2' is not terminated.
- When call to 'f2' is made, the call stack which has stack frames in it looks like following -

Runtime stack space for recursive algorithm



Recursion tree generated for computing 5th number of fibonacci sequence


Stack frame for function 'f2'
address of caller to return to
function parameters
local variables
Stack frame for function 'f1'
address of caller to return to
function parameters
local variables
Stack frame for function 'f0'
address of caller to return to
function parameters
local variables

Call stack in memory for call sequence f0->f1->f2

Space complexity of recursive algorithm

- In the recursion tree for computing 5th fibonacci number, when left and bottom most state f(1) is getting executed, the call sequence which led to this state would be f(5)->f(4)->f(3)->f(2)->f(1) and all the corresponding stack frames would be present in the memory and when f(1) is returned its stack frame would be removed from the memory(or call stack).
- Space complexity of recursive algorithm is proportional to maximum depth of recursion tree generated.
- If each function call of recursive algorithm takes **O(m)** space and if the **maximum depth of recursion tree** is **'n'** then space complexity of recursive algorithm would be **O(nm)**.

Space Complexity comparison of Sorting Algorithms

Algorithm	Data Structure	Worst Case Auxiliary Space Complexity
Quicksort	**Array**	**O(n)**
Mergesort	**Array**	**O(n)**
Heapsort	Array	O(1)
Bubble Sort	Array	O(1)
Insertion Sort	Array	O(1)
Select Sort	Array	O(1)
Bucket Sort	Array	O(nk)
Radix Sort	Array	O(n+k)

Time Complexity

Why Time Complexity ?

- Is the algorithm "fast enough" for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose

Why Time Complexity ?

- Often more important than space complexity
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us
- 3-4GHz processors on the market
  - still …
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 Terra HZ computer would take about 1 year to run to completion
- Algorithms running time is an important issue

Time Complexity

The time complexity of an algorithm is the amount of computer time required to run to completion

- The time complexity of a problem is
  - the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm.
- The exact number of steps will depend on exactly what machine or language is being used.
- To avoid that problem, the **Asymptotic notation** is generally used.

Time Complexity

$$T(P)=C+t_P(I)$$

- Compile time (C) independent of instance characteristics.
- Run (execution) time denoted as $t_P(Instance)$
- Many of the factors t_p depends on are not known at the time a program is conceived it, is reasonable to attempt only to estimate t_p

$$t_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n) \ldots$$

- Running time of an algorithm as a function of input size n for large n, where n denotes the instance characteristics.
- Expressed using only the highest-order term in the expression for the exact running time.

Time Complexity comparison of Sorting Algorithms

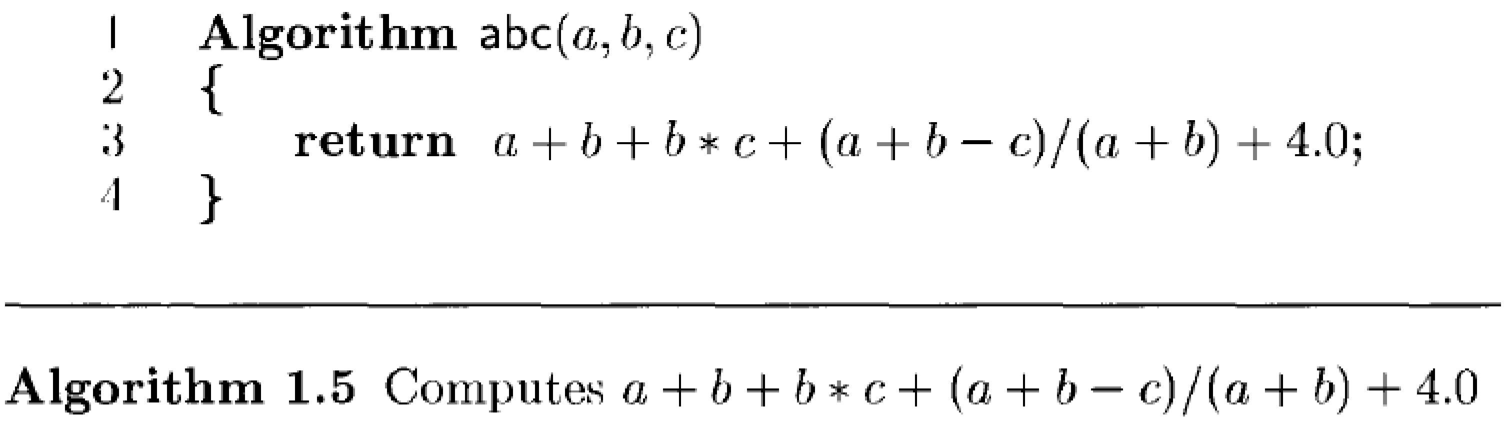
Algorithm	**Data Structure**	**Time Complexity**		
		Best	Average	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	Array	O(n)	$O(n^2)$	$O(n^2)$
Select Sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bucket Sort	Array	O(n+k)	O(n+k)	$O(n^2)$
Radix Sort	Array	O(nk)	O(nk)	O(nk)

Program Steps

- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example : **abc = a + b + b * c + (a + b - c) / (a + b) + 4.0**

