Algorithm Design

Deepak Mitra
Asst. Prof., Soft. Dev. & Trainer
CDAC CoE Gaya
Email : d_mitra123@yahoo.com

Design & Analysis of Algorithms

- **Algorithm analysis**

  Analysis of resource usage of given algorithms (time , space)

- **Efficient algorithms**

  Algorithms that make an efficient usage of resources

- **Algorithm design**

  Methods for designing efficient algorithms

Design & Analysis of Algorithms

"algos" = Greek word for pain.

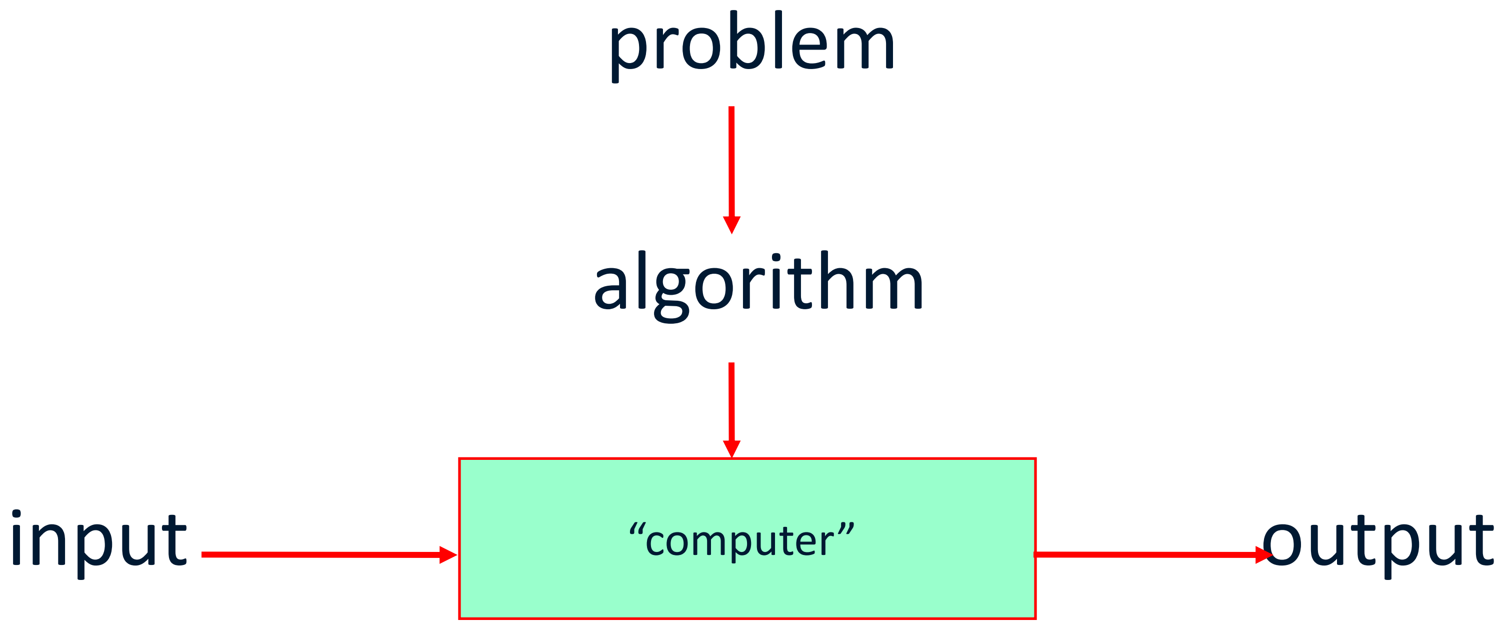
"algor" = Latin word for to be cold.

Why study this subject?

- Efficient algorithms lead to efficient programs.
- Efficient programs sell better.
- Efficient programs make better use of hardware.
- Programmers who write efficient programs are preferred.

What is an Algorithm?

An *algorithm* is a list of steps (sequence of unambiguous instructions) for solving a problem that transforms the input into the output.



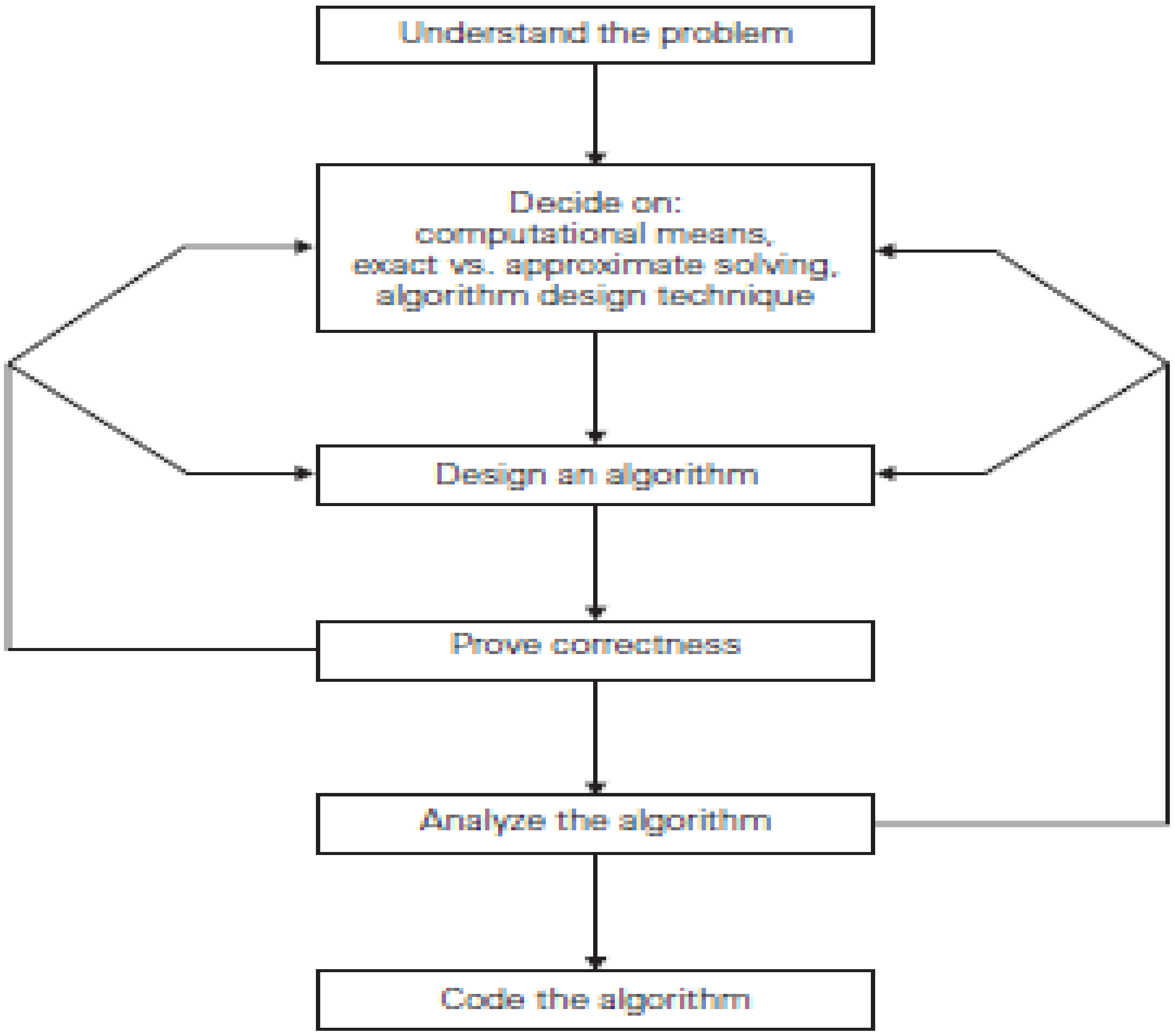
What is an Algorithm?

- An algorithm is “a finite set of precise instructions for performing a computation or for solving a problem”
  - A program is one type of algorithm
    - All programs are algorithms
    - Not all algorithms are programs!
  - Directions to somebody’s house is an algorithm
  - A recipe for cooking a cake is an algorithm
  - The steps to compute the cosine of 90° is an algorithm

Difference between Algorithm and Program

S.No	Algorithm	Program
1	Algorithm is finite	Program need not to be finite
2	Algorithm is written using natural language or algorithmic language	Programs are written using a specific programming language

Fundamentals of Algorithm and Problem Solving



Some Algorithms are harder than others

- Some algorithms are easy
  - Finding the largest (or smallest) value in a list
  - Finding a specific value in a list
- Some algorithms are a bit harder
  - Sorting a list
- Some algorithms are very hard
  - Finding the shortest path between Miami and Seattle
- Some algorithms are essentially impossible
  - Factoring large composite numbers

Properties of Algorithms

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Problem Solving Techniques

1. Understand the problem or Review the Specifications.
2. Plan the logic
3. a) (Informal Design)
   - i. List major tasks
   - ii. List subtasks, sub-subtasks & so on

   b) (Formal Design)
   - i. Create formal design from task lists
   - ii. Desk check design
4. Writing an algorithm 5. Flowcharting 6. Coding
7. Translate the program into machine language
8. Test the program
   - i. If necessary debug the program

10. Documentation
11. Put the program into production. If necessary maintain the program.

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

Some of these problems don't have efficient algorithms, or algorithms at all!

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency (or complexity) analysis
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm Design Strategies

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

PROPERTIES OF AN ALGORITHM

1. An algorithm takes zero or more inputs
2. An algorithm results in one or more outputs
3. All operations can be carried out in a finite amount of time
4. An algorithm should be efficient and flexible
5. It should use less memory space as much as possible
6. An algorithm must terminate after a finite number of steps.
7. Each step in the algorithm must be easily understood for some reading it
8. An algorithm should be concise and compact to facilitate verification of their correctness.

Properties of Algorithms

- Algorithms generally share a set of properties:
  - **Input**: what the algorithm takes in as input
  - **Output**: what the algorithm produces as output
  - **Definiteness**: the steps are defined precisely
  - **Correctness**: should produce the correct output
  - **Finiteness**: the steps required should be finite
  - **Effectiveness**: each step must be able to be performed in a finite amount of time
  - **Generality**: the algorithm *should* be applicable to all problems of a similar form

STEPS FOR WRITING AN ALGORITHM

- An algorithm consists of two parts.
  - The first part is a paragraph, which tells the purpose of the algorithm, which identifies the variables, occurs in the algorithm and the lists of input data.
  - The second part consists of the list of steps that is to be executed.

STEPS FOR WRITING AN ALGORITHM (Contd…)

Step 1: Identifying Number

Each algorithm is assigned an identifying number.
Example: Algorithm 1. Algorithm 2 etc.,

Step 2: Comment

Each step may contain comment brackets, which identifies or indicates the main purpose of the step.

The Comment will usually appear at the beginning or end of the step. It is usually indicated with two square brackets [ ].

- **Example :**

```
Step 1: [Initialize]
        set K : = 1
```

STEPS FOR WRITING AN ALGORITHM (Contd…)

Step 3 : Variable Names It uses capital letters. Example MAX, DATA. Single letter names of variables used as counters or subscripts.

Step 4 : Assignment Statement It uses the dot equal notation (: =). Some text uses $\rightarrow$ or $\leftarrow$ or = notations

Step 5 : Input and Output Data may be input and assigned to variables by means of a

Read statement. Its syntax is : READ : variable names

Example: READ: a, b, c

Similarly, messages placed in quotation marks, and data in variables may be output by means of a write or print statement. Its syntax is:

WRITE : Messages and / or variable names.

Example: Write: a,b,c

STEPS FOR WRITING AN ALGORITHM (Contd...)

Step 7 : Controls:

It has three types

(i) : Sequential Logic :

It is executed by means of numbered steps or by the order in which the modules are written

(ii) : Selection or Conditional Logic

It is used to select only one of several alternative modules. The end of structure is usually indicated by the statement.

[End - of-IF structure]

The selection logic consists of three types

Single Alternative, Double Alternative and Multiple Alternative

STEPS FOR WRITING AN ALGORITHM (Contd…)

a). Single Alternative : Its syntax is :

```
IF condition,  then :
        [ module a]
    [ End - of-IF structure]
```

b) Double alternative : Its syntax is

```
IF condition, then :
            [module A]
Else :
            [ module B]
[ End - of-IF structure]
```

c). Multiple Alternative : Its syntax is :

```
IF condition(1),  then :
    [module A1]
Else  IF condition (2), then:
            [Module A2]
Else IF condition (2) then.
            [module A2]
                 .............
      Else IF condition (M) then :
             [ Module  Am]
Else      [ Module  B]
[ End - of-IF structure]
```

STEPS FOR WRITING AN ALGORITHM (Contd…)

Iteration or Repetitive

It has two types. Each type begins with a repeat statement and is followed by the module, called body of the loop. The following statement indicates the end of structure.

```
                    [End of loop ]
```

(a) Repeat for loop

It uses an index variable to control the loop.

```
Repeat for K = R to S by T:
          [Module]
 [End of loop]
```

(b) Repeat while loop

It uses a condition to control the loop.

```
Repeat while condition:
          [Module]
 [End of loop]
```

iv) EXIT :

The algorithm is completed when the statement EXIT is encountered.

Important Problem Types

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinatorial Problems
- Geometric Problems
- Numerical Problems

Real-World Applications

- Hardware design: VLSI chips
- Compilers
- Computer graphics: movies, video games
- Routing messages in the Internet
- Searching the Web
- Distributed file sharing
- Computer aided design and manufacturing
- Security: e-commerce, voting machines
- Multimedia: CD player, DVD, MP3, JPG, HDTV
- DNA sequencing, protein folding
- and many more!

Some Important Problem Types

- Sorting
  - a set of items
- Searching
  - among a set of items
- String processing
  - text, bit strings, gene sequences
- Graphs
  - model objects and their relationships
- Combinatorial
  - find desired permutation, combination or subset
- Geometric
  - graphics, imaging, robotics
- Numerical
  - continuous math: solving equations, evaluating functions

Algorithm Design Techniques

- Brute Force & Exhaustive Search
  - follow definition / try all possibilities
- Divide & Conquer
  - break problem into distinct subproblems
- Transformation
  - convert problem to another one
- Dynamic Programming
  - break problem into overlapping subproblems
- Greedy
  - repeatedly do what is best now
- Iterative Improvement
  - repeatedly improve current solution
- Randomization
  - use random numbers

Searching

- Find a given value, called a search key, in a given set.
- Examples of searching algorithms
  - Sequential search
  - Binary search
  - Interpolation search
  - Robust interpolation search

String Processing

- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.

Examples:

searching for a word or phrase on WWW or in a Word document

searching for a short read in the reference genomic sequence

Graph Problems

- **Informal definition**
  - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- **Modeling real-life problems**
  - Modeling WWW
  - Communication networks
  - Project scheduling …
- **Examples of graph algorithms**
  - Graph traversal algorithms
  - Shortest-path algorithms
  - Topological sorting

Write an algorithm to add two numbers entered by the user.

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Algorithm Examples

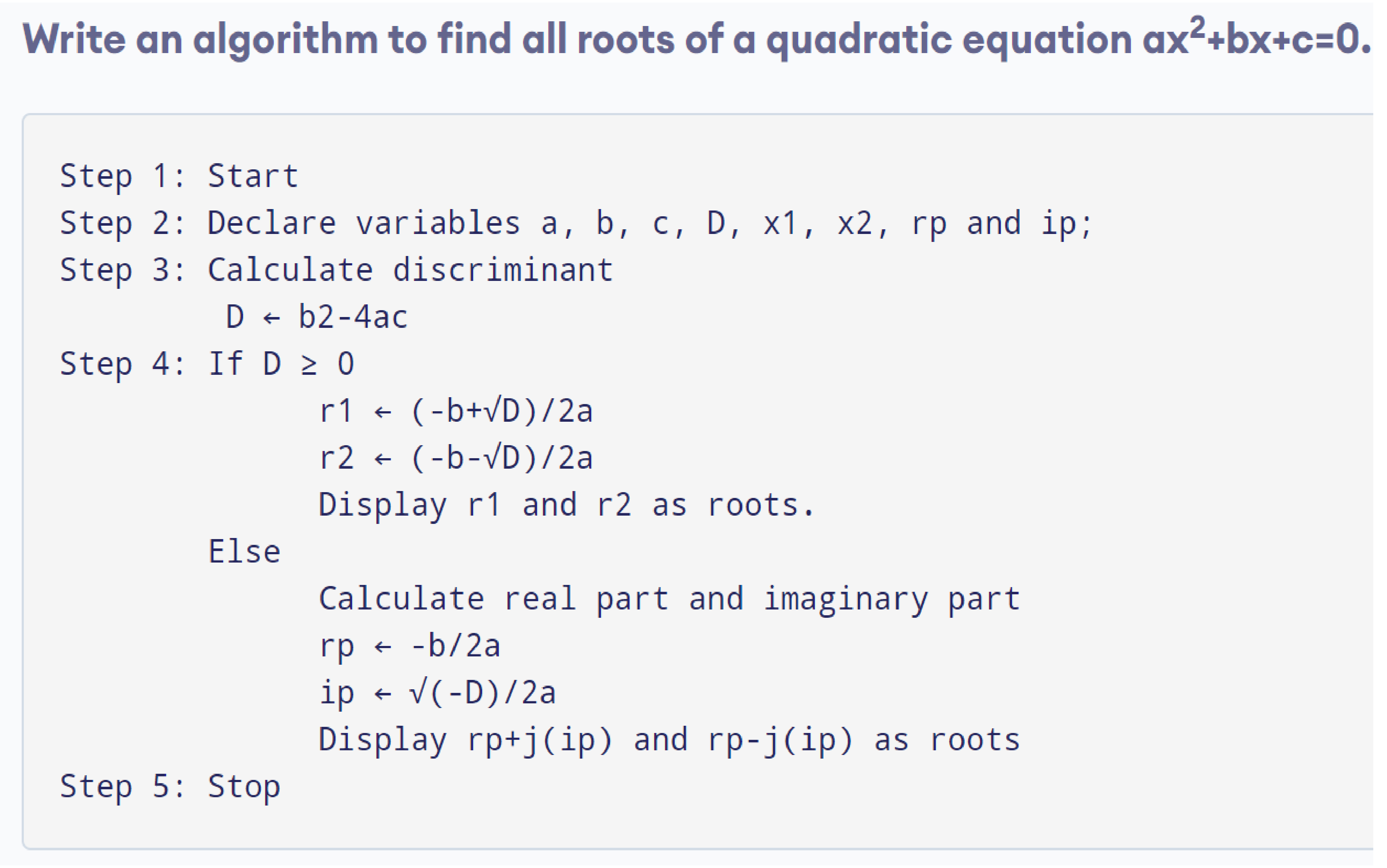
- We will use a pseudocode to specify algorithms, which slightly reminds us of Basic and Pascal.