Regard as the same unit
machine independent



```
Algorithm abc(a, b, c)
{
    return $a + b + b * c + (a + b - c)/(a + b) + 4.0$;
}
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

Methods to compute the step count

1. **Introduce variable count into programs**
2. **Tabular method**
   - Determine the total number of steps contributed by each statement

     step per execution × frequency
   - add up the contribution of all statements

Method –I: to determine step count

- In the first method, we introduce a new variable, **count**, into the program.
- This is a global variable with initial value **count =0.**
- Statements to **increment count** by the appropriate amount are introduced into the program.
- This is done so that each time a statement in the original program is executed, count is incremented by the **step count** of that statement.

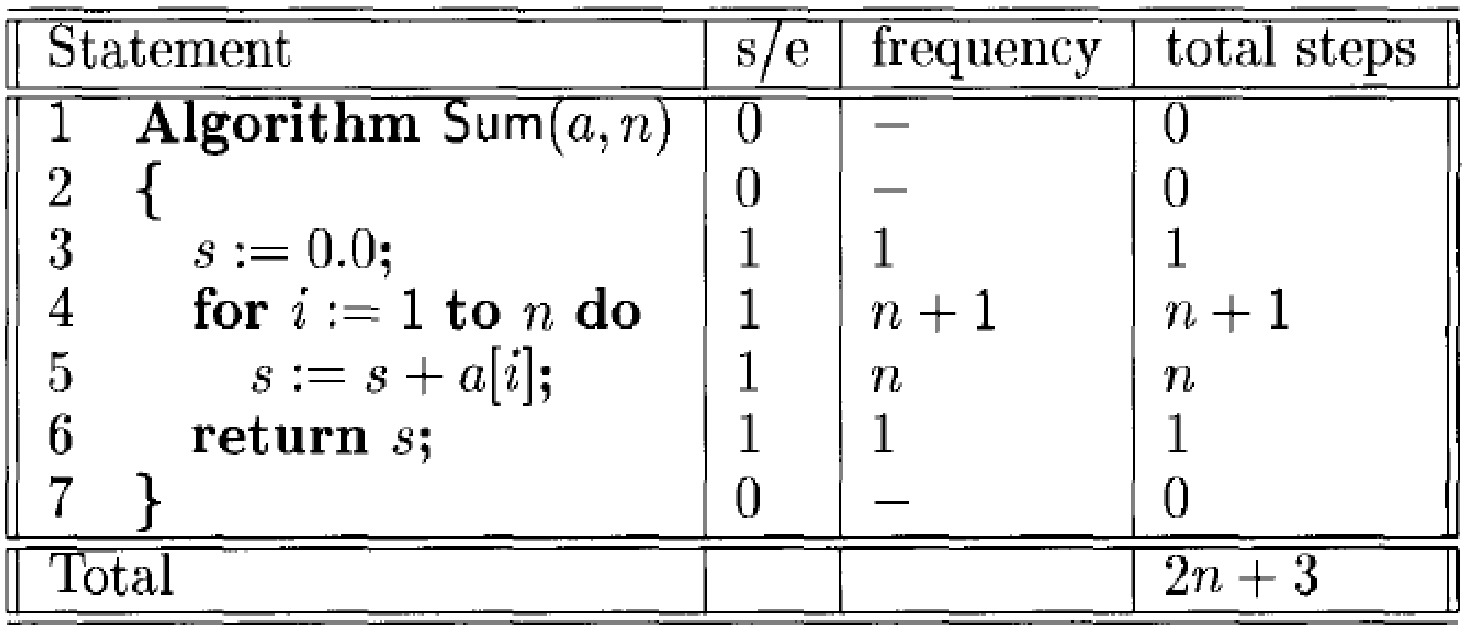
Algorithm 1.6 with count statement

```
**Algorithm** Sum($a, n$)
{
    $s := 0.0$;
    $count := count + 1$; // $count$ is global; it is initially zero.
    **for** $i := 1$ **to** $n$ **do**
    {
        $count := count + 1$; // For **for**
        $s := s + a[i]$; $count := count + 1$; // For assignment
    }
    $count := count + 1$; // For last time of **for**
    $count := count + 1$; // For the **return**
    **return** $s$;
}
```

```
**Algorithm** Sum($a, n$)
{
    $s := 0.0$;
    **for** $i := 1$ **to** $n$ **do**
        $s := s + a[i]$;
    **return** $s$;
}
```

Algorithm 1.6 Iterative function for sum

Step table for Algorithm 1.6



Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

Table 1.1 Step table for Algorithm 1.6

Algorithm 1.6 with count statement

```
Algorithm Sum($a, n$)
{
    $s := 0.0;$
    $count := count + 1;$ // $count$ is global; it is initially zero.
    for $i := 1$ to $n$ do
    {
        $count := count + 1;$ // For for
        $s := s + a[i];$ $count := count + 1;$ // For assignment
    }
    $count := count + 1;$ // For last time of for
    $count := count + 1;$ // For the return
    return $s$;
}
```

```
Algorithm Sum($a, n$)
{
    for $i := 1$ to $n$ do $count := count + 2;$
    $count := count + 3;$
}
```

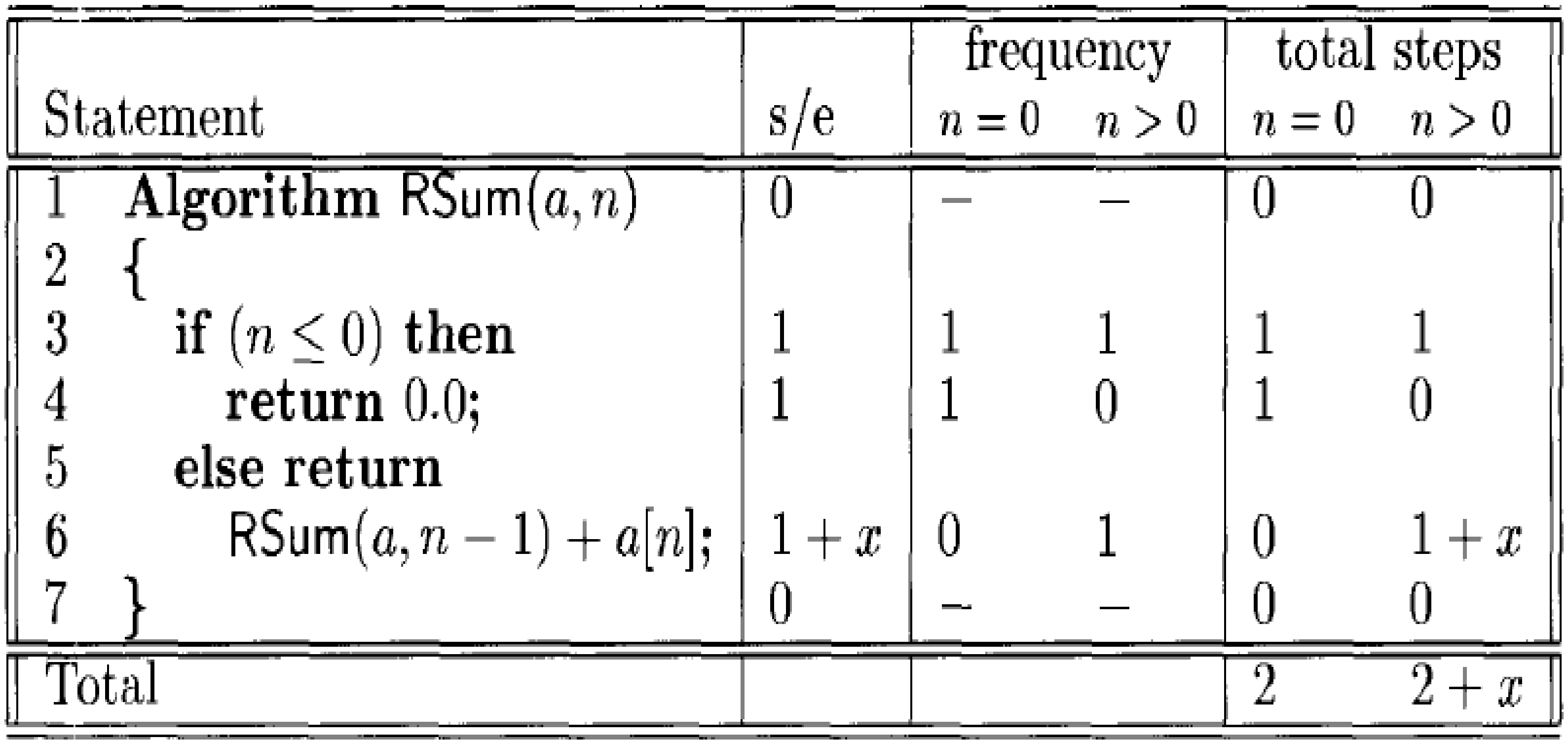
Algorithm 1.7 with count statement

```
Algorithm RSum($a, n$)
{
    $count := count + 1$; // For the if conditional
    if ($n \leq 0$) then
    {
        $count := count + 1$; // For the return
        return 0.0;
    }
    else
    {
        $count := count + 1$;  // For the addition, function
                               // invocation and return
        return RSum($a, n - 1$) + $a[n]$;
    }
}
```

```
Algorithm RSum($a, n$)
{
    if ($n \leq 0$) then return 0.0;
    else return RSum($a, n - 1$) + $a[n]$;
}
```

Algorithm 1.7 Recursive function for sum

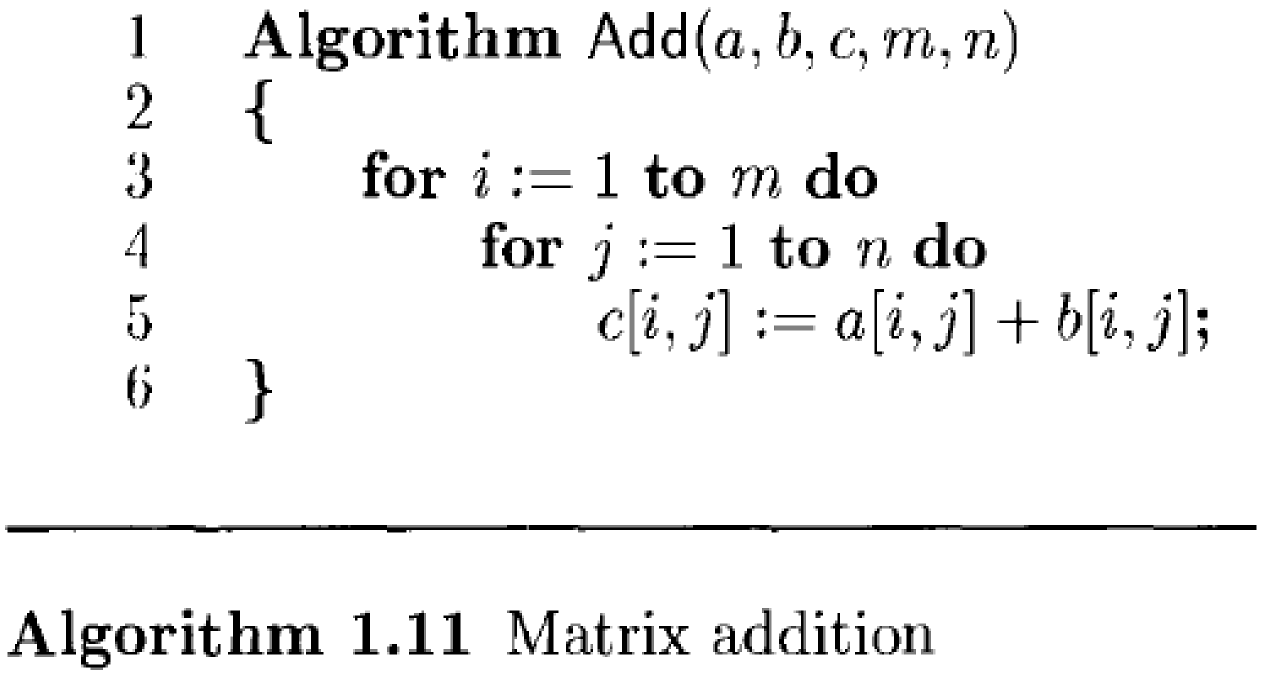
Step table for Algorithm 1.7



Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \le 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

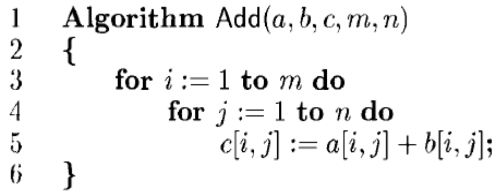
Matrix addition



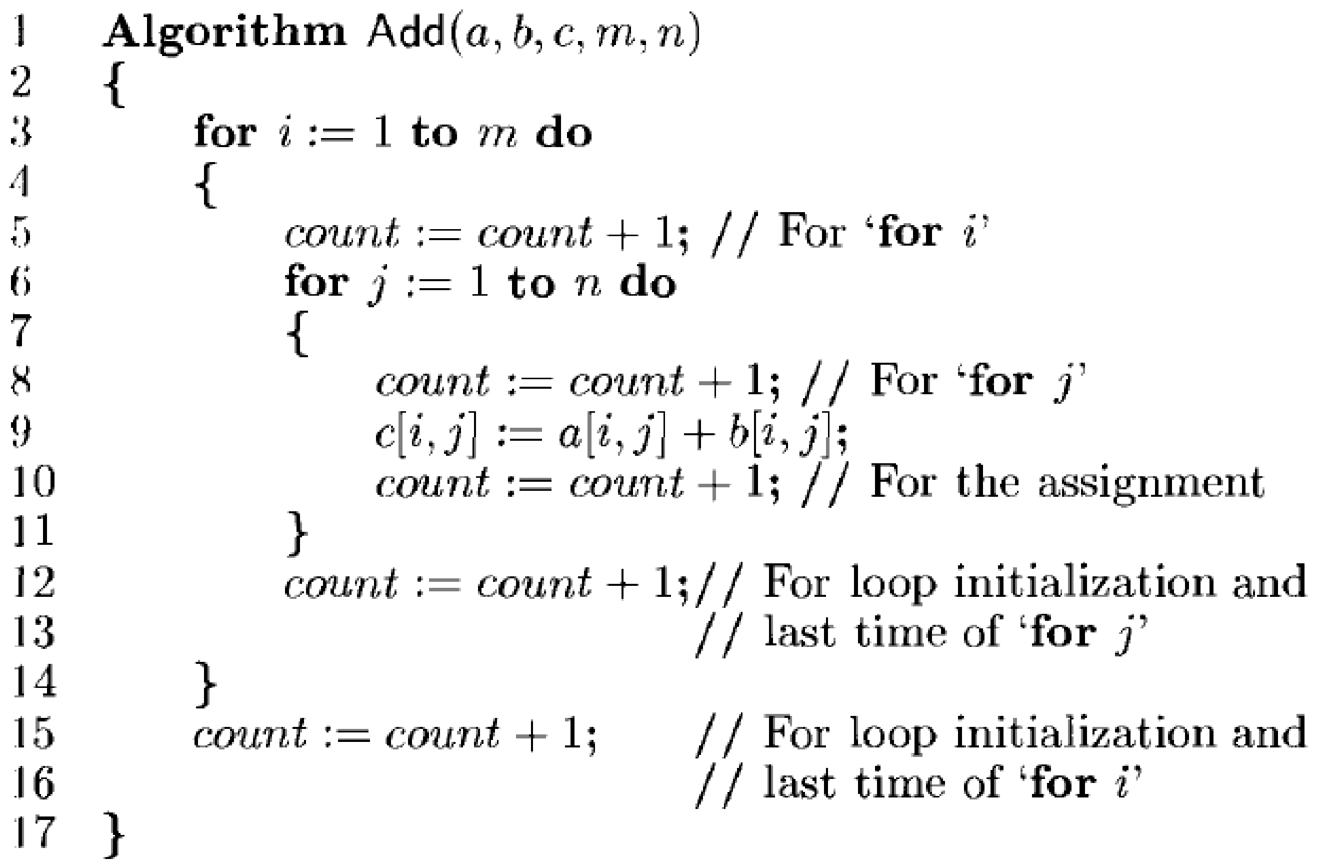
```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
        for $j := 1$ to $n$ do
            $c[i, j] := a[i, j] + b[i, j]$;
}
```

Algorithm 1.11 Matrix addition

Matrix addition