- Example: an algorithm that finds the maximum element in a finite sequence

```
**procedure** max($a_1, a_2, \ldots, a_n$: integers)
max := $a_1$
**for** i := 2 **to** n
        **if** max < $a_i$ **then** max := $a_i$
{max is the largest element}
```

Write an algorithm to find the largest among three different numbers entered by the user.

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4:
 If a > b
        If a > c
                Display a is the largest number.
        Else
                Display c is the largest number.
Else
        If b > c
                Display b is the largest number.
        Else
                Display c is the greatest number.
Step 5: Stop
```



Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

```
Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
         D ← b2-4ac
Step 4: If D ≥ 0
              r1 ← (-b+√D)/2a
              r2 ← (-b-√D)/2a
              Display r1 and r2 as roots.
        Else
              Calculate real part and imaginary part
              rp ← -b/2a
              ip ← √(-D)/2a
              Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop
```

Write an algorithm to find the factorial of a number entered by the user.

```
Step 1: Start
Step 2: Declare variables n, factorial and i.
Step 3: Initialize variables
          factorial ← 1
          i ← 1
Step 4: Read value of n
Step 5: Repeat the steps until i = n
     5.1: factorial ← factorial*i
     5.2: i ← i+1
Step 6: Display factorial
Step 7: Stop
```

Write an algorithm to check whether a number entered by the user is prime or not.

```
Step 1: Start
Step 2: Declare variables n, i, flag.
Step 3: Initialize variables
        flag ← 1
        i ← 2
Step 4: Read n from the user.
Step 5: Repeat the steps until i=(n/2)
     5.1 If remainder of n÷i equals 0
            flag ← 0
            Go to step 6
     5.2 i ← i+1
Step 6: If flag = 0
           Display n is not prime
        else
           Display n is prime
Step 7: Stop
```

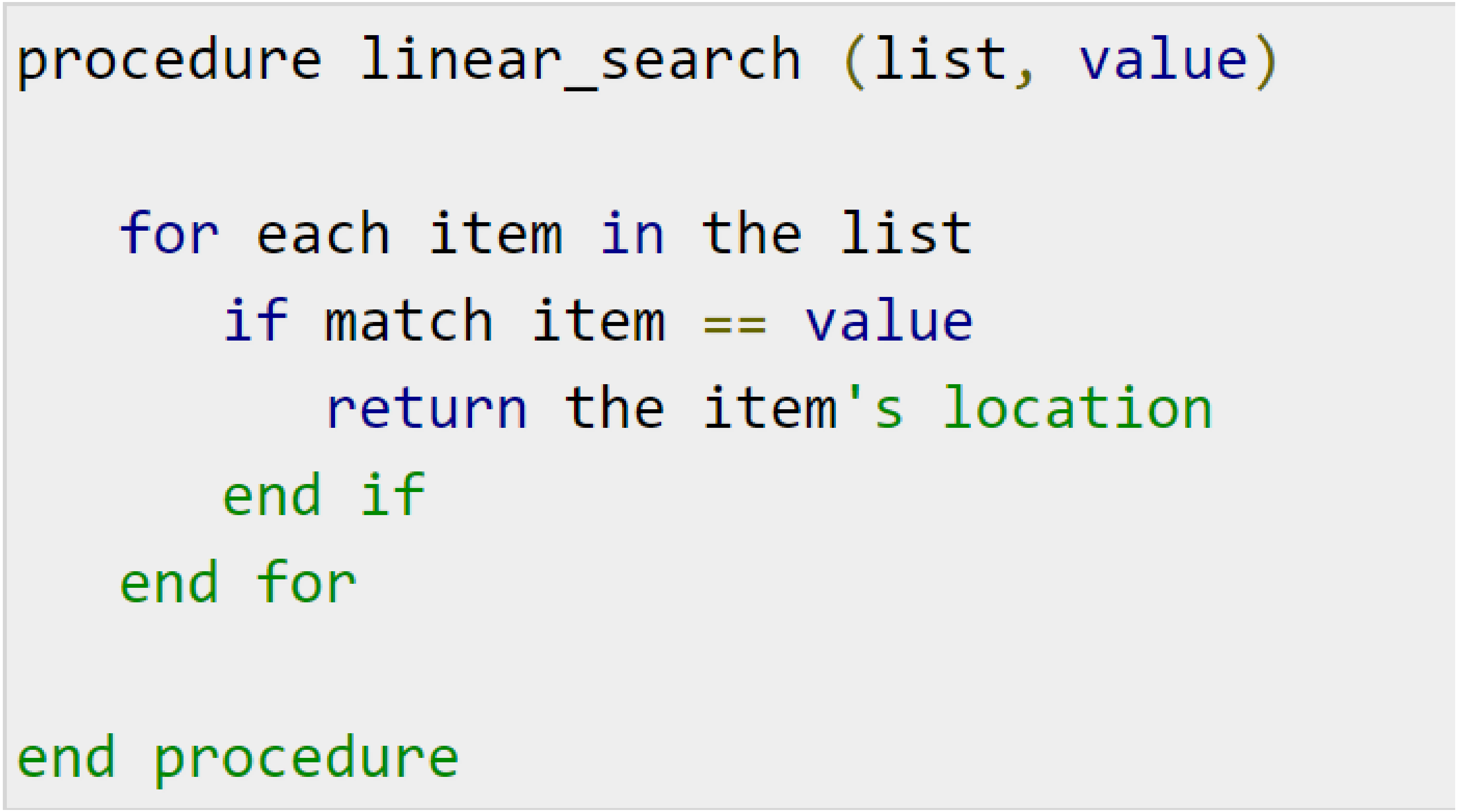
Write an algorithm to find the Fibonacci series till term≤1000.

```
Step 1: Start
Step 2: Declare variables first_term,second_term and temp.
Step 3: Initialize variables first_term ← 0 second_term ← 1
Step 4: Display first_term and second_term
Step 5: Repeat the steps until second_term ≤ 1000
     5.1: temp ← second_term
     5.2: second_term ← second_term + first_term
     5.3: first_term ← temp
     5.4: Display second_term
Step 6: Stop
```

Algorithm for Linear Search

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

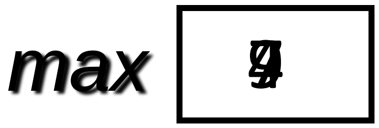
Pseudocode for Linear Search

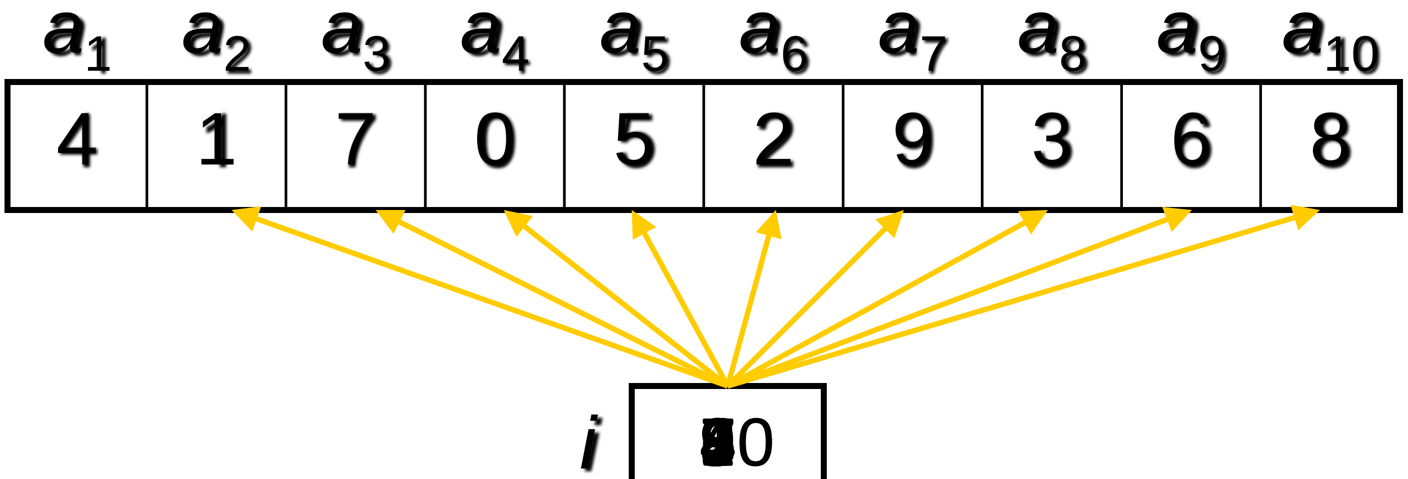


```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
   end for

end procedure
```

Algorithm 1: Maximum element

```
**procedure** max ($a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i$ := 2 to n
    if max < $a_i$ then $max := a_i$
```




a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
4 1 7 0 5 2 9 3 6 8
i
10

Algorithm 1: Maximum element

- Algorithm for finding the maximum element in a list:

```
**procedure** max ($a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
**for** i := 2 **to** n
    **if** max < $a_i$ **then** $max := a_i$