```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
        for $j := 1$ to $n$ do
            $c[i, j] := a[i, j] + b[i, j]$;
}
```



```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
    {
        $count := count + 1$; // For 'for $i$'
        for $j := 1$ to $n$ do
        {
            $count := count + 1$; // For 'for $j$'
            $c[i, j] := a[i, j] + b[i, j]$;
            $count := count + 1$; // For the assignment
        }
        $count := count + 1$;// For loop initialization and
                             // last time of 'for $j$'
    }
    $count := count + 1$;    // For loop initialization and
                             // last time of 'for $i$'
}
```

Algorithm 1.12 Matrix addition with counting statements

Matrix addition

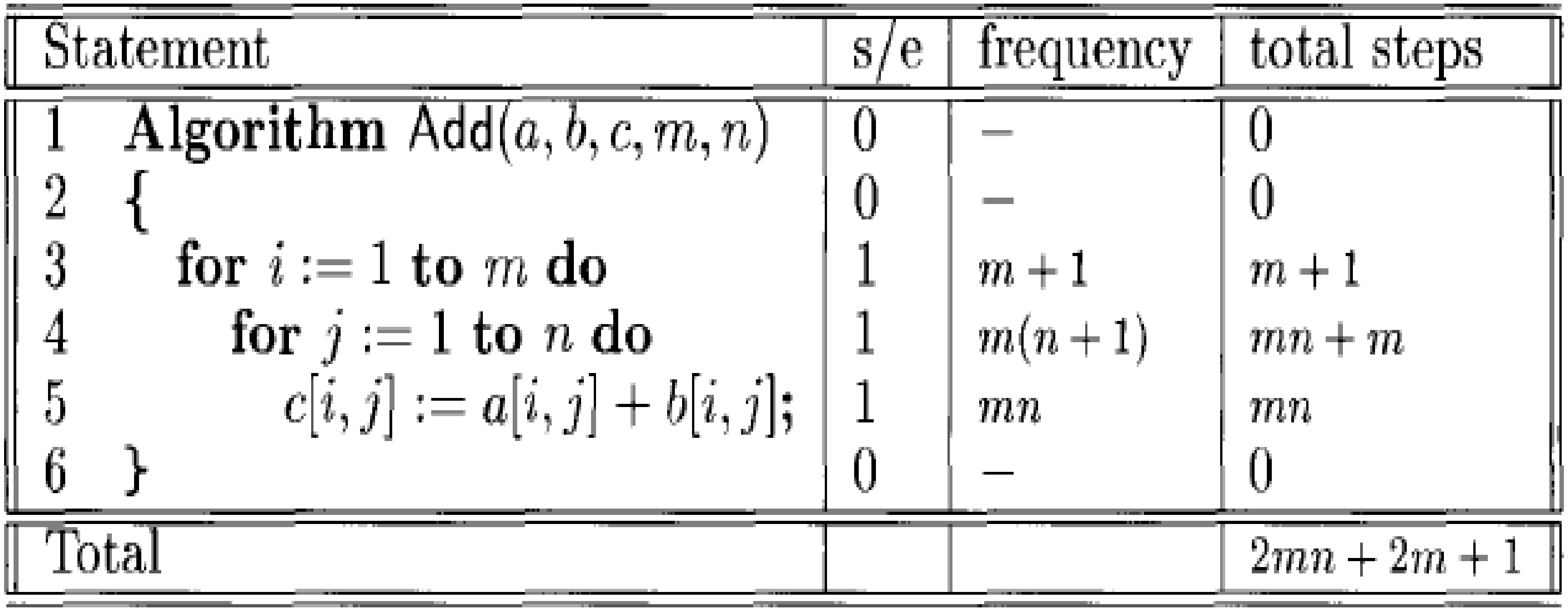
```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
    {
        $count := count + 2$;
        for $j := 1$ to $n$ do
            $count := count + 2$;
    }
    $count := count + 1$;
}
```

Algorithm 1.13 Simplified algorithm with counting only

Method –II: to determine step count

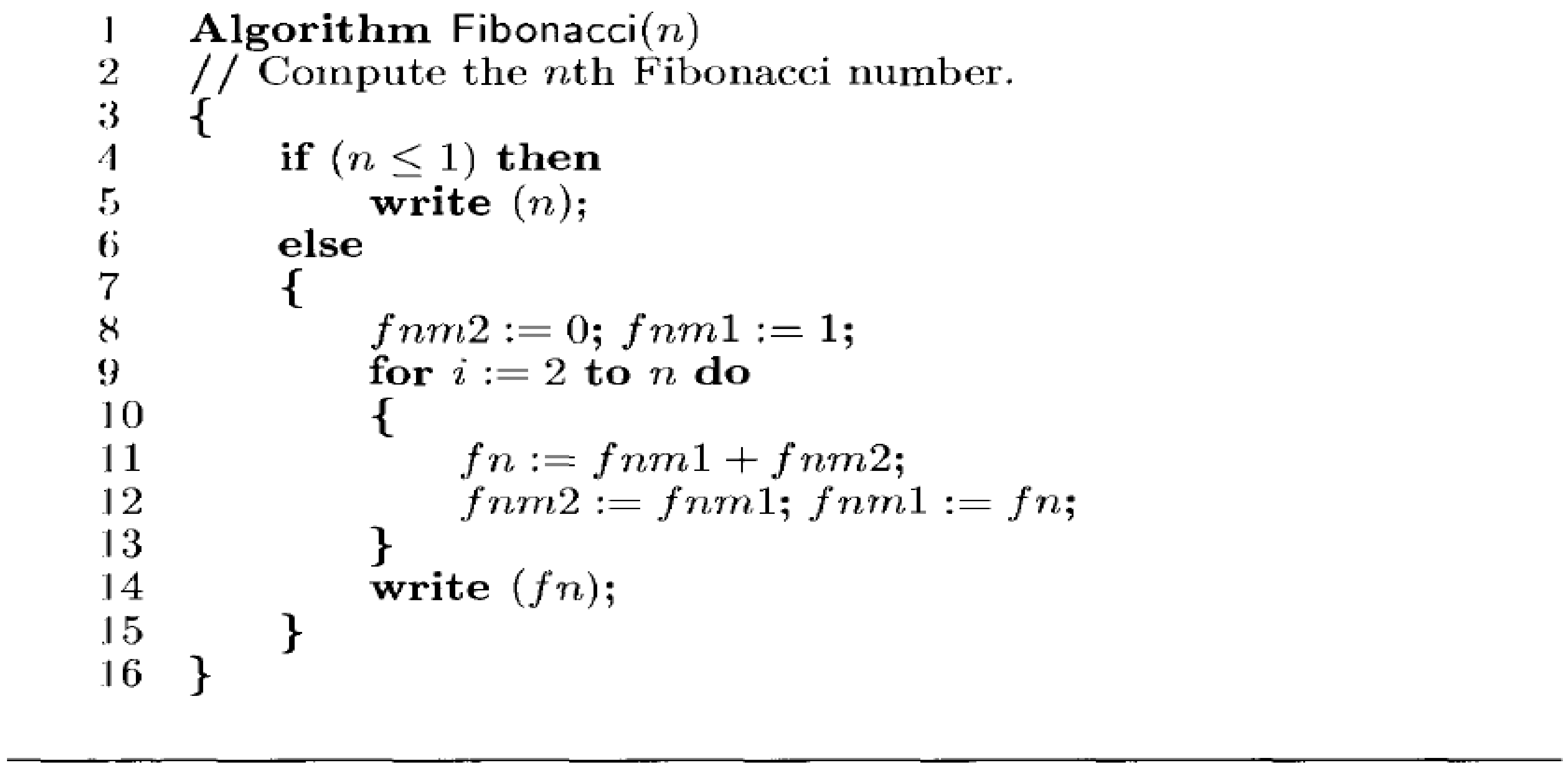
- To determine the **step count** of an **algorithm**, we first determine the number of **steps** per execution (s/e) of each statement and the total number of times (i.e., frequency) each statement is executed.
- The step count method is one of the method to analyze the algorithm. In this method, we count number of times one instruction is executing.
- The table lists the total number of steps per execution(s/e) of the statement and the total number of times(i.e., frequency)each statement is executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

Matrix Addition



Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j];$	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

Algorithm n^{th} Fibonacci



```
Algorithm Fibonacci($n$)
// Compute the $n$th Fibonacci number.
{
    if ($n \leq 1$) then
        write ($n$);
    else
    {
        $fnm2 := 0;$ $fnm1 := 1;$
        for $i := 2$ to $n$ do
        {
            $fn := fnm1 + fnm2;$
            $fnm2 := fnm1;$ $fnm1 := fn;$
        }
        write ($fn$);
    }
}
```

Algorithm 1.14 Fibonacci numbers

Fibonacci(Algorithm 1.14) takes as input any non-negative integer n and prints the value f_n.

Time complexity of Algorithm Fibonacci(n)

Case 1: n = 0 or 1.

- When n = 0 or 1, lines4 and 5 get executed once each. Since each line has an s/e of 1,the total step count for this case is 2.

Case 2: n > 1.

- When n > 1, lines4, 8,and 14 are each executed once. Line 9 gets executed n times, and lines11and 12get executed (n-1) times each.
- Line8 has an s/e of 2, line12 has an s/e of 2,and line13 has an s/e of 0.
- The remaining lines that get executed have s/e‘s of 1. The total Steps for the case n > 1 is therefore **4n + 1**.

Summary

- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for.
- The number of steps is itself a function of the instance characteristics Although any specific instance may have several characteristics(e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number of steps is computed as a function of some subset of these.
- Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in run time as the instance characteristics change.
- Determining the exact step count(best case, worst case, or average)of an algorithm can prove to be an exceedingly difficult task.
- Expending immense effort to determine the step count exactly is not a very worth while endeavor, since the notion of a step is itself inexact

Amortized Analysis

Amortized Analysis

- In computer science, **amortized analysis** is a method for analyzing a given algorithm's time complexity, or how much of a resource, especially time or memory, it takes to execute.
- In an ***amortized analysis,*** *the* ***time required to perform a sequence of data-structure*** **operations** is averaged over all the operations performed.
- Amortized analysis can be used to show that the **average cost** of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from **average-case analysis** in that probability is not involved.

Amortized Analysis

- In an ***amortized analysis***, the time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

Amortized complexity

- **Amortized complexity** is the total expense per operation, evaluated over a sequence of operations. The idea is to guarantee the total expense of the entire sequence, while permitting individual operations to be much more expensive than the **amortized** cost.

- Amortized complexity analysis is a different way to estimate run times. The main ideas is to spread out the cost of operations, charging more than necessary when they are cheap — thereby "saving up for later use" when they occasionally become expensive.

- Worst case analysis of run time complexity is often **too pessimistic**. Average case analysis **may be difficult** because (i) it is not clear what is "average data", (ii) uniformly random data is usually not average, (iii) probabilistic arguments can be difficult.

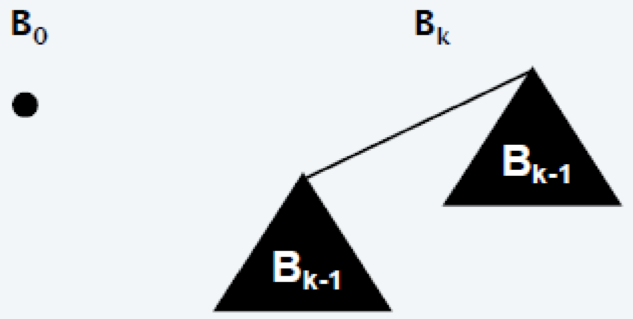
Amortized Analysis: Data structures

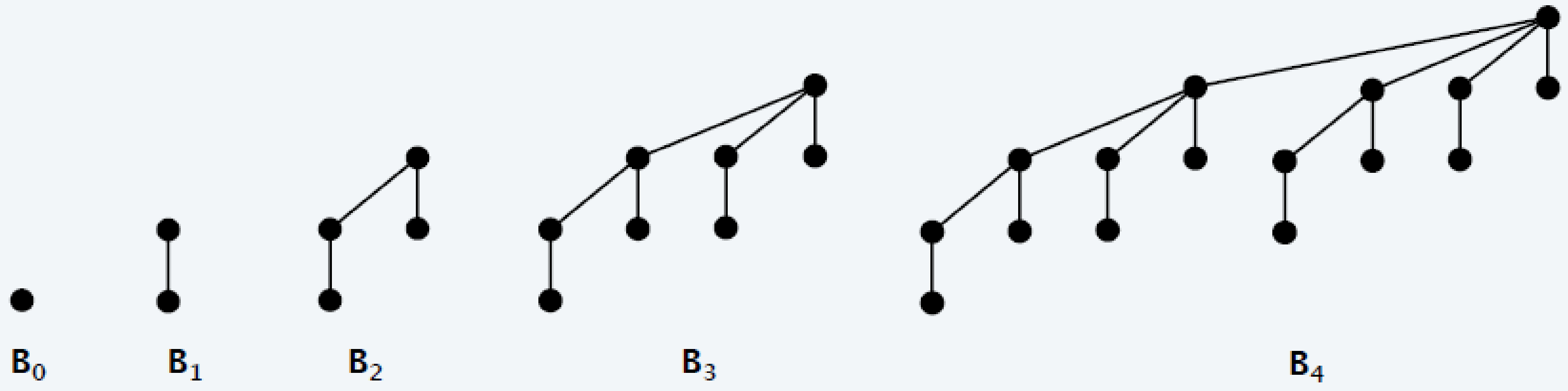
- Data structures typically support several different types of operations, each with its own cost (e.g., time cost or space cost).
- The idea behind amortized analysis is that, even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the average cost per operation is not so high.
- It is important to realize that these “average costs” are not expected values; there needn’t be any random events.
- Instead, we are considering the worst-case average cost per operation in a sequence of many operations.