{$max$ is the largest element}
```

Analysis of Algorithms

- In theoretical analysis of algorithms, it is common **to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.** The term **"analysis of algorithms" was coined by Donald Knuth**.
- Algorithm analysis is an important **part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.** Most algorithms are designed to work with **inputs of arbitrary length**. Analysis of algorithms is the **determination of the amount of time** and **space resources required to execute** it.
- Usually, the **efficiency or running time of an algorithm is stated as a function** relating the **input length to the number of steps**, known as **time complexity**, or volume of memory, known as **space complexity**.

The Need for Analysis

- The **need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.**
- By considering an **algorithm for a specific problem**, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

- **Algorithms are often quite different from one another**, though the **objective of these algorithms are the same**. For example, we know that a set of numbers can be sorted using different algorithms. **Number of comparisons** performed by one algorithm may vary with others for the same input. Hence, **time complexity** of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

- To solve a problem, we need to consider **time** as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Analysis of algorithm is the **process of analyzing the problem-solving capability of the algorithm in terms of the time and size required** (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the **required time** or **performance**. Generally, we perform the following types of analysis

Analysis of Algorithms

- **Issues**:
  - correctness
  - time efficiency
  - space efficiency
  - optimality
- **Approaches**:
  - theoretical analysis
  - empirical analysis

Theoretical Analysis of Time Efficiency

Time efficiency is analyzed by determining the **number of repetitions** of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes most towards the running time of the algorithm

input size

$$T(n) \approx c_{op}C(n)$$

running time

execution time for basic operation

Number of times basic operation is executed

Input Size and Basic Operation Examples

Problem	Input size measure	Basic operation
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Empirical analysis of time efficiency

- Select a specific (typical) **sample of inputs**

- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions

- Analyze the empirical data

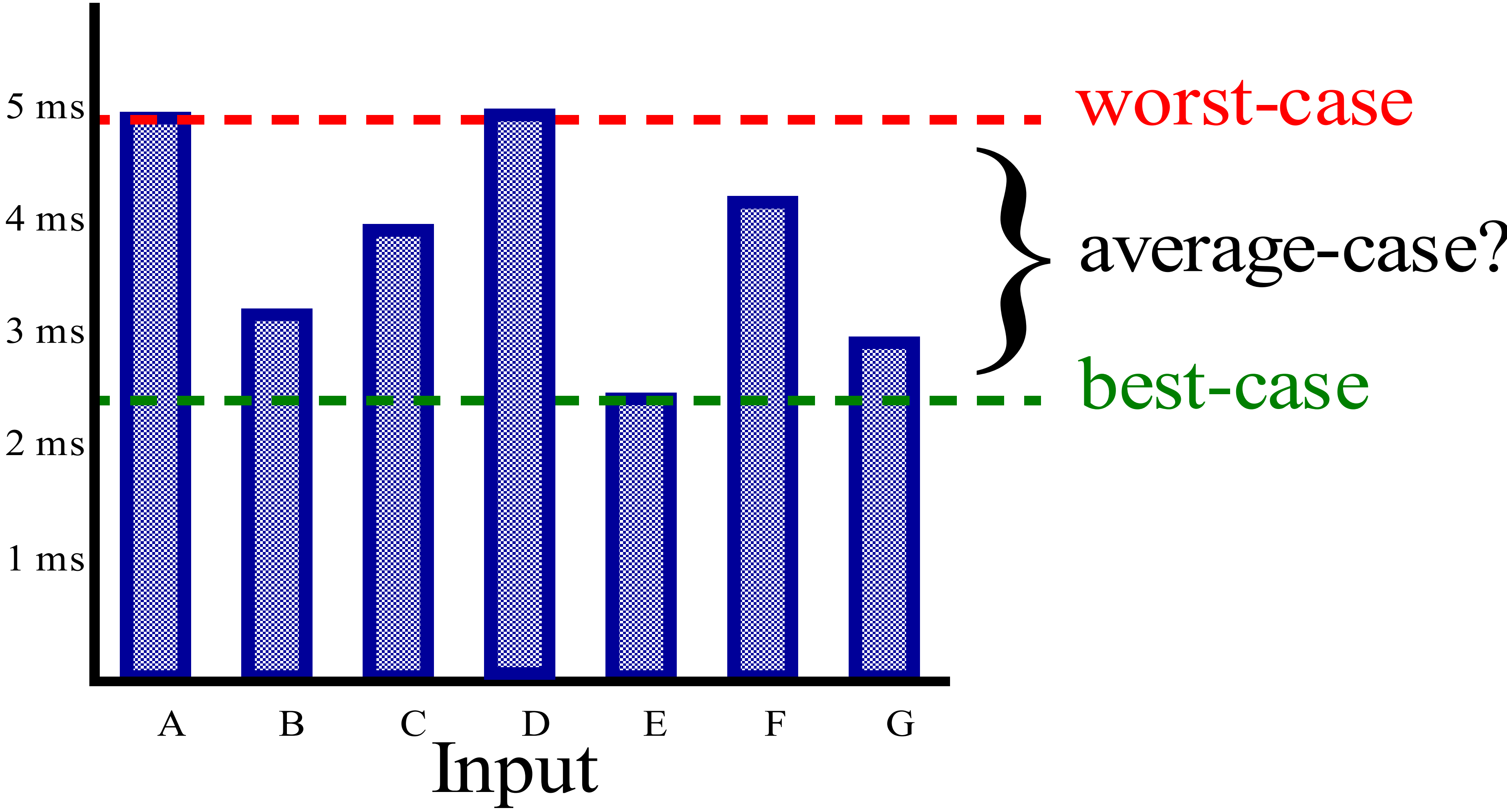
Analysis of algorithm is the process of analyzing the **problem-solving capability of the algorithm in terms of the time and size required** (the size of memory for storage while implementation). However, the **main concern of analysis of algorithms is the required time or performance**. Generally, we perform the following types of analysis

Best-Case, Average-Case, Worst-Case

For some algorithms efficiency depends on form of input:

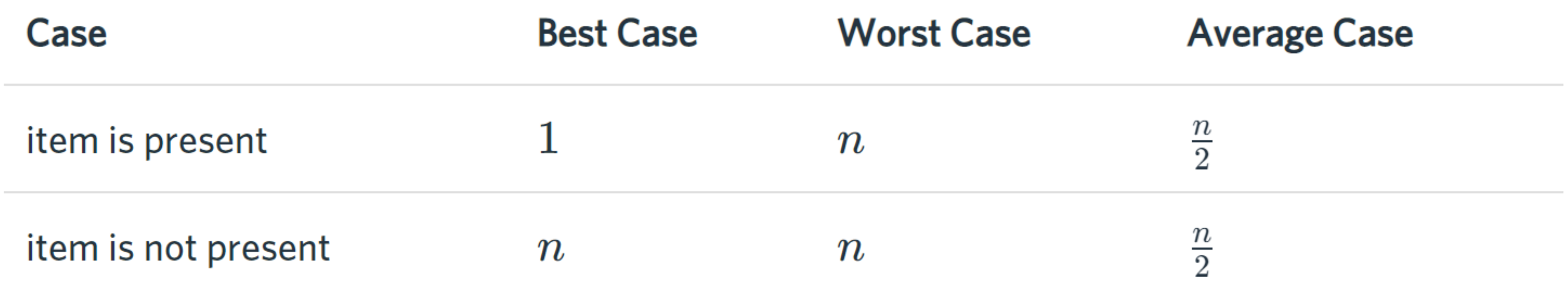
- Worst case: $C_{worst}(n)$ – maximum over inputs of size *n*
- Best case: $C_{best}(n)$ – minimum over inputs of size *n*
- Average case: $C_{avg}(n)$ – "average" over inputs of size *n*
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

- **Worst-case** – The **maximum number of steps taken** on any instance of size **a**.
- **Best-case** – The **minimum number of steps** taken on any instance of size **a**.
- **Average case** – An **average number of steps** taken on any instance of size **a**.
- **Amortized** – A sequence of operations applied to the input of size **a** averaged over time.


5 ms
4 ms
3 ms
2 ms
1 ms
A
B
C
D
E
F
G
Input
worst-case
average-case?
best-case

Assume that the list of items was constructed so that the items were in **ascending order, from low to high**. If the item we are **looking for is present in the list**, the chance of it being in any one of the *n* positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. The diagram below shows this process as the algorithm looks for the item 50. Notice that items are still compared in sequence until 54. At this point, however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted.

In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately.



Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	$\frac{n}{2}$

Math You Need to Review

Logarithms and Exponents

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\mathrm{Log}_b a = \log_x a/\log_x b$

- **properties of exponentials:**

  $a^{(b+c)} = a^b a^{c}$

  $a^{bc} = (a^b)^c$

  $a^b /a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

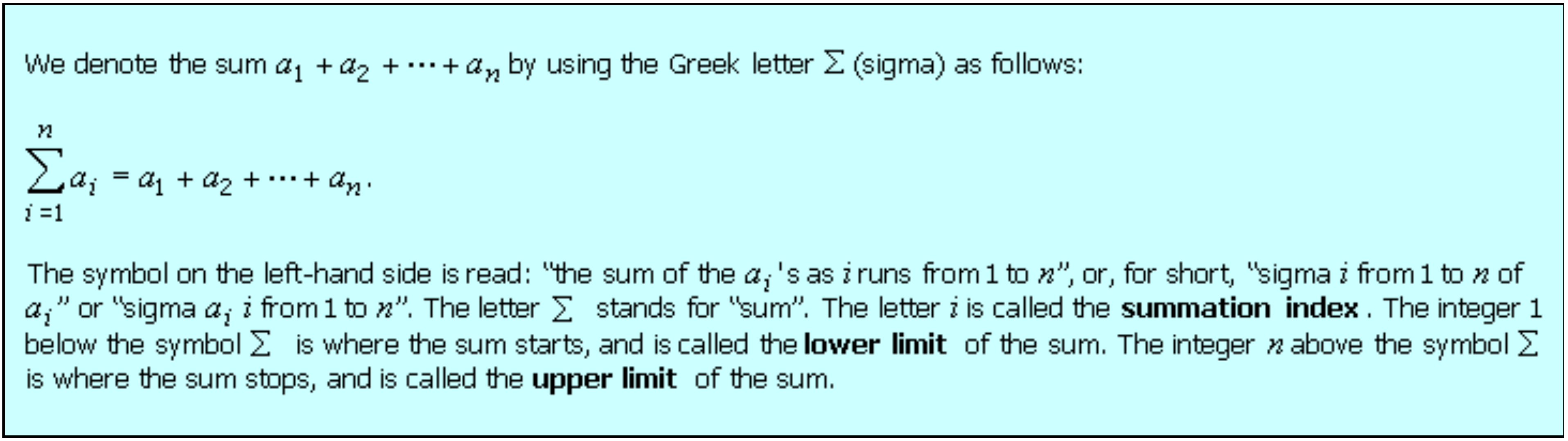
- Floor: $\lfloor x \rfloor$ = the largest integer $\leq x$
- Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$

Summation formulas and properties

Given a sequence $a_1, a_2, \ldots$ of numbers, the finite sum $a_1 + a_2 + \cdots + a_n$, where n is an nonnegative integer, can be written

$$\sum_{k=1}^{n} a_k .$$

If $n = 0$, the value of the summation is defined to be 0. The value of a finite series is always well defined, and its terms can be added in any order.



We denote the sum $a_1 + a_2 + \cdots + a_n$ by using the Greek letter Σ (sigma) as follows:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n .$$

The symbol on the left-hand side is read: "the sum of the a_i's as i runs from 1 to n", or, for short, "sigma i from 1 to n of a_i" or "sigma a_i i from 1 to n". The letter $\sum$ stands for "sum". The letter i is called the **summation index**. The integer 1 below the symbol $\sum$ is where the sum starts, and is called the **lower limit** of the sum. The integer n above the symbol $\sum$ is where the sum stops, and is called the **upper limit** of the sum.

Write out these sums:

$$\sum_{i=1}^{5} f(x_i)\Delta x_i, \quad \sum_{j=1}^{6} j^2, \quad \sum_{k=0}^{n} a_k x^k.$$

Solution

$$\sum_{i=1}^{5} f(x_i)\Delta x_i = f(x_1)\Delta x_1 + f(x_2)\Delta x_2 + f(x_3)\Delta x_3 + f(x_4)\Delta x_4 + f(x_5)\Delta x_5,$$

$$\sum_{j=1}^{6} j^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2,$$

$$\sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \quad (\text{note: } x^0 = 1 \text{ and } x^1 = x).$$

Properties

i. $$\sum_{i=1}^{n}(a_i + b_1) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i .$$

ii. $$\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i .$$

iii. $$\sum_{i=1}^{n}(a_i + b_i)^2 = \sum_{i=1}^{n} a_i^2 + 2\sum_{i=1}^{n} a_i b_i + \sum_{i=1}^{n} b_i^2 .$$

Proof

i. $$\sum_{i=1}^{n}(a_i + b_i) = (a_1 + b_1) + (a_2 + b_2) + \cdots + (a_n + b_n)$$
$$= (a_1 + a_2 + \cdots + a_n) + (b_1 + b_2 + \cdots + b_n)$$
$$= \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i .$$

ii and **iii**. Similar to **i**.

Formulas

i. $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

ii. $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6},$$

iii. $$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4},$$

iv. $$\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}, \quad r \neq 1 \quad \text{(geometric sum).}$$

Arithmetic Series: For $n \geq 0$,

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2).$$

Geometric Series: Let $x \neq 1$ be any constant (independent of i), then for $n \geq 0$,

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}.$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

Harmonic Series: This arises often in probabilistic analyses of algorithms. For $n \geq 0$,

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n = \Theta(\ln n).$$

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

- **properties of exponentials:**

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

Types of formulas for basic operation's count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

½(n*(n-1)
0.5(n*(n-1)
0.5*(n2 =n)
0.5n2-0.5n

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

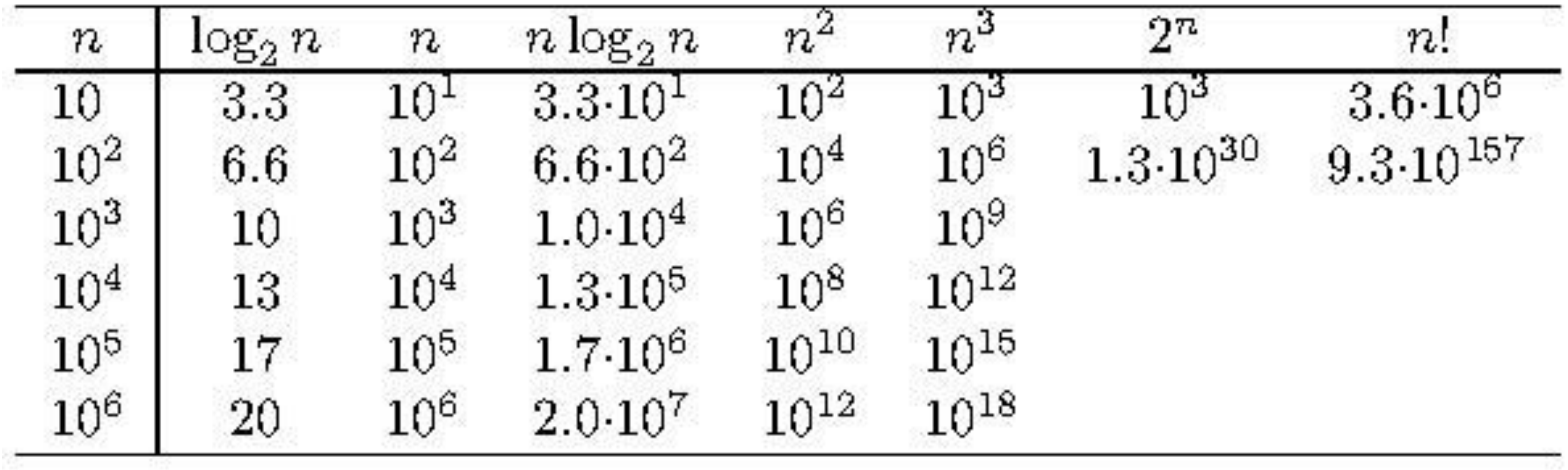
- Formula indicating order of growth with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

Order of growth

- Most important: Order of growth within a constant multiple as $\boldsymbol{n \to \infty}$
- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Values of some important functions as *n* → ∞



n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic Analysis

adjective Mathematics.

•of or relating to an asymptote.

•(of a function) **approaching a given value as an expression containing a variable tends to infinity.**

•(of two functions) so **defined that their ratio approaches** unity as the independent variable approaches a limit or infinity.

•(of a formula) becoming **increasingly exact as a variable approaches a limit, usually infinity**.

•coming into consideration as a **variable approaches a limit, usually infinity:** ***asymptotic property; asymptotic behavior.***

Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. **To study Function growth efficiently**, we reduce the function down to the important part.

Let $f(n) = an^2+bn+c$

In this function, the **n^2 term dominates the function that is when n gets sufficiently large.**

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

Asymptotic notation:

The word **Asymptotic means approaching a value or curve arbitrarily closely** (i.e., as some sort of limit is taken).

Asymptotic Analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to **very large input datasets**

The simplest example is a function $f(n) = n^2+3n$, the term **3n becomes insignificant compared to n^2** when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to **write fastest and slowest possible running time for an algorithm.** These are also referred to as **'best case' and 'worst case'** scenarios respectively.