- In other words, we are interested in the asymptotic behavior of the function **C(n) = 1/n . (worst-case total cost of a sequence of n operations)**
- (possibly with some condition on how many times each type of operation may occur). “Worst-case” means that no adversary could choose a sequence of n operations that gives a worse running time.

Binomial tree

Def. A binomial tree of order k is defined recursively:

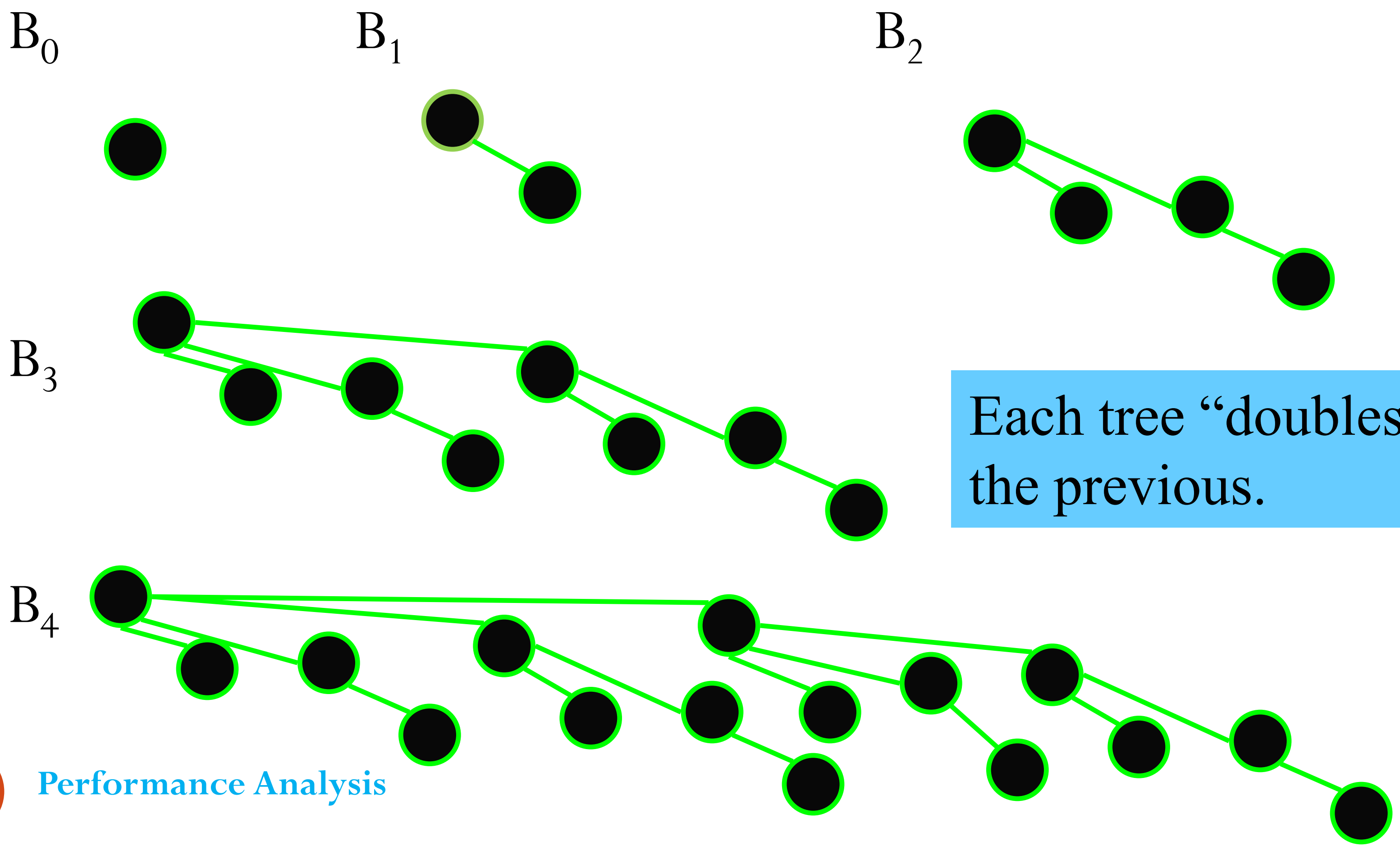
- Order 0: single node.
- Order k: one binomial tree of order $k-1$ linked to another of order $k-1$.





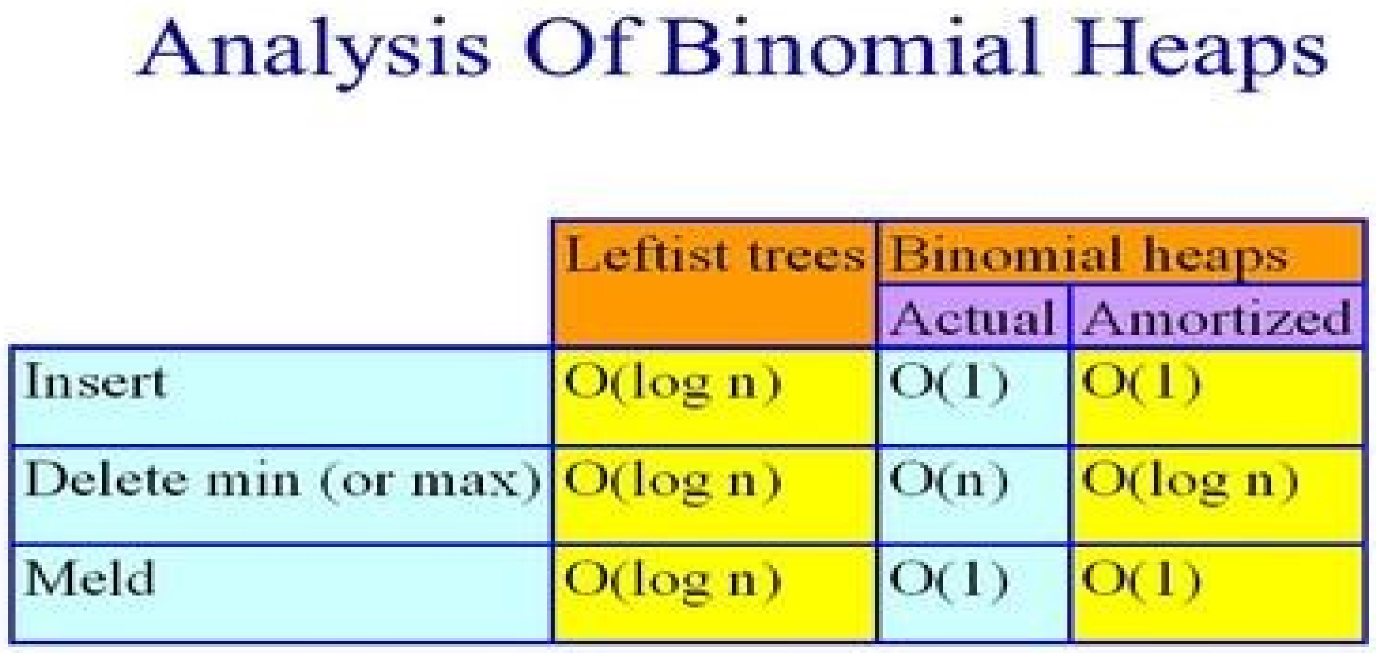
Binomial Queue

- **Binomial Trees:**



Each tree "doubles" the previous.

Binomial Queue



Analysis Of Binomial Heaps

	Leftist trees	Binomial heaps	
		Actual	Amortized
Insert	O(log n)	O(1)	O(1)
Delete min (or max)	O(log n)	O(n)	O(log n)
Meld	O(log n)	O(1)	O(1)

Amortized Analysis

- Suppose we have an ADT and we want to analyze its operation using amortized time analysis.
- Amorized time analysis is based on the following equation, which applies to each individual operation of this ADT.

amortized cost = actual cost + accounting cost

- The creative part is to design a system of accounting costs for individual operations that achieves the two goals:

1. In any legal sequence of operations, beginning from the creation of the ADT object being analyzed, the sum of the accounting cost is nonnegative.
2. Although the actual cost may fluctuate widely from one individual operation to the next, it is feasible to analyze the amortized cost of each operation.

Amortized Analysis:

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.

Probabilistic analysis:

- Average case running time: average over all possible inputs for one algorithm (operation).
- If using probability, called expected running time.

Amortized analysis:

- No involvement of probability
- Average performance on a sequence of operations, even some operation is expensive.
- Guarantee average performance of each operation among the sequence in **worst case**.

Three Methods of Amortized Analysis

- **Aggregate analysis:**
  - Total cost of *n* operations/*n*,
- **Accounting method:**
  - Assign each type of operation an (different) amortized cost
  - overcharge some operations,
  - store the overcharge as credit on specific objects,
  - then use the credit for compensation for some later operations.
- **Potential method:**
  - Same as accounting method
  - But store the credit as "potential energy" and as a whole.

Three Methods of Amortized Analysis

1. **Aggregate Method**: we determine an upper bound T(n) on the total sequence of n operations. The cost of each will then be T(n)/n.

2. **Accounting Method**: we overcharge some operations early and use them to as prepaid charge later.

3. **Potential Method**: we maintain credit as potential energy associated with the structure as a whole.

- The other two methods we shall study, the ***accounting method*** and the ***potential method***, may assign different amortized costs to different types of operations.

[1] Aggregate Method

- In the ***aggregate method*** of amortized analysis, we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total.
- In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)$ / n.
- The amortized cost applies to each operation, even when there are several types of operations in the sequence.

Example: Stack operations

- The two fundamental stack operations, each of which takes $O(1)$ time:
- **PUSH**(S, x) pushes object x onto stack S.
- **POP**(S) pops the top of stack S and returns the popped object.
- Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore (n).

Aggregate Method: Stack operations

- Let us add the stack operation MULTIPOP(*S*, *k*), which removes the *k* top objects of stack *S*, or pops the entire stack if it contains less than *k* objects.
- In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

```
MULTIPOP(S,k)

1  while not STACK-EMPTY(S) and $k \neq 0$

2      do POP(S)

3          $k \leftarrow k - 1$
```

- The running time of MULTIPOP(*S*, *k*) on a stack of *s* objects?

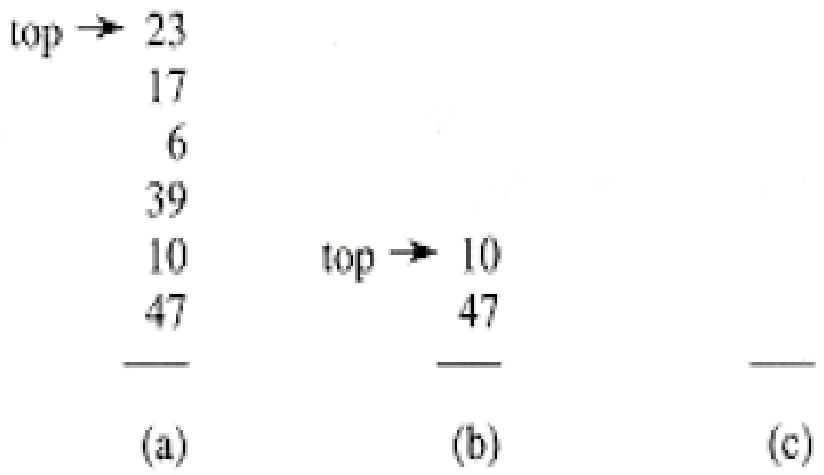
Aggregate Method: Stack operations

- The running time of MULTIPOP(*S*, *k*) on a stack of *s* objects?
- The actual running time is linear in the number of POP operations actually executed, and thus it suffices to analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP.
- The number of iterations of the **while** loop is the number min(*s*, *k*) of objects popped off the stack. For each iteration of the loop, one call is made to POP in line 2. Thus, the total cost of MULTIPOP is min(*s*, *k*), and the actual running time is a linear function of this cost.