"In asymptotic notations, **we derive the complexity concerning the size of the input.** (Example in terms of n)"

"These notations are important because **without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms.**"

Why is Asymptotic Notation Important?

1. They give simple characteristics of an **algorithm's efficiency**.

2. They allow the **comparisons of the performances of various algorithms**.

Asymptotic Notations:

Asymptotic Notation is a **way of comparing function that ignores constant factors and small input sizes.** Three notations are used to calculate the running time complexity of an algorithm:

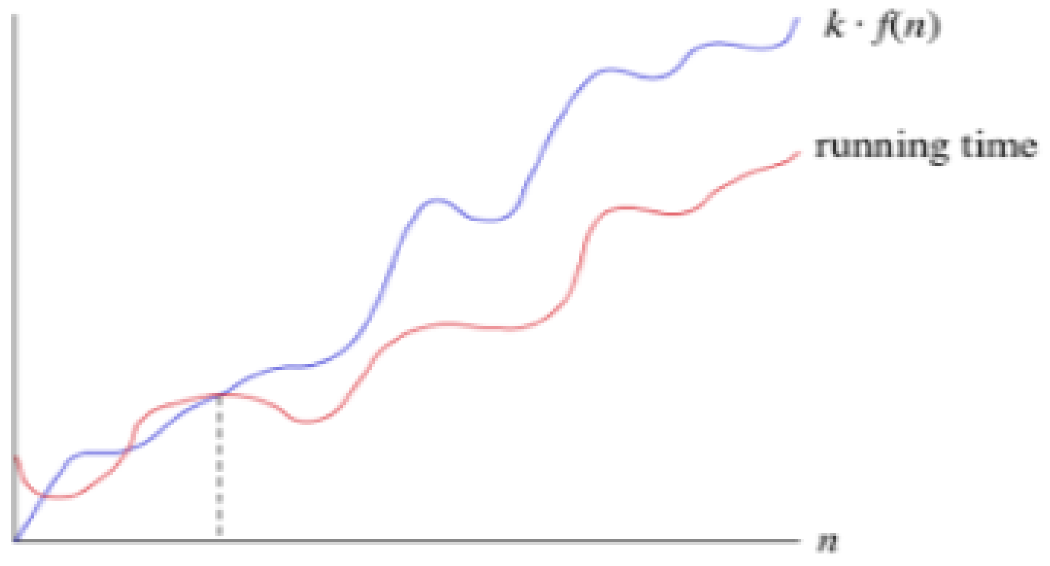
1. Big-oh notation: Big-oh is the formal method of expressing the **upper bound** of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant k and such that

$$\mathbf{f\ (n) \leqslant k^* g\ (n)\ for\ n \geq n_0\ in\ all\ case}$$

The values of k and $\mathbf{n_0}$ must be fixed for the function f and must not depend on n.

Hence, **function g (n) is an upper bound for function f (n),** as g (n) grows faster than f (n)

The Big O notation **defines an upper bound of an algorithm**, it bounds a function only from above.



ASYMPTOTIC UPPER BOUND

For Example:

1. 3n+2=O(n) as 3n+2≤4n **for** all n≥2

2. 3n+3=O(n) as 3n+3≤4n **for** all n≥3

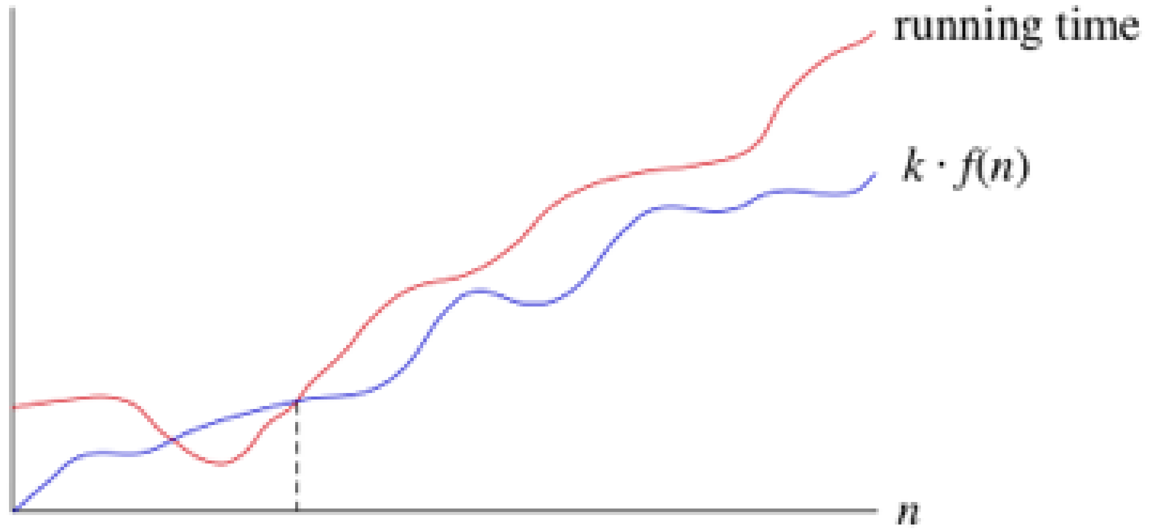
Hence, the complexity of **f(n)** can be represented as O (g (n))

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Omega () Notation: The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant k and n_0 such that

f (n) ≥ k* g (n) for all n, n≥ n0



For Example:

f (n) $=8n^2+2n-3 \geq 8n^2-3$

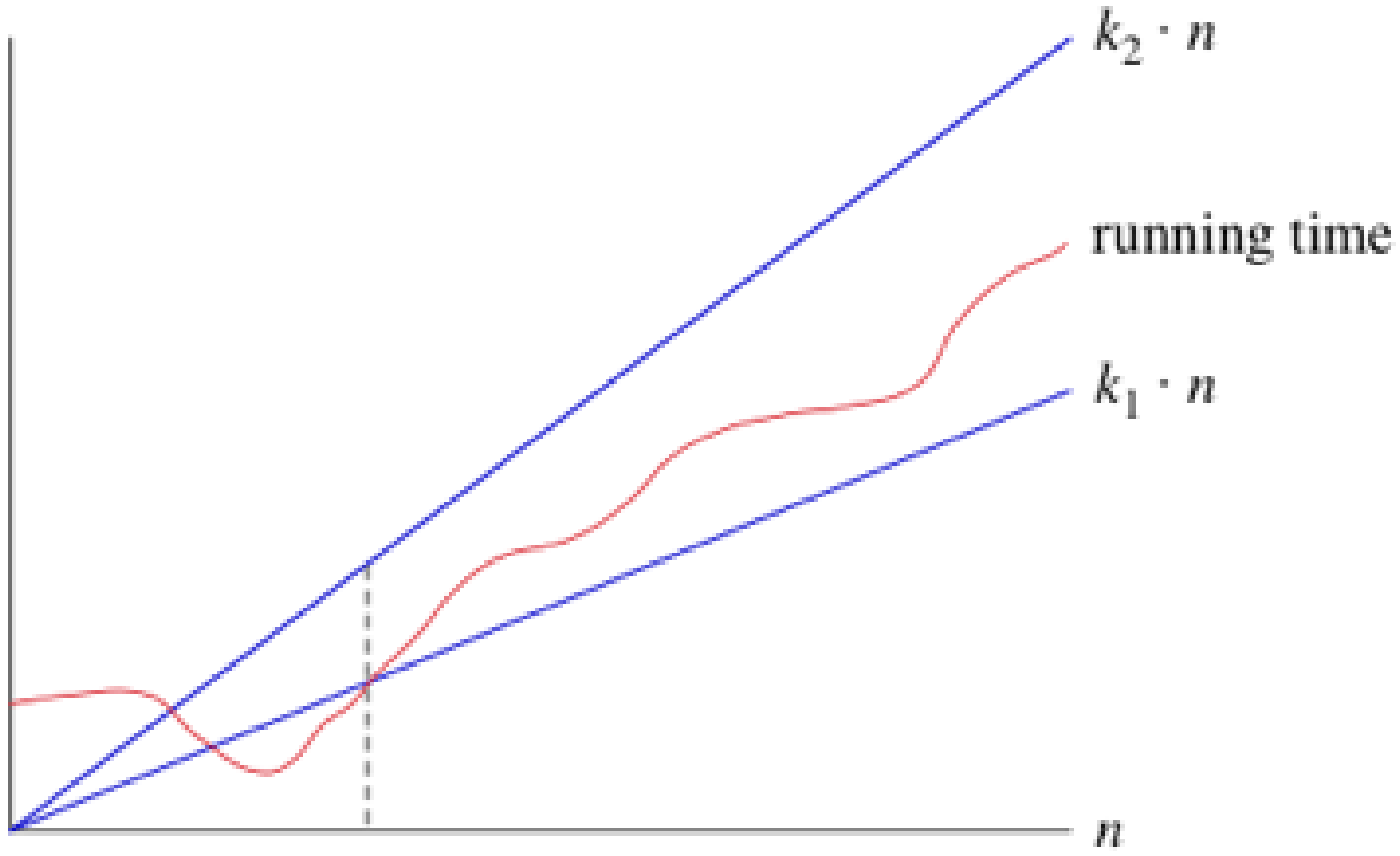
$=7n^2+(n^2-3) \geq 7n^2$ (g(n))

Thus, $k_1=7$

Hence, the complexity of f (n) can be represented as Ω (g (n))

Theta (θ): The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2\, g(n)$ for all n, $n \geq n0$



ASYMPTOTIC TIGHT BOUND

For Example:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n k1=3,k2=4, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- O($g(n)$): class of functions $f(n)$ that grow **no faster** than $g(n)$
- Θ($g(n)$): class of functions $f(n)$ that grow **at same rate** as $g(n)$
- Ω($g(n)$): class of functions $f(n)$ that grow **at least as fast** as $g(n)$

Big-oh

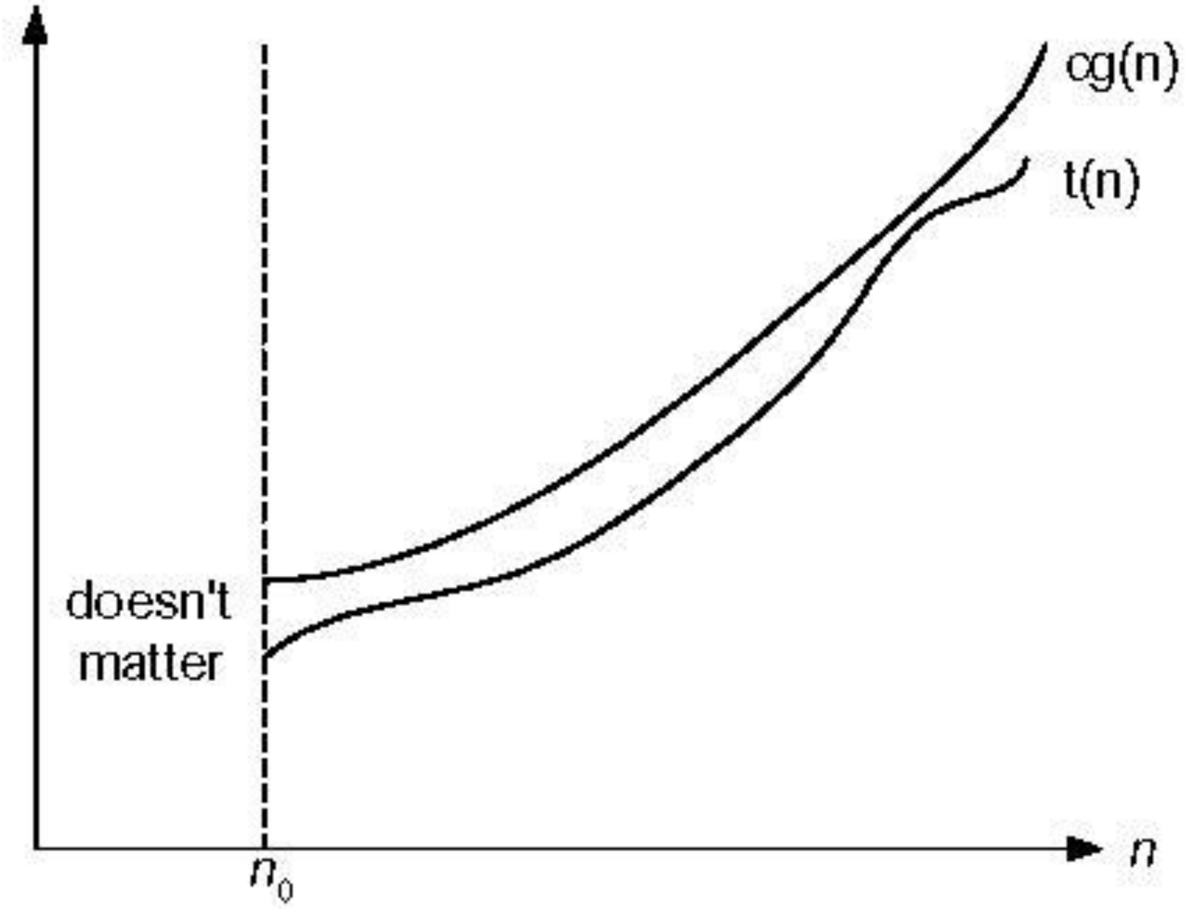


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

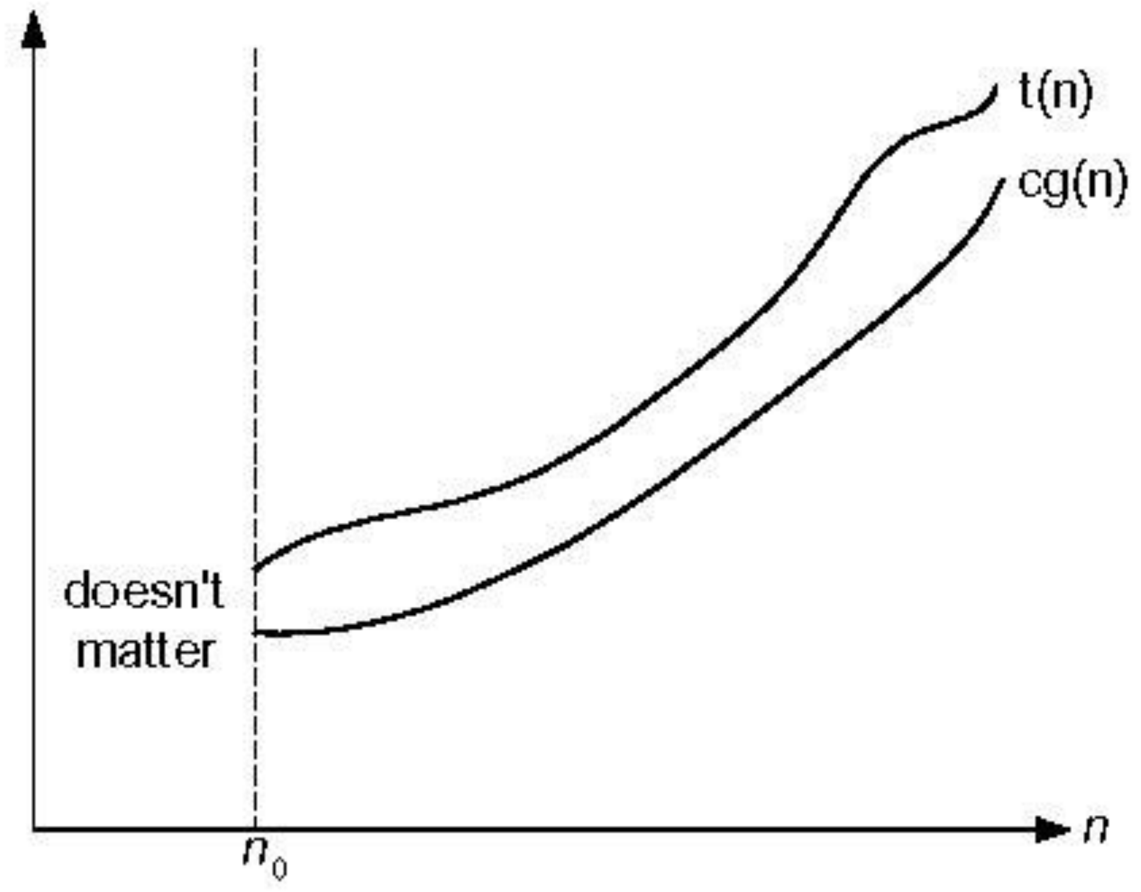


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

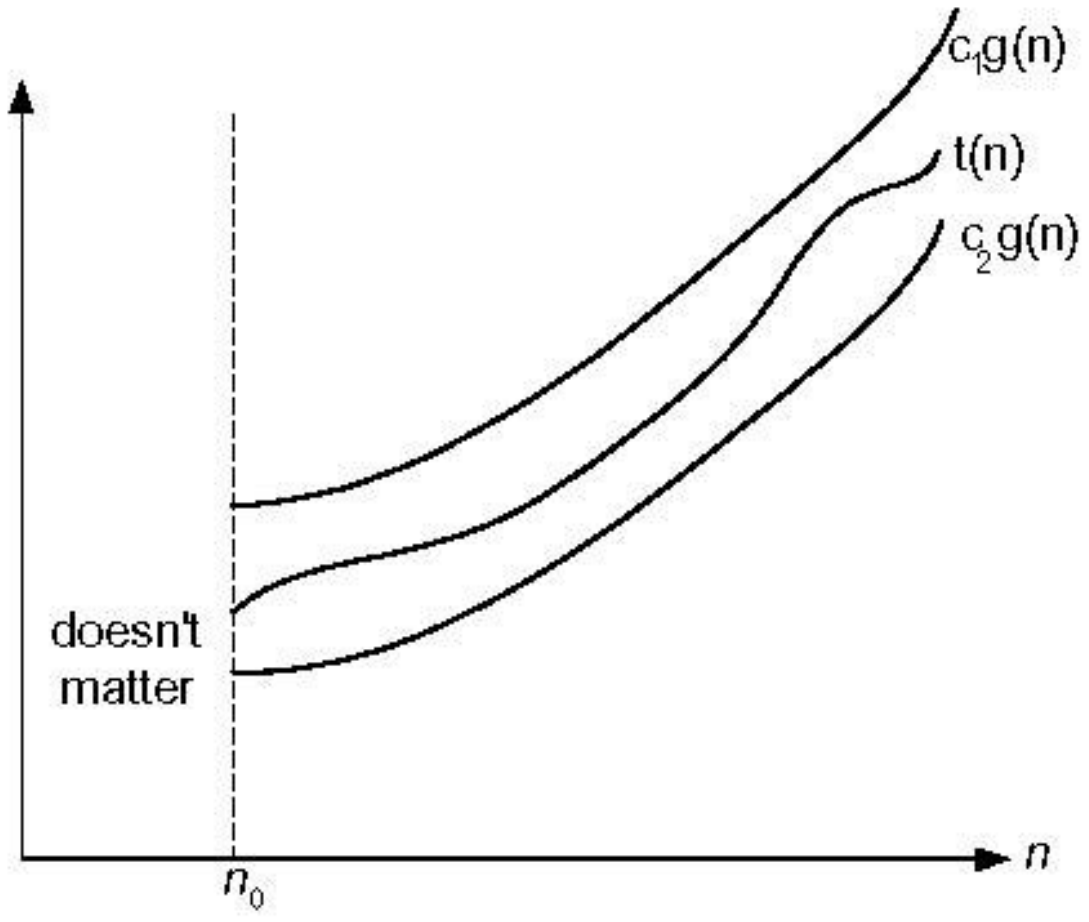
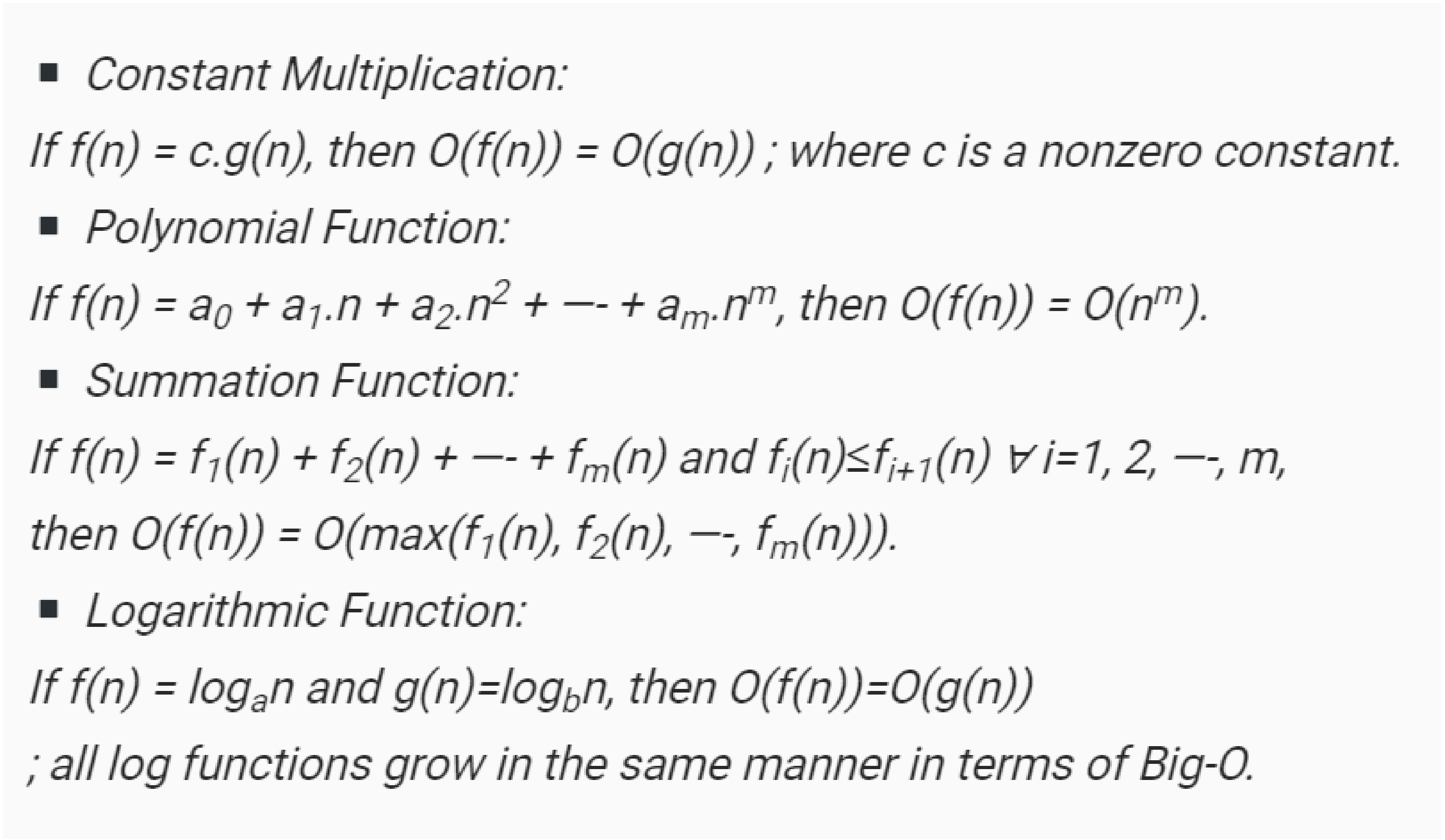


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Some of the useful properties on Big-O notation analysis are as follow:



- *Constant Multiplication:*

If $f(n) = c.g(n)$, then $O(f(n)) = O(g(n))$; where c is a nonzero constant.

- *Polynomial Function:*

If $f(n) = a_0 + a_1.n + a_2.n^2 + —- + a_m.n^m$, then $O(f(n)) = O(n^m)$.

- *Summation Function:*

If $f(n) = f_1(n) + f_2(n) + —- + f_m(n)$ and $f_i(n) \leq f_{i+1}(n)$ $\forall$ $i=1, 2, —-, m$,

then $O(f(n)) = O(max(f_1(n), f_2(n), —-, f_m(n)))$.

- *Logarithmic Function:*

If $f(n) = log_a n$ and $g(n)=log_b n$, then $O(f(n))=O(g(n))$

; all log functions grow in the same manner in terms of Big-O.

Basically, this asymptotic notation is used to measure and compare the worst-case scenarios of algorithms theoretically. For any algorithm, the Big-O analysis should be straightforward as long as we correctly identify the operations that are dependent on n, the input size.

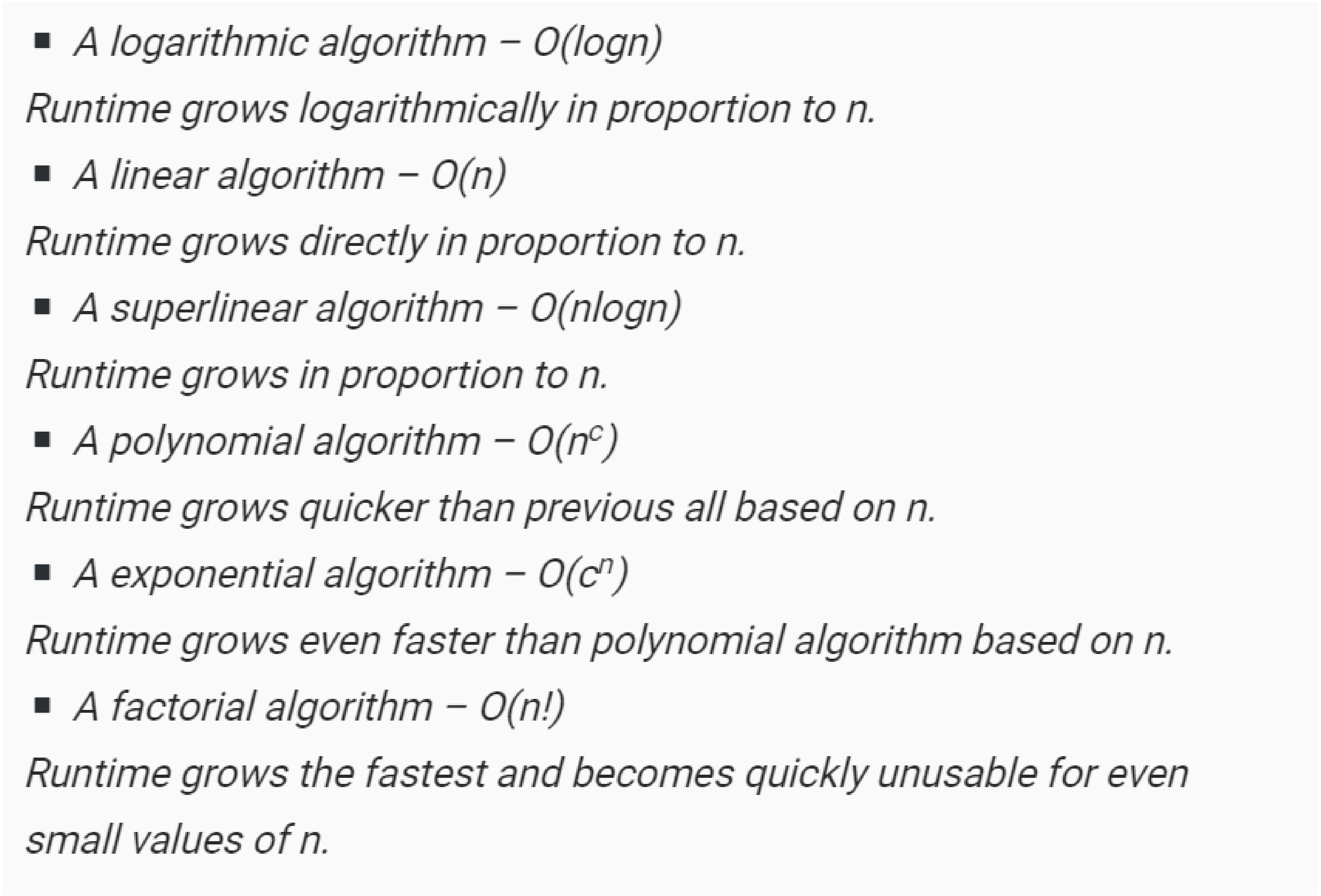
Runtime Analysis of Algorithms

In general cases, we **mainly used to measure and compare the worst-case theoretical running time complexities of algorithms for the performance analysis.**

The fastest possible running time for any algorithm is O(1), commonly referred to as **Constant Running Time**. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it’s rarely achievable.

In actual cases, **the performance (Runtime) of an algorithm depends on n**, **that is the size of the input or the number of operations is required for each input item**.

The algorithms can be classified as follows from the best-to-worst performance (Running Time Complexity):



- *A logarithmic algorithm – $O(logn)$*

Runtime grows logarithmically in proportion to n.

- *A linear algorithm – $O(n)$*

Runtime grows directly in proportion to n.

- *A superlinear algorithm – $O(nlogn)$*

Runtime grows in proportion to n.

- *A polynomial algorithm – $O(n^c)$*

Runtime grows quicker than previous all based on n.

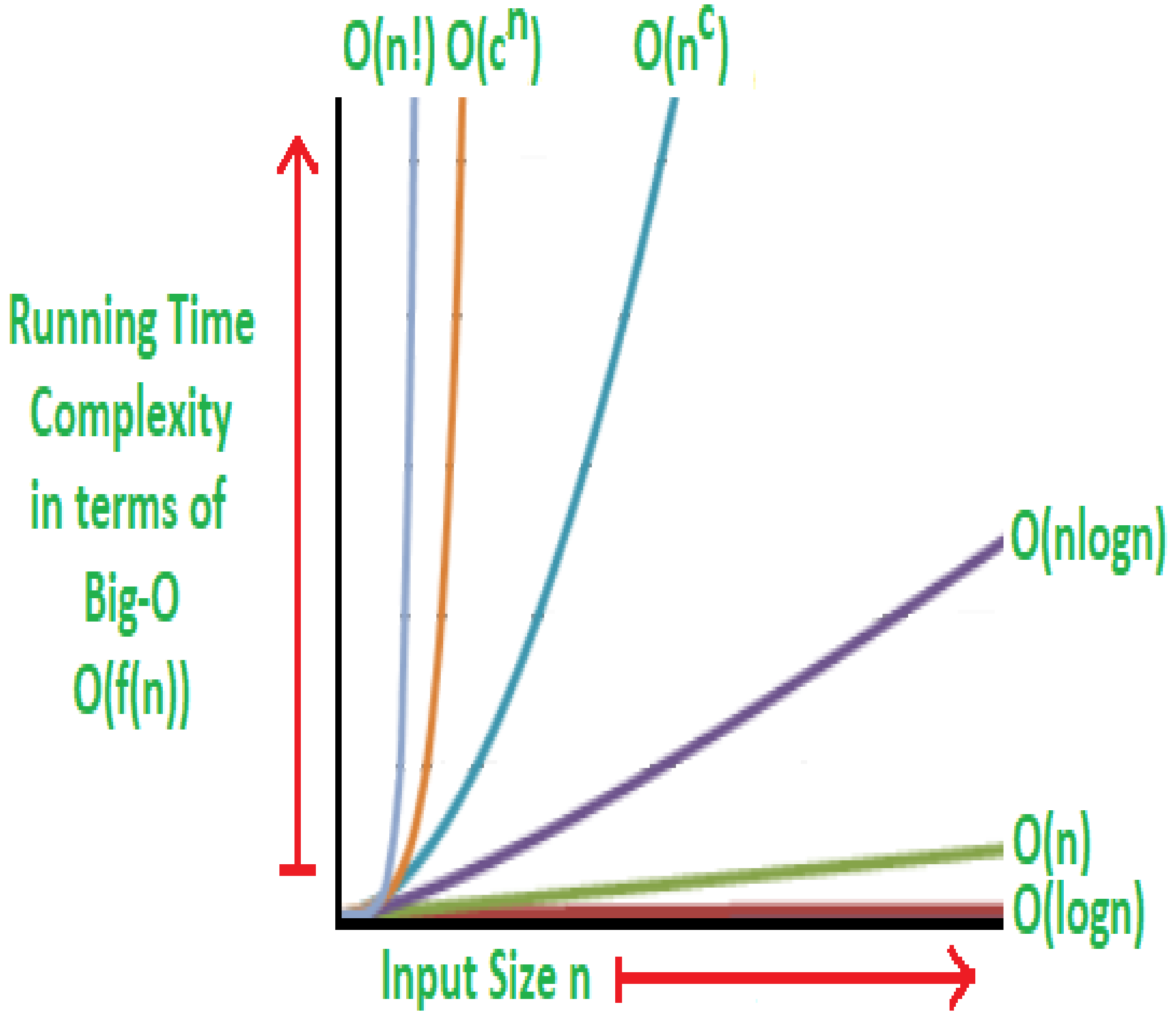
- *A exponential algorithm – $O(c^n)$*

Runtime grows even faster than polynomial algorithm based on n.

- *A factorial algorithm – $O(n!)$*

Runtime grows the fastest and becomes quickly unusable for even small values of n.

Where, n is the input size and c is a positive constant.



O(n!),O(c^n),O(n^c) - Worst

O(nlogn) - Bad

O(n) - Fair

O(logn) - Good

O(1) - Best

Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
n log n	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

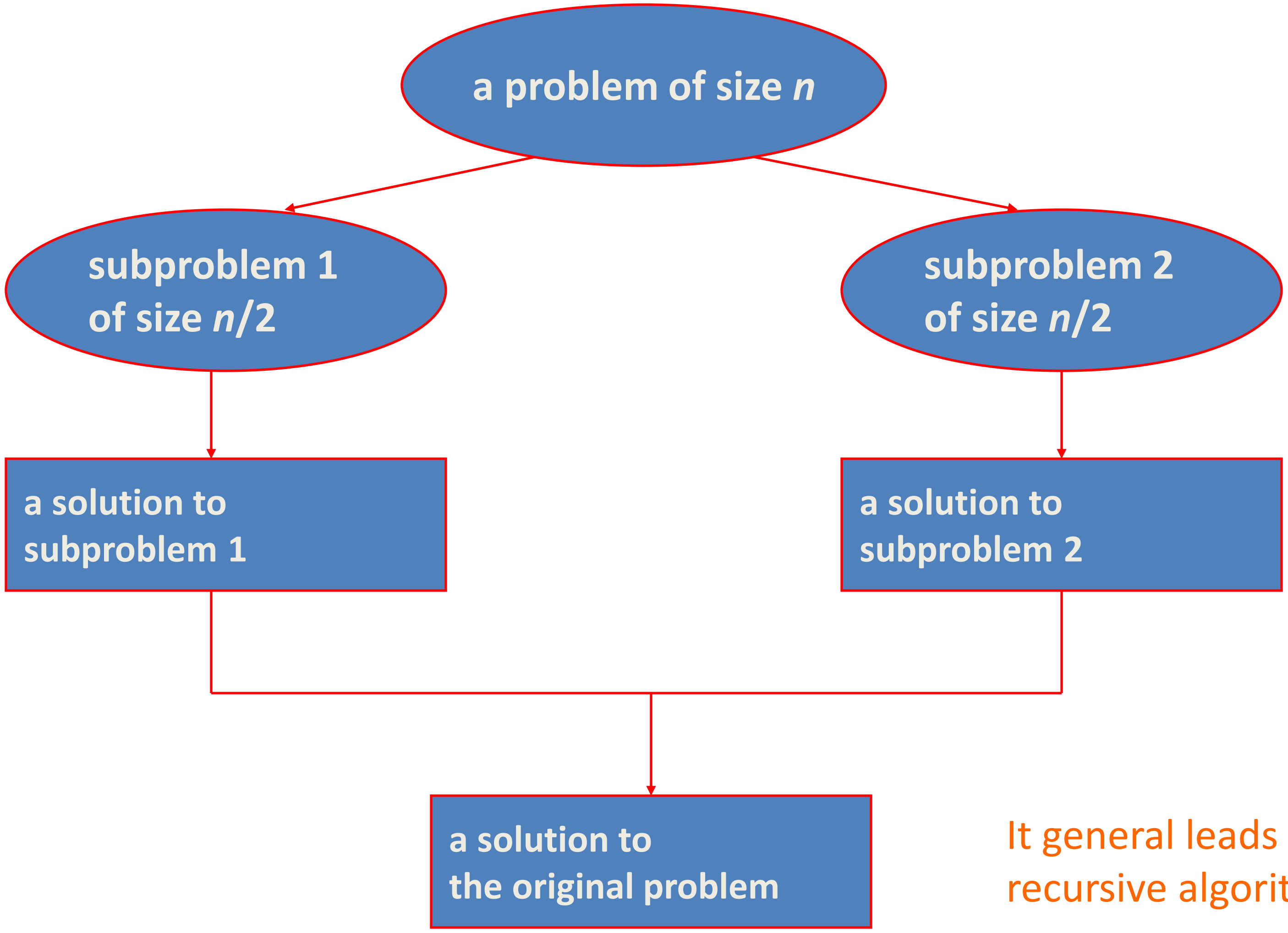
Divide and Conquer

A **divide and conquer algorithm** is a strategy of solving a large problem by breaking the problem into smaller sub-problems solving the sub-problems, and

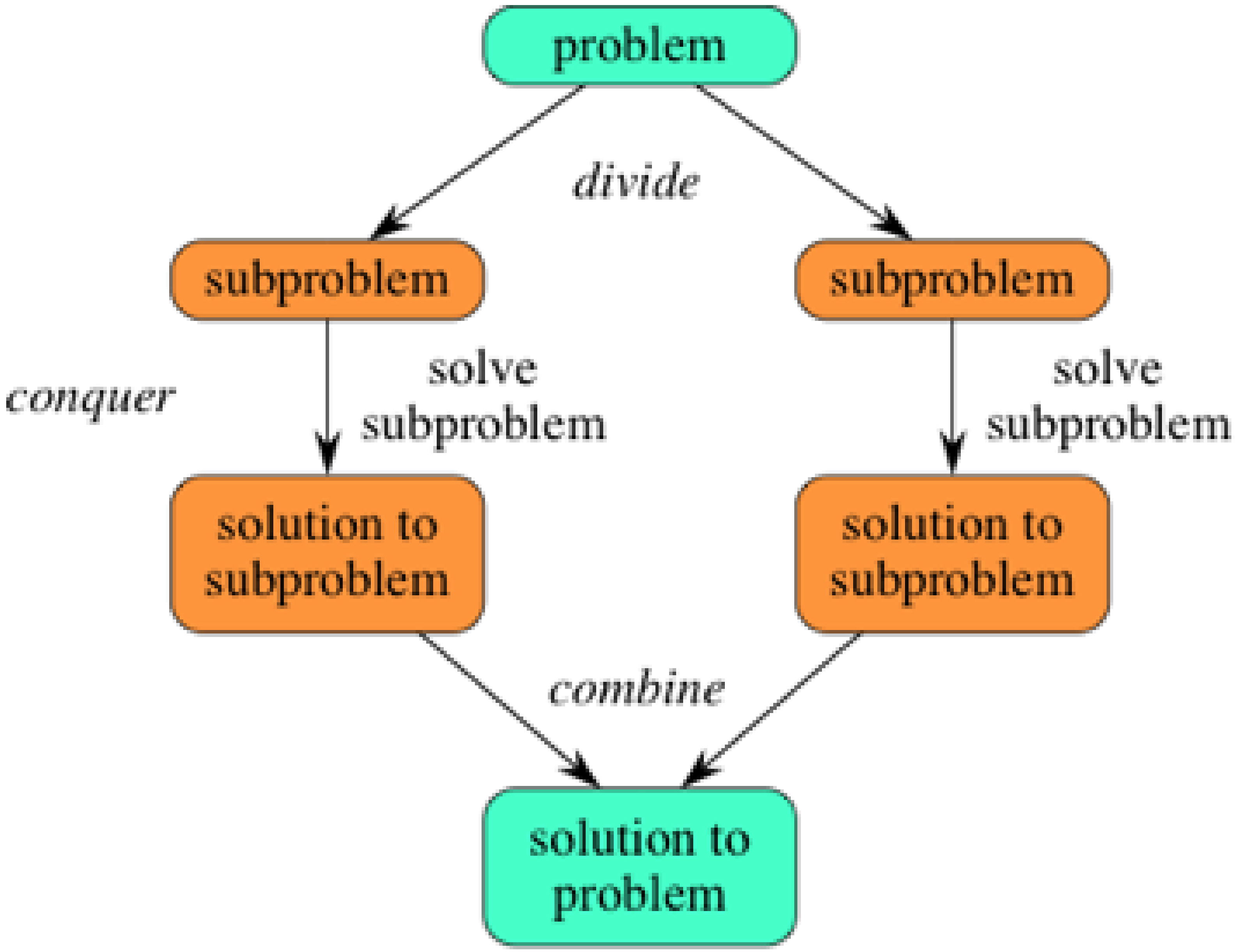
combining them to get the desired output.

To use divide and conquer algorithms, **recursion** is used.

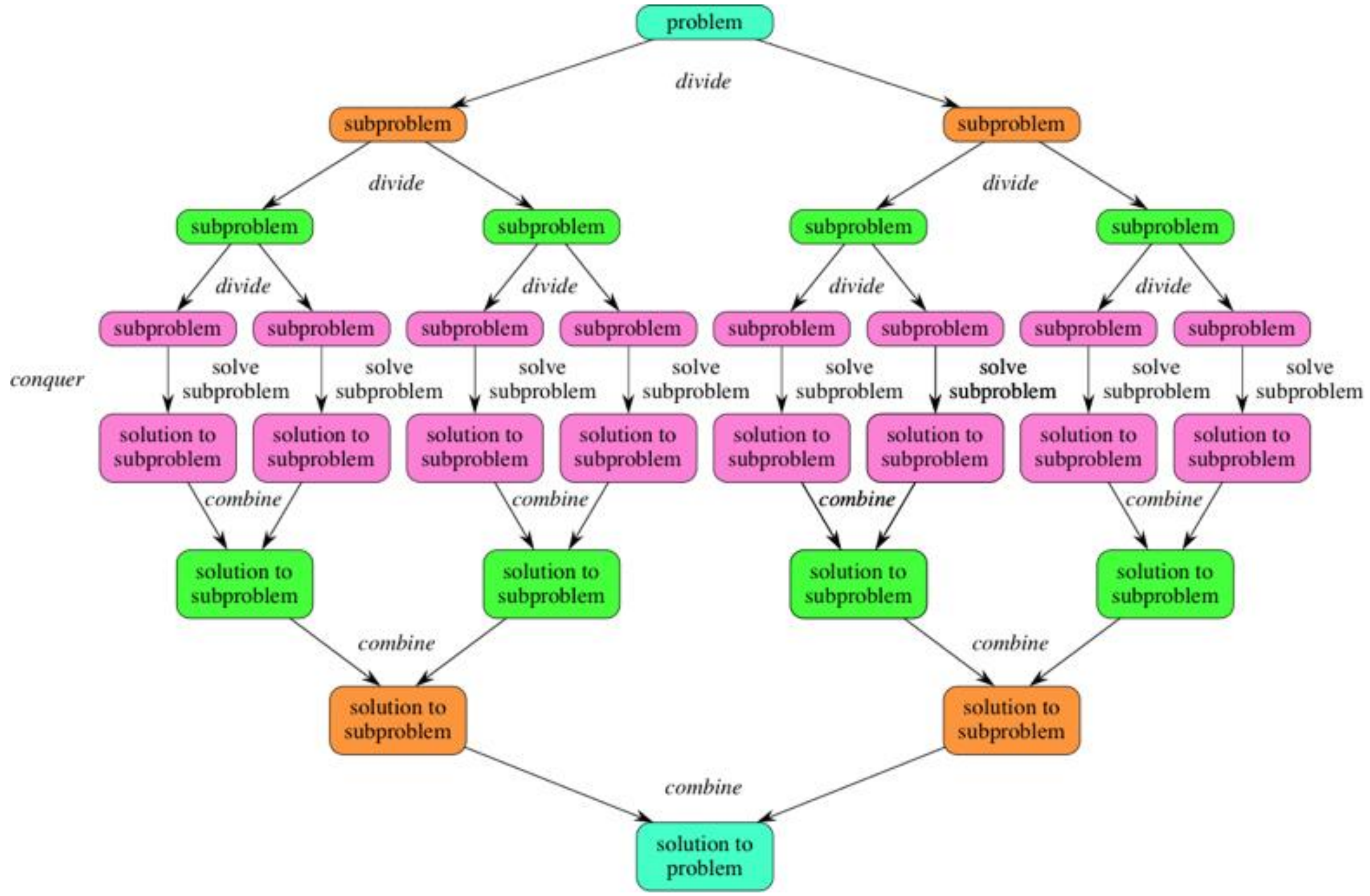
Divide-and-Conquer Technique


a problem of size *n*
subproblem 1 of size *n*/2
subproblem 2 of size *n*/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

It general leads to a recursive algorithm!


problem
divide
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
combine
solution to problem

If we expand out two more recursive steps, it looks like this:



Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

Divide and Conquer Application

- Binary Search
- Merge Sort
- Quick Sort

Greedy Algorithms

History of Greedy Algorithms

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Esdger Djikstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.
- In the '70s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.
- The greedy paradigm was registered as a different type of optimization strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

As the name implies, this is a simple approach which tries to find the **best** solution at every step. Thus, it aims to find the local optimal solution at every step so as to find the global optimal solution for the entire problem.

Consider that there is an **objective function** that has to be optimized (maximized/ minimized). This approach makes greedy choices at each step and makes sure that the objective function is optimized.

The greedy algorithm has only one chance to compute the optimal solution and thus, cannot go back and look at other alternate solutions. However, in many problems, this strategy fails to produce a global optimal solution. Let's consider the following binary tree to understand how a basic greedy algorithm works:

$$\begin{tikzcd}
 & & & 8 \arrow[dll] \arrow[drr] & & & \\
 & 2 \arrow[dl] \arrow[dr] & & & & 12 \arrow[dl] \arrow[dr] & \\
3 & & 89 & & 10 & & 5
\end{tikzcd}$$

For the above problem the objective function is:

To find the path with largest sum.

Since we need to maximize the objective function, Greedy approach can be used. Following steps are followed to find the solution:

Step 1: Initialize sum = 0

Step 2: Select the root node, so its value will be added to sum, sum = 0+8 = 8

Step 3: The algorithm compares nodes at next level, selects the largest node which is 12, making the sum = 20.

Step 4: The algorithm compares nodes at the next level, selects the largest node which is 10, making the sum = 30.

Thus, using the greedy algorithm, we get 8-12-10 as the path. But this is not the optimal solution, since the path 8-2-89 has the largest sum ie 99.

This happens because the algorithm makes decision based on the information available at each step without considering the overall problem.

When to use Greedy Algorithms?

For a problem with the following properties, we can use the greedy technique:

Greedy Choice Property: This states that a globally optimal solution can be obtained by locally optimal choices.

Optimal Sub-Problem: This property states that an optimal solution to a problem, contains within it, optimal solution to the sub-problems. Thus, a globally optimal solution can be constructed from locally optimal sub-solutions.

Generally, **optimization problem**, or the problem where we have to find maximum or minimum of something or we have to find some optimal solution, greedy technique is used.

An optimization problem has two types of solutions:

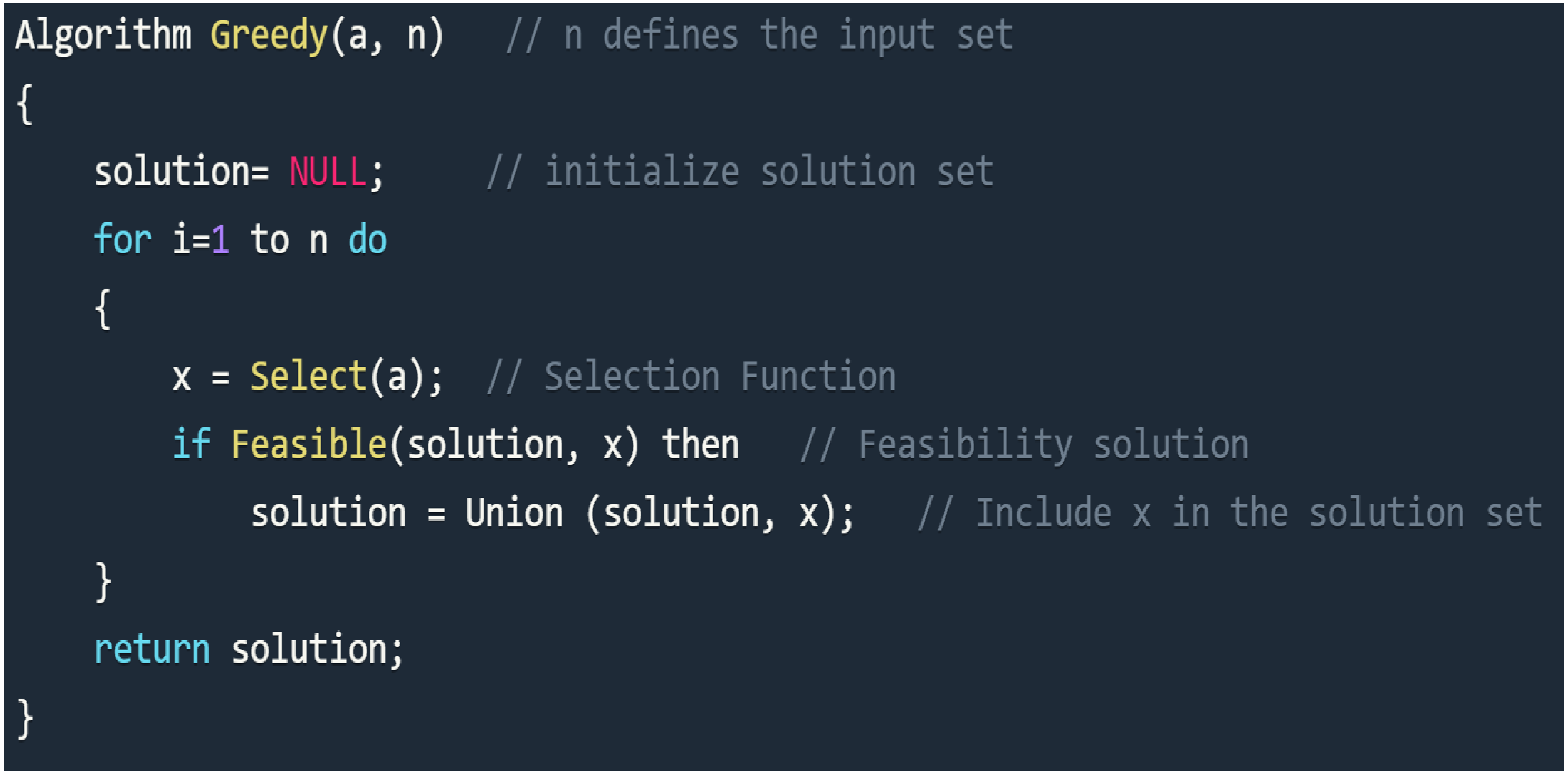
Feasible Solution: This can be referred as approximate solution (subset of solution) satisfying the objective function and it may or may not build up to the optimal solution.

Optimal Solution: This can be defined as a feasible solution that either maximizes or minimizes the objective function.

Key Terminologies used in Greedy Algorithms

- **Objective Function**: This can be defined as the function that needs to be either maximized or minimized.
- **Candidate Set**: The global optimal solution is created from this set.
- **Selection Function**: Determines the best candidate and includes it in the solution set.
- **Feasibility Function**: Determines whether a candidate is feasible and can contribute to the solution.

Standard Greedy Algorithm