```
MULTIPOP(S,k)
1  while not STACK-EMPTY(S) and $k \neq 0$
2       do POP(S)
3          $k \leftarrow k - 1$
```

Aggregate Method: The action of MULTIPOP on a stack S

- The top 4 objects are popped by MULTIPOP(S, 4), whose result is shown in (b)
- The next operation is MULTIPOP(S, 7), which empties the stack shown in (c)



Aggregate Method: Stack operations

- Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack.
- The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n.
- The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each.
- Although this analysis is correct, the $O(n^2)$ result, obtained by considering the worst-case cost of each operation individually, is not tight.
- Using the aggregate method of amortized analysis, we can obtain a better upper bound that considers the entire sequence of n operations.

Aggregate Method: Stack operations

- In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$. Why?
- Each object can be popped at most once for each time it is pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.
- For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. **The amortized cost of an operation is the average: $O(n)/n = O(1)$.**
- We emphasize again that although we have just shown that the average cost, and hence running time, of a stack operation is $O(1)$, no probabilistic reasoning was involved.
- We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

[2] Accounting Method

- In the ***accounting method*** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost.
- The amount we charge an operation is called its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as ***credit***.
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up.
- This is very different from the aggregate method, in which all operations have the same amortized cost.

Accounting Method

- If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence.
- Moreover, as in the aggregate method, this relationship must hold for all sequences of operations. Thus, the total credit associated with the data structure must be nonnegative at all times, since it represents the amount by which the total amortized costs incurred exceed the total actual costs incurred.
- If the total credit were ever allowed to become negative (the result of Undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Accounting Method: Stack

- When we push a plate onto a stack, we use \$1 to pay actual cost of the push and we leave \$1 on the plate. At any point, every plate on the stack has a dollar on top of it.
- When we execute a pop operation, we charge it nothing and pay its cost with the dollar that is on top of it.

PUSH	1 ,	PUSH	2 ,
POP	1 ,	POP	0 ,
MULTIPOP	min(k,s) ,	MULTIPOP	0 .

where k is the argument supplied to MULTIPOP and s is the stack size when it is called.

Accounting Method: Stack

- Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are $O(l)$, although in general the amortized costs of the operations under consideration may differ asymptotically.
- Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we put on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.
- The dollar stored on the plate is prepayment for the cost of popping it from the stack.

Accounting Method: Stack

- When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack.
- To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we needn't charge the POP operation anything.
- Moreover, we needn't charge MULTIPOP operations anything either.
- To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation.
- To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged at least enough up front to pay for MULTIPOP operations.

Accounting Method: Stack

- In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

[3] Potential Method

- Instead of representing prepaid work as credit stored with specific objects, the potential method represents the prepaid work as potential energy than can be released to pay for future operations.
- Potential is associated with the data structure as a whole rather than with specific objects.

Example:

- We start with an initial data structure D_0 on which n operations are performed.
- Let c_i be the cost the i^{th} operations and D_i be the data structure that results after applying the i^{th} operation to data structure D_{i-1}
- A potential function Ø maps Di to a real number Ø(Di) which is the potential associated with Di

Potential Method

- ***amortized cost*** of the i^{th} operation with respect to potential function is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

- The amortized cost of each operation is therefore its actual cost plus the increase in potential due to the operation. By the above equation the total amortized cost of the *n* operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .$$

Potential Method

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost. In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we guarantee, as in the accounting method, that we pay in advance. It is often convenient to define $\Phi(D_0)$ to be 0 and then to show that $\Phi(D_i) \geq 0$ for all i. (See Exercise 18.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i)$ - $\Phi(D_{i-}1)$ of the ith operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the actual cost of the operation is paid by the decrease in the potential.

Potential Method

- The amortized costs defined by the following equations depend on the choice of the potential function .

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned}$$

- Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs.
- There are often trade-offs that can be made in choosing a potential function; the best potential function to use depends on the desired time bounds

Dynamic Tables: Dynamically expanding and contracting a table

- In some applications, we do not know in advance how many objects will be stored in a table. We might allocate space for a table, only to find out later that it is not enough.
- The table must then be reallocated with a larger size, and all objects stored in the original table must be copied over into the new, larger table.
- Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size.
- . Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, we shall see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Dynamic Tables

- We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE.
- TABLE-INSERT inserts into the table an item that occupies a single ***slot***, that is, a space for one item.
- TABLE-DELETE can be thought of as removing an item from the table, thereby freeing a slot.
- The details of the data-structuring method used to organize the table are unimportant.
- We define the ***load factor*** (T) of a nonempty table T to be the number of items stored in the table divided by the size (number of slots) of the table.
- We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

Dynamic Tables

- Let us assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.
- When the table becomes full, i.e. load factor = 1, then the next operation triggers an expansion.
- We allocate a new area twice the size of the old and copy all items from the old table to the new table.
- If an operation does not trigger an expansion, its cost is 1. If it does trigger, its cost is num [T] + 1.

Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Use an aggregate method of analysis to determine the amortized cost per operation.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

Thanks for Your Attention!

Exercises

Exercise

```
Algorithm D($x, n$)
{
    $i := 1;$
    repeat
    {
        $x[i] := x[i] + 2;$ $i := i + 2;$
    } until $(i > n);$
    $i := 1;$
    while $(i \leq \lfloor n/2 \rfloor)$ do
    {
        $x[i] := x[i] + x[i+1];$ $i := i + 1;$
    }
}
```

Algorithm 1.23 Example algorithm

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.

   (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).

   (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.

   (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.

Exercise

- Let S be a set of n positive integers, where n is even. Give an efficient algorithm to partition S into two subsets $S1$ and $S2$ of n=2 elements each with the property that the difference between the sum of the elements in $S1$ and the sum of the elements in $S2$ is maximum. What is the time complexity of your algorithm?
- Let $A[1::n]$ be an array of integers, where $n > 2$. Give an $O(1)$ time algorithm to find an element in A that is neither the maximum nor the minimum.
- Consider the element uniqueness problem: Given a set of integers, determine whether two of them are equal. Give an *efficient* algorithm to solve this problem. Assume that the integers are stored in array $A[1::n]$. What is the time complexity of your algorithm?

Exercise

- If a MULTIPUSH operation were included in the set of stack operations, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?
- Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as (nk) time.
- A sequence of n operations is performed on a data structure. The i^{th} operation costs i if i is an exact power of 2,and 1 otherwise. Use an aggregate method of analysis to determine the amortized cost per operation.
- A sequence of stack operations is performed on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Exercise

- Consider an ordinary binary heap data structure with n elements that supports the instructions INSERT and EXTRACT-MIN in $O(\log n)$ worst-case time. Give a potential function such that the amortized cost of INSERT is $O(\log n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.
- What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?
- Suppose that a counter begins at a number with b 1's in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = (b)$. (Do not assume that b is constant.)

Next class : Asymptotic Notations ...

- f(n) = **O**(g(n)), There exist $c > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq f(n) \leq cg(n)}$ for each $n \geq n_0$

- f(n) = **Ω**(g(n)), There exist $c > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq cg(n) \leq f(n)}$ for each $n \geq n_0$

- f(n) = **Θ**(g(n)), There exist c_1, $c_2 > 0$ and $n_0 > 0$ such that: $\mathbf{0 \leq c_1g(n) \leq f(n) \leq c_2g(n)}$ for each $n \geq n_0$