```
Algorithm Greedy(a, n)   // n defines the input set
{
    solution= NULL;     // initialize solution set
    for i=1 to n do
    {
        x = Select(a);  // Selection Function
        if Feasible(solution, x) then   // Feasibility solution
            solution = Union (solution, x);   // Include x in the solution set
    }
    return solution;
}
```

Advantages of Greedy Approach/Technique

- This technique is easy to formulate and implement.
- It works efficiently in many scenarios.
- This approach minimizes the time required for generating the solution.

Disadvantages of Greedy Approach/Technique

This approach does not guarantee a global optimal solution since it never looks back at the choices made for finding the local optimal solution.

Few popular problems which use greedy technique:

- Knapsack Problem
- Activity Selection Problem
- Dijkstra’s Problem
- Prim’s Algorithmfor finding Minimum Spanning Tree
- Kruskal’s Algorithmfor finding Minimum Spanning Tree
- Huffman Coding
- Travelling Salesman Problem

Coin changing problem

- Problem: Return correct change using a minimum number of coins.
- Greedy choice: coin with highest coin value
- A greedy solution (next slide):
- American money

  The amount owed = 37 cents.

  The change is: 1 quarter, 1 dime, 2 cents.

  Solution is optimal.
- Is it optimal for all sets of coin sizes?
- Is there a solution for all sets of coin sizes?

A Greedy Solution:

```
Input: Set of coins of different denominations, amount-owed
change = {}
while (more coin-sizes && valueof(change)<amount-owed)
    Choose the largest remaining coin-size // Selection
    // feasibility check
      while (adding the coin does not make the
       valueof(change)   exceed the amount-owed ) then
       add coin to change
      //check if solved
       if ( valueof(change) equals amount-owed)
       return change
    else delete coin-size
return "failed to compute change"
```

Elements of the Greedy Strategy

Cast problem as one in which we make a greedy choice and are left with one subproblem to solve.

To show optimality:

1. Prove there is always an **optimal solution** to original problem that makes the greedy choice.

Elements of the Greedy Strategy

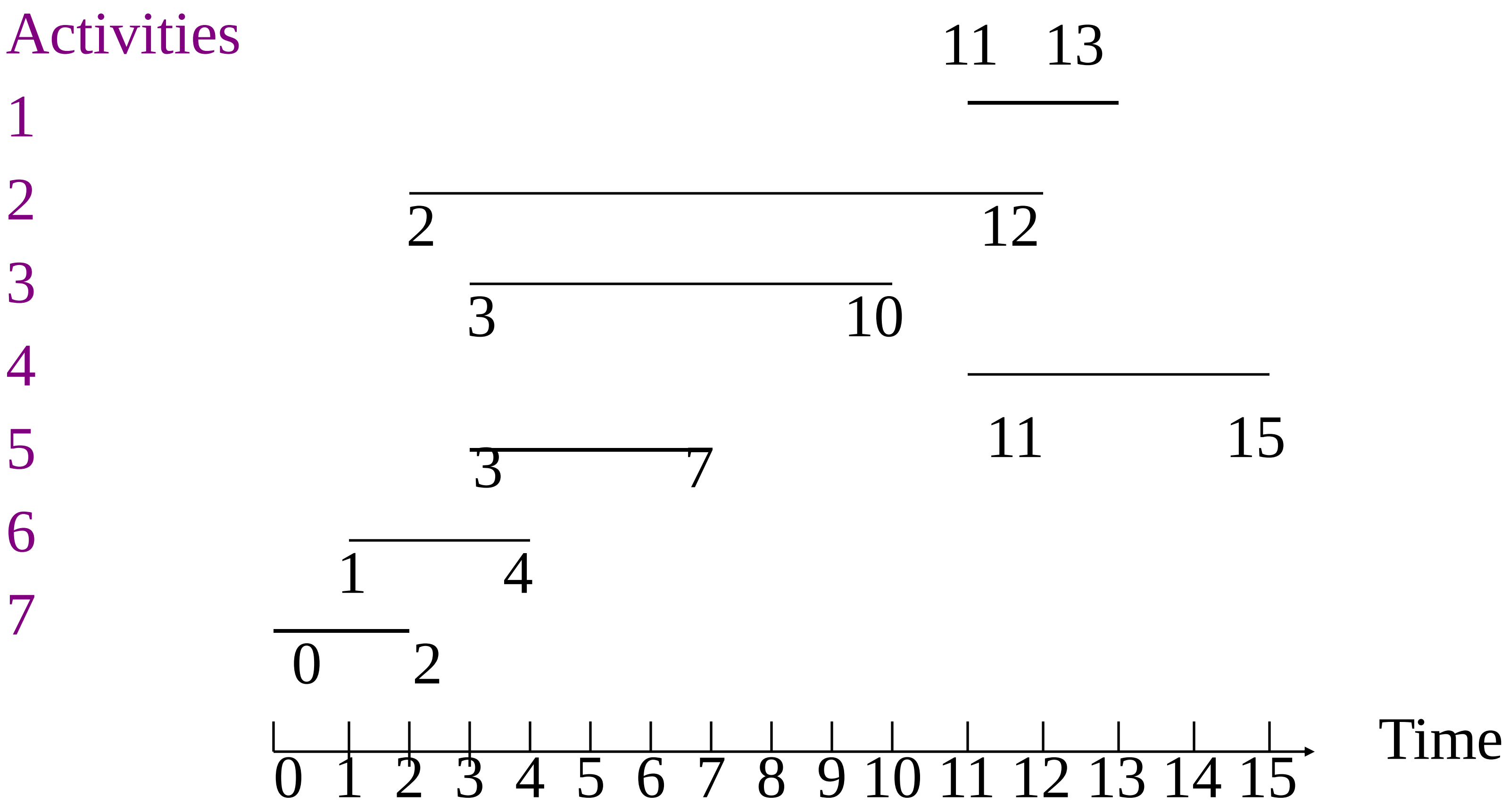
2. Demonstrate that what remains is a subproblem with property:

   If we combine the optimal solution of the subproblem with the greedy choice we have an optimal solution to original problem.

Activity Selection

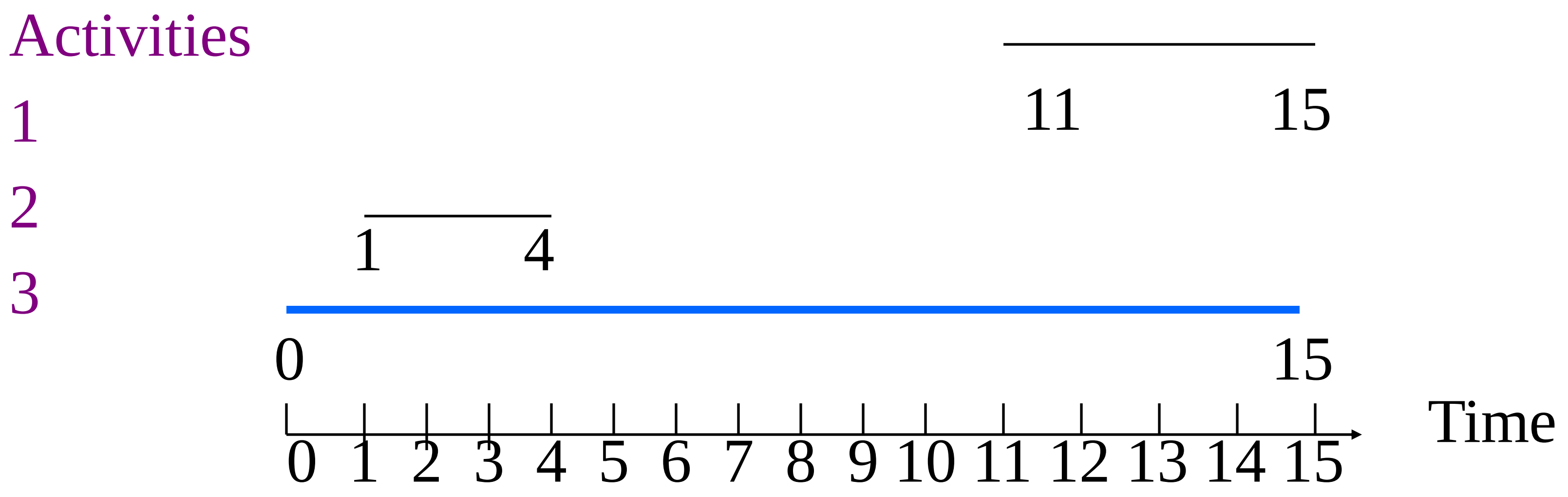
- Given a set *S* of *n activities* with *start* time s_i and *finish* time f_i of activity *i*
- Find a maximum size subset *A* of **compatible** activities (maximum *number* of activities).
- Activities are **compatible** if they do not overlap
- Can you suggest a greedy choice?

Example


Activities
1
2
3
4
5
6
7
11
13
2
12
3
10
11
15
3
7
1
4
0
2
Time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

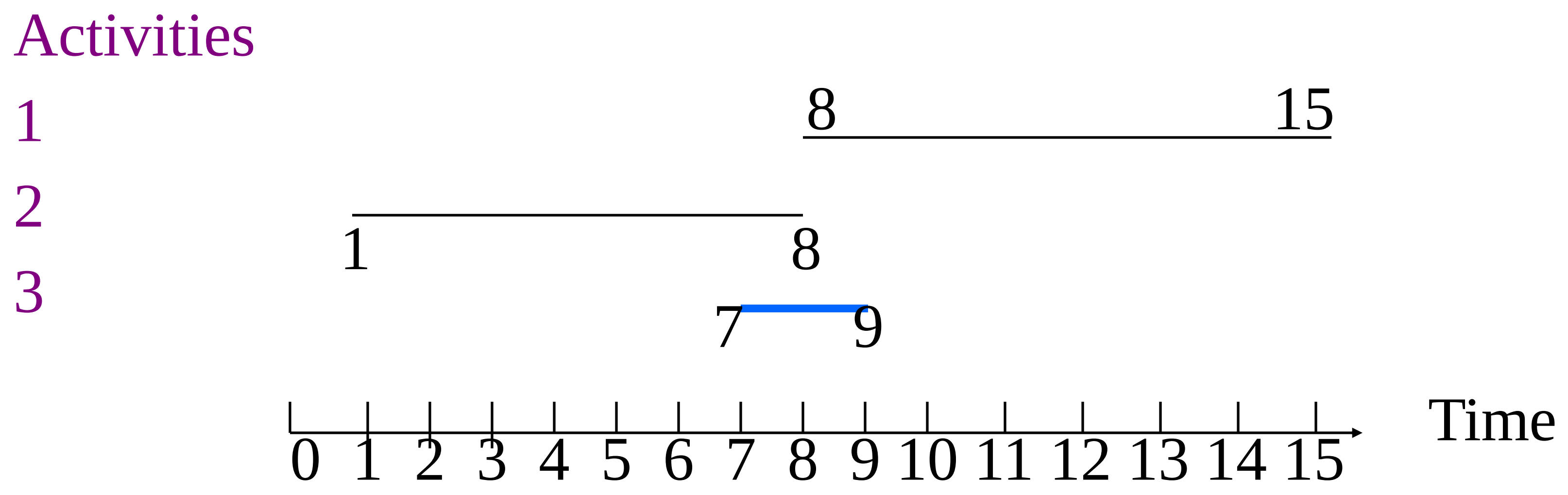
Counter Example 1

- Select by start time

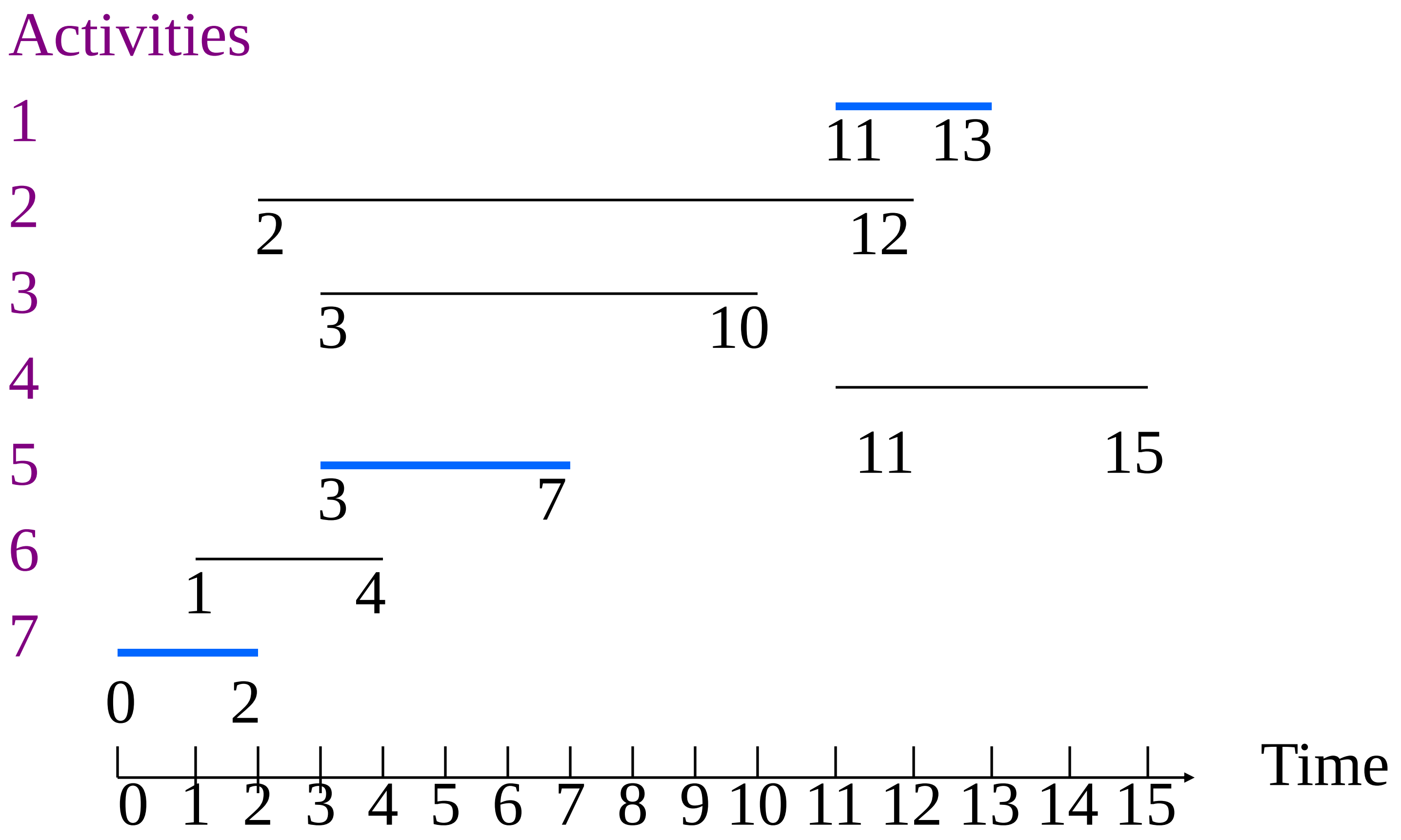


Counter Example 2

- Select by minimum duration



Select by finishing time



Activity Selection

- Assume without loss of generality that we number the intervals in order of *finish time*. So $f_1 \leq \ldots \leq f_n$.
- Greedy choice: choose activity with minimum finish time
- The following greedy algorithm starts with A={1} and then adds all compatible jobs. (Theta(n))
- Theta(nlogn) when including sort

Greedy-Activity-Selector(s,f)

```
n = length[s] // number of activities
A = {1}
j = 1 //last activity added
for i = 2 to n //select
   if $s_i >= f_j$ then //compatible (feasible)
      add {i} to A
      j = i //save new last activity
return A
```

Dynamic Programming

Dynamic programming (usually referred to as **DP**) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again.. shortly *'Remember your Past'* :) . If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, if you observe some over-lapping subproblems, then its a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem

Overview of Serial Dynamic Programming

- *Dynamic programming* (DP) is used to solve a wide variety of discrete optimization problems such as scheduling, string-editing, packaging, and inventory management.
- Break problems into subproblems and combine their solutions into solutions to larger problems.
- In contrast to divide-and-conquer, there may be relationships across subproblems.

- Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.
- Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that –

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

There are two ways of doing this.

1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

2.) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as ***Dynamic Programming***.

Example

The following computer problems can be solved using dynamic programming approach –

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call **Memoization** - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

A code for it using pure recursion:

```
int fib (int n)
{
        if (n < 2)
            return 1;
        return fib(n-1) + fib(n-2);
}
```

Memoization

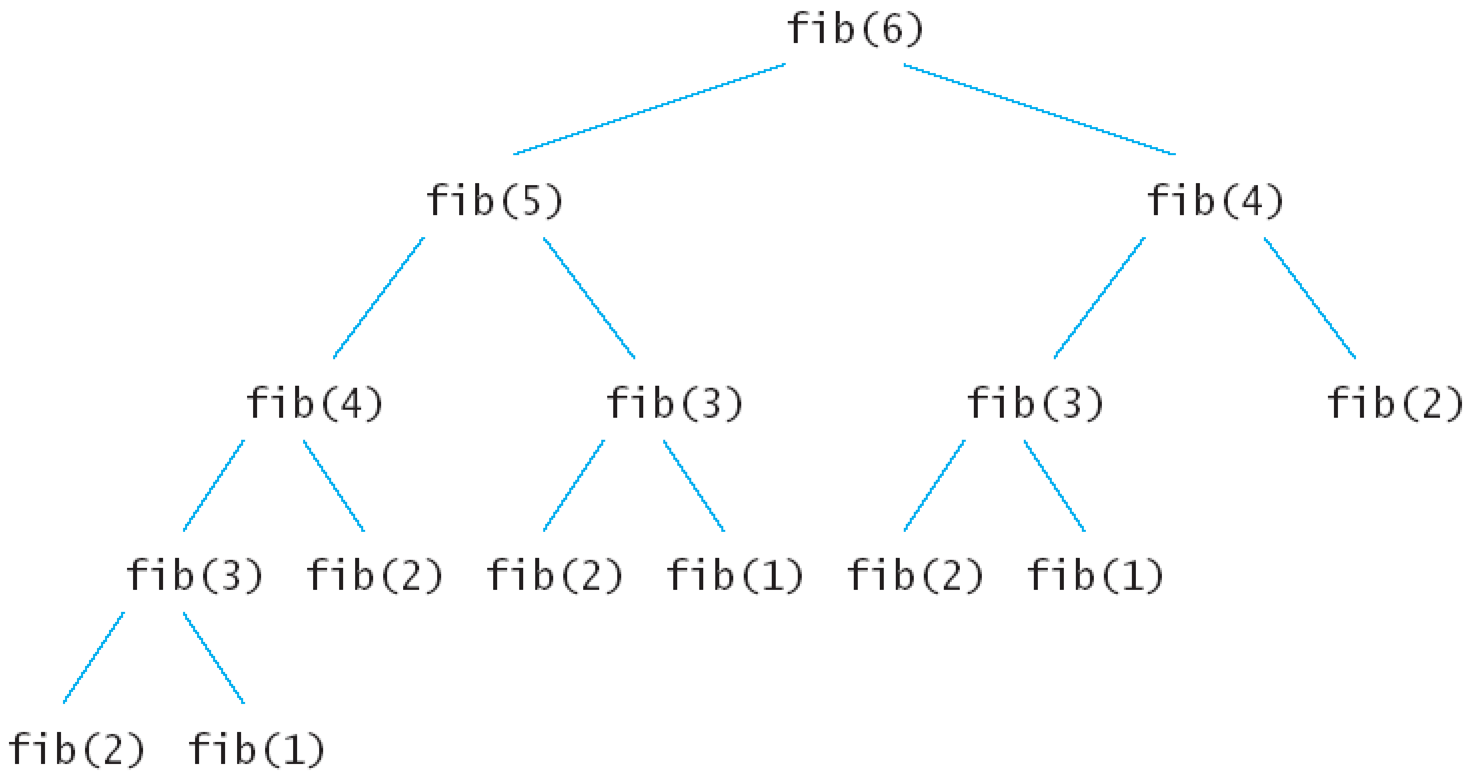
In computing, memoization is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously processed inputs. Memoization has also been used in other contexts, such as in simple mutually recursive descent parsing in a general top-down parsing algorithm that accommodates ambiguity and left recursion in polynomial time and space. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, memoization is also known as tabling; see also lookup table.

Using Dynamic Programming approach with memoization:

```
void fib () {
        fibresult[0] = 1;
        fibresult[1] = 1;
        for (int i = 2; i<n; i++)
           fibresult[i] = fibresult[i-1] + fibresult[i-2];
    }
```

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes.

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)

Majority of the Dynamic Programming problems can be categorized into two types:

1. **Optimization problems.**
2. **Combinatorial problems.**

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Memoization is very easy to code and might be your first line of approach for a while. Though, with dynamic programming, you don't risk blowing stack space, you end up with lots of liberty of when you can throw calculations away. The downside is that you have to come up with an ordering of a solution which works.

One can think of dynamic programming as a table-filling algorithm: you know the calculations you have to do, so you pick the best order to do them in and ignore the ones you don't have to fill in.

Let's look at a sample problem:

Let us say that you are given a number N, you've to find the number of different ways to write it as the sum of 1, 3 and 4.

For example, if N = 5, the answer would be 6.

- 1 + 1 + 1 + 1 + 1
- 1 + 4
- 4 + 1
- 1 + 1 + 3
- 1 + 3 + 1
- 3 + 1 + 1

Sub-problem: DP_n be the number of ways to write **N** as the sum of 1, 3, and 4.

Finding recurrence: Consider one possible solution, n = x1 + x2 + ... x_n. If the last number is 1, the sum of the remaining numbers should be n - 1. So, number of sums that end with 1 is equal to $DP_{n-1.}$. Take other cases into account where the last number is 3 and 4. The final recurrence would be:

$$DP_n = DP_{n-1} + DP_{n-3} + DP_{n-4}.$$

Take care of the base cases. $DP_0 = DP_1 = DP_2 = 1$, and $DP_3 = 2$.

Implementation:

```
DP[0] = DP[1] = DP[2] = 1;
DP[3] = 2;
for (i = 4; i <= n; i++)
{
        DP[i] = DP[i-1] + DP[i-3] + DP[i-4];
}
```

The technique above, takes a bottom up approach and uses memoization to not compute results that have already been computed.

Brute Force Algorithms

Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.

So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take 10^4, or 10,000 tries to find your combination.

- A classic example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance traveled is minimized?
- The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.
- The time complexity of brute force is **O(mn)**, which is sometimes written as **O(n*m)** . So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Brute force approach is not an important algorithm design strategy for the following reasons:

- First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. Its used for many elementary but algorithmic tasks such as computing the sum of n numbers, finding the largest element in a list and so on.
- Second, for some important problems—e.g., sorting, searching, matrix multiplication, string matching—the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.

- Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
- Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
- Finally, a brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

Backtracking

A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output.

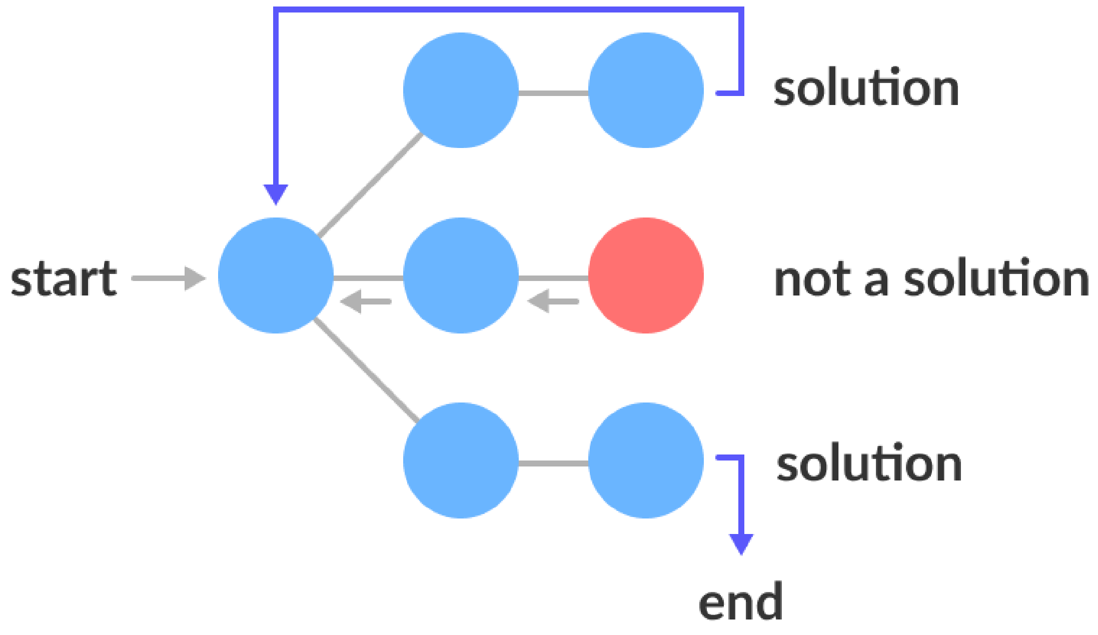
The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.

The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

This approach is used to solve problems that have multiple solutions.

State Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.



Backtracking Algorithm

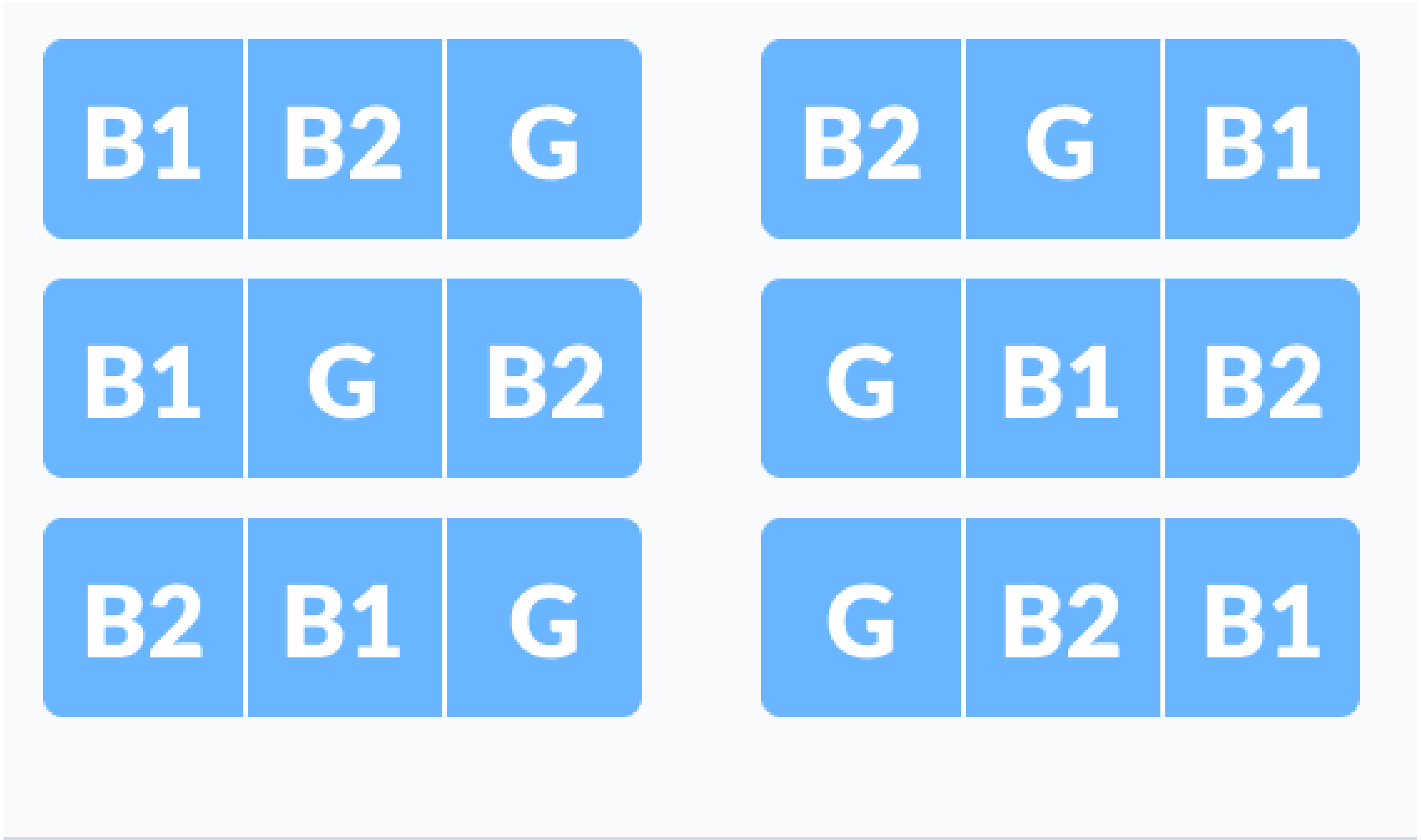
```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

Example Backtracking Approach

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle bench.

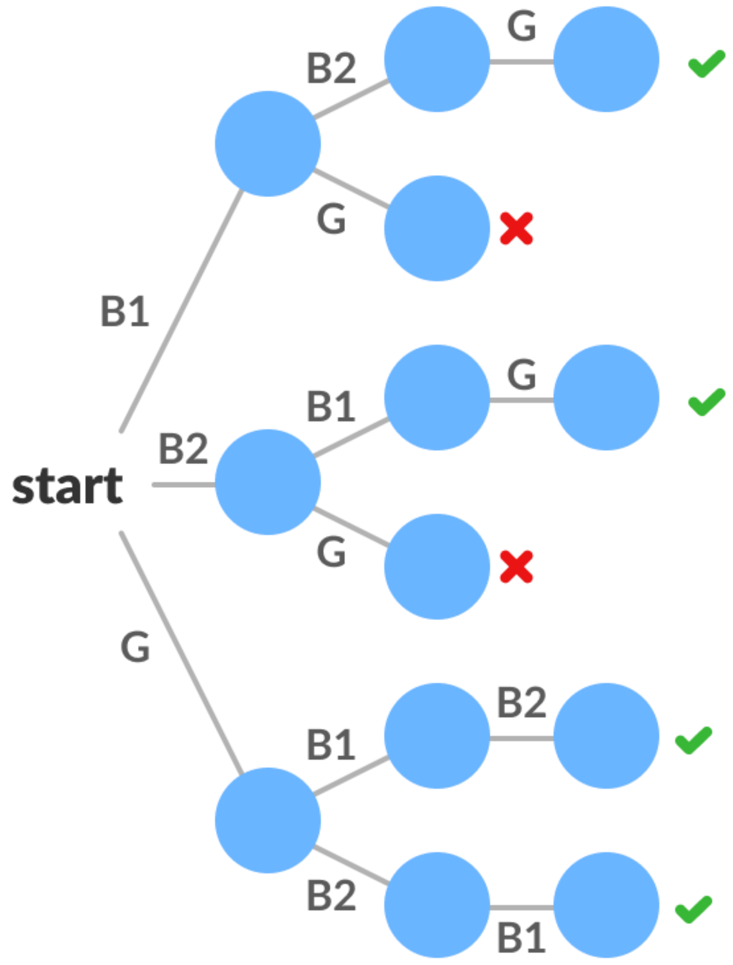
Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

All the possibilities are:



All the possibilities

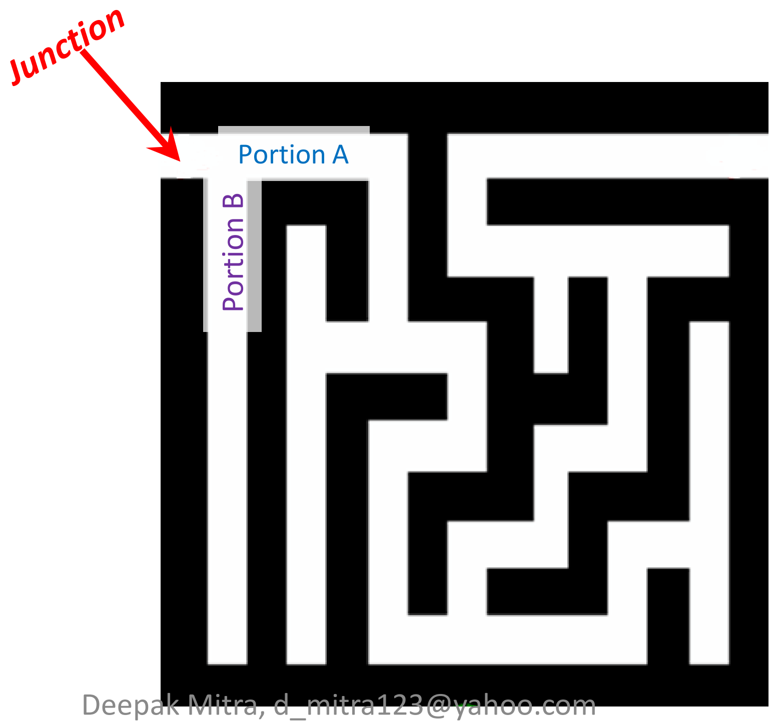
The following state space tree shows the possible solutions.



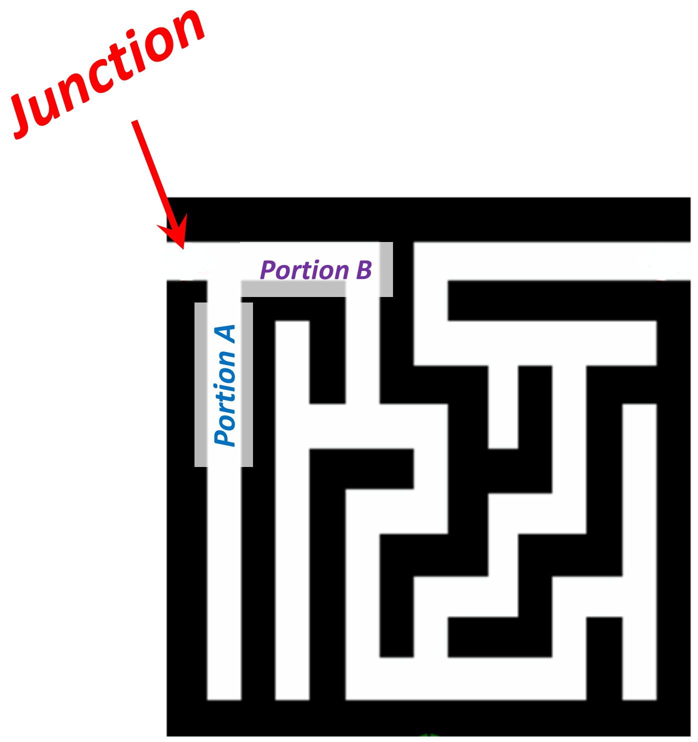
State tree with all the solutions

Maze Problem

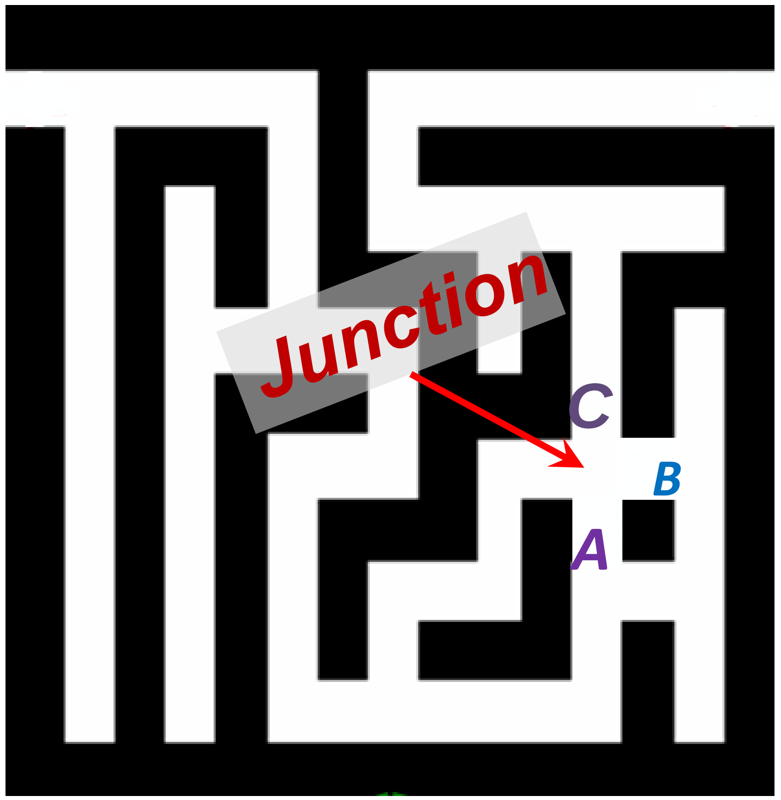
- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point in a maze, you might have two options of which direction to go:



- One strategy would be to try going through **Portion A** of the maze.
    - If you get stuck before you find your way out, then you ***"backtrack"*** to the junction.
- At this point in time you know that **Portion A** will ***NOT*** lead you out of the maze,
    - so you then start searching in **Portion B**

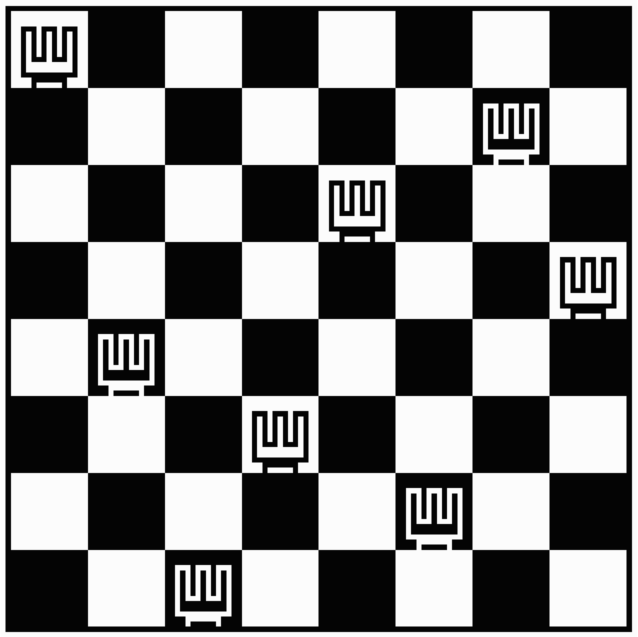


- Clearly, at a single junction you could have even more than 2 choices.
- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, ***"backtrack"*** to the junction and try the next choice.
- If you try all choices and never found a way out, then there IS no solution to the maze.



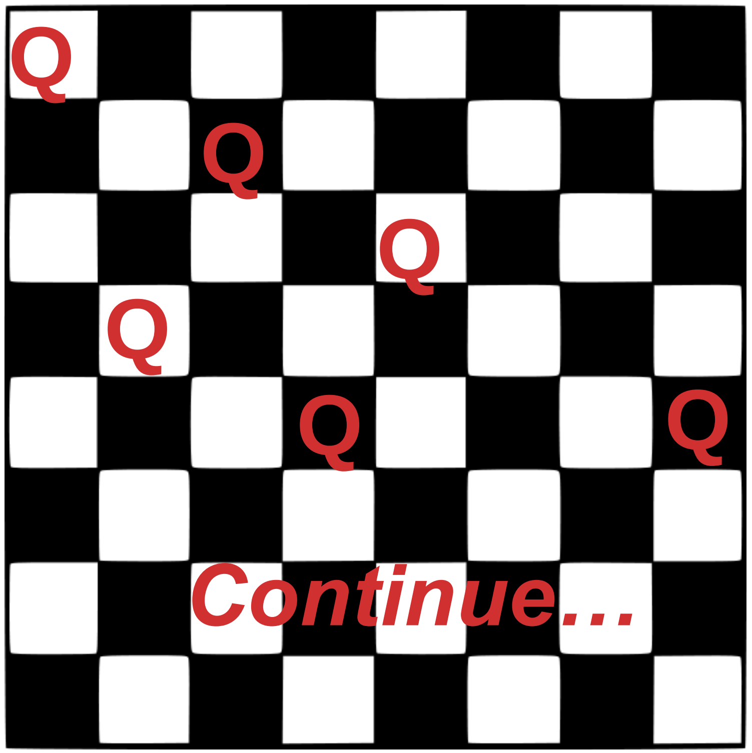
Eight Queens Problem

- Find an arrangement of **8** queens on a single chess board such that no two queens are attacking one another.
- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).
  - Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.



Eight Queens Problem

- The backtracking strategy is as follows:
  1) Place a queen on the first available square in row 1.
  2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
  3) Continue in this fashion until either:
     a) you have solved the problem, or
     b) you get stuck.
        – When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.



Animated Example:
http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

Eight Queens Problem

- When we carry out backtracking, an easy way to visualize what is going on is a tree that shows all the different possibilities that have been tried.
- On the board we will show a visual representation of solving the 4 Queens problem (placing 4 queens on a 4x4 board where no two attack one another).

Eight Queens Problem

- The neat thing about coding up backtracking, is that it can be done recursively, without having to do all the bookkeeping at once.
  - Instead, the stack or recursive calls does most of the bookkeeping
  - (ie, keeping track of which queens we've placed, and which combinations we've tried so far, etc.)

Branch and Bound
Searching Strategies

The Branch and Bound (BB or B&B) algorithm is first proposed by A. H. Land and A. G. Doig in 1960 for discrete programming. It is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. A branch and bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are fathomed, by using upper and lower estimated bounds of the quantity being optimized.

NP-Completeness

So far we've seen a lot of good news: such-and-such a problem can be solved quickly (in close to linear time, or at least a time that is some small polynomial function of the input size).NP-completeness is a form of bad news: evidence that many important problems can't be solved quickly.

Why should we care?

These NP-complete problems really come up all the time. Knowing they're hard lets you stop beating your head against a wall trying to solve them, and do something better:

Use a heuristic. If you can't quickly solve the problem with a good worst case time, maybe you can come up with a method for solving a reasonable fraction of the common cases.

Solve the problem approximately instead of exactly. A lot of the time it is possible to come up with a provably fast algorithm, that doesn't solve the problem exactly but comes up with a solution you can prove is close to right.

Use an exponential time solution anyway. If you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution.

Choose a better abstraction. The NP-complete abstract problem you're trying to solve presumably comes from ignoring some of the seemingly unimportant details of a more complicated real world problem. Perhaps some of those details shouldn't have been ignored, and make the difference between what you can and can't solve.

Classification of problems

The subject of *computational complexity theory* is dedicated to classifying problems by how hard they are. There are many different classifications; some of the most common and useful are the following. (One technical point: these are all really defined in terms of yes-or-no problems -- does a certain structure exist rather than how do I find the structure.)

P. Problems that can be solved in polynomial time. ("P" stands for polynomial.) These problems have formed the main material of this course.

NP. This stands for "nondeterministic polynomial time" where nondeterministic is just a fancy way of talking about guessing a solution. A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution). Problems in NP are still relatively easy: if only we could guess the right solution, we could then quickly test it.

NP-completeness

The theory of NP-completeness is a solution to the practical problem of applying complexity theory to individual problems. NP-complete problems are defined in a precise sense as the hardest problems in P. Even though we don't know whether there is any problem in NP that is not in P, we can point to an NP-complete problem and say that if there are any hard problems in NP, that problems is one of the hard ones.(Conversely if everything in NP is easy, those problems are easy. So NP-completeness can be thought of as a way of making the big P=NP question equivalent to smaller questions about the hardness of individual problems.)

So if we believe that P and NP are unequal, and we prove that some problem is NP-complete, we should believe that it doesn't have a fast algorithm.

For unknown reasons, most problems we've looked at in NP turn out either to be in P or NP-complete. So the theory of NP-completeness turns out to be a good way of showing that a problem is likely to be hard, because it applies to a lot of problems. But there are problems that are in NP, not known to be in P, and not likely to be NP-complete; for instance the code-breaking example I gave earlier.

Difference between NP hard and NP complete problem

NP Problem:

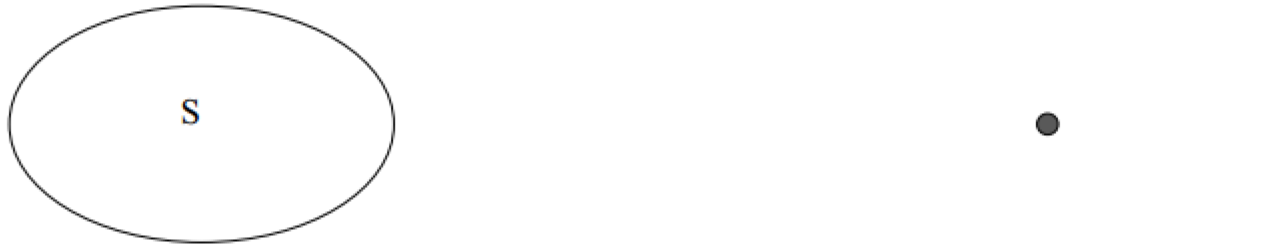
The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.

NP-Hard Problem:

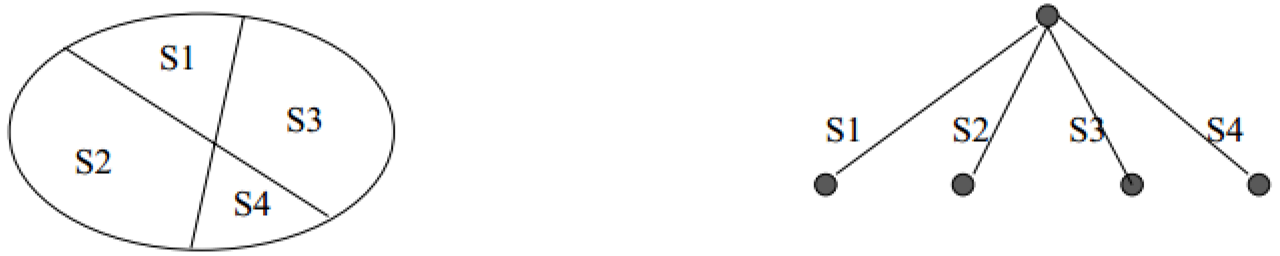
Any decision problem P_i is called NP-Hard if and only if every problem of NP(say P<subj) is reducible to P_i in polynomial time.

NP-Complete Problem: Any problem is NP-Complete if it is a part of both NP and NP-Hard Problem.

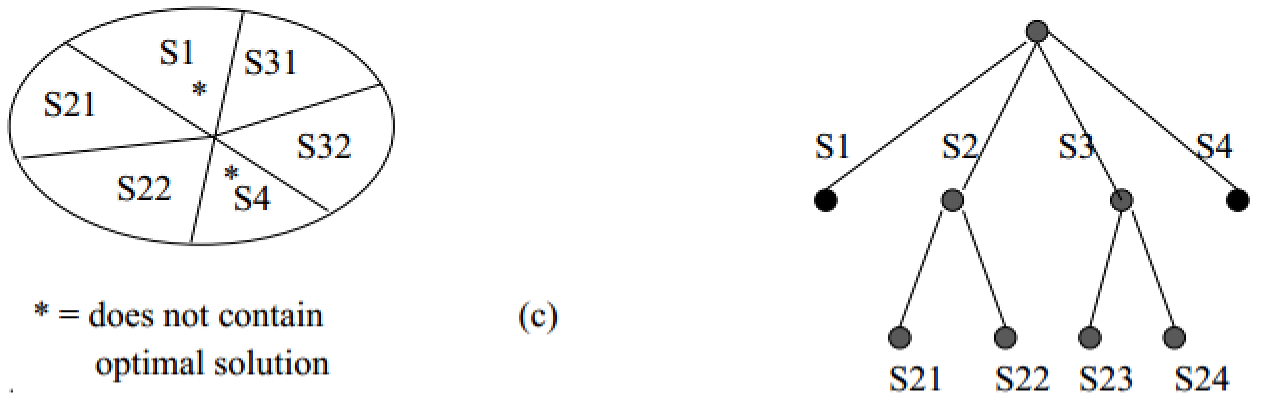
Branch and Bound algorithm, as a method for global optimization for discrete problems, which are usually NP-hard, searches the complete space of solutions for a given problem for the optimal solution. By solving a relaxed problem of the original one, fractional solutions are recognized and for each discrete variable, B&B will do branching and creating two new nodes, thus dividing the solution space into a set of smaller subsets and obtain the relative upper and lower bound for each node. Since explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions, the use of bounds for the function to be optimized combined with the value of the current best solution found enables this B&B algorithm to search only parts of the solution space implicitly.


S

(a)


S1
S2
S3
S4
S1
S2
S3
S4

(b)


S1
S31
S21
S32
S22
S4
*
*
* = does not contain optimal solution
S1
S2
S3
S4
S21
S22
S23
S24

(c)

Branching Strategy

According to the work of Gupta and Ravindran, Generally there are two ways to do branching:

Branching on the node with the smallest bound

Search all the nodes and find the one with the smallest bound and set it as the next branching node. Advantage: Generally it will inspect less subproblems and thus saves computation time. Disadvantage: Normally it will require more storage.

Branching on the newly created node with the smallest bound

Search the newly created nodes and find the one with the smallest bound and set it as the next branching node. Advantage: Saves storage space. Disadvantage: Require more branching computation and thus less computational efficiently.

Feasible Solution vs. Optimal Solution

- DFS, BFS, hill climbing and best-first search can be used to solve some searching problem for searching a feasible solution.
- However, they cannot be used to solve the optimization problems for searching an (the) optimal solution.

The branch-and-bound strategy

- This strategy can be used to solve optimization problems without an exhaustive search in the average case.

Branch-and-bound strategy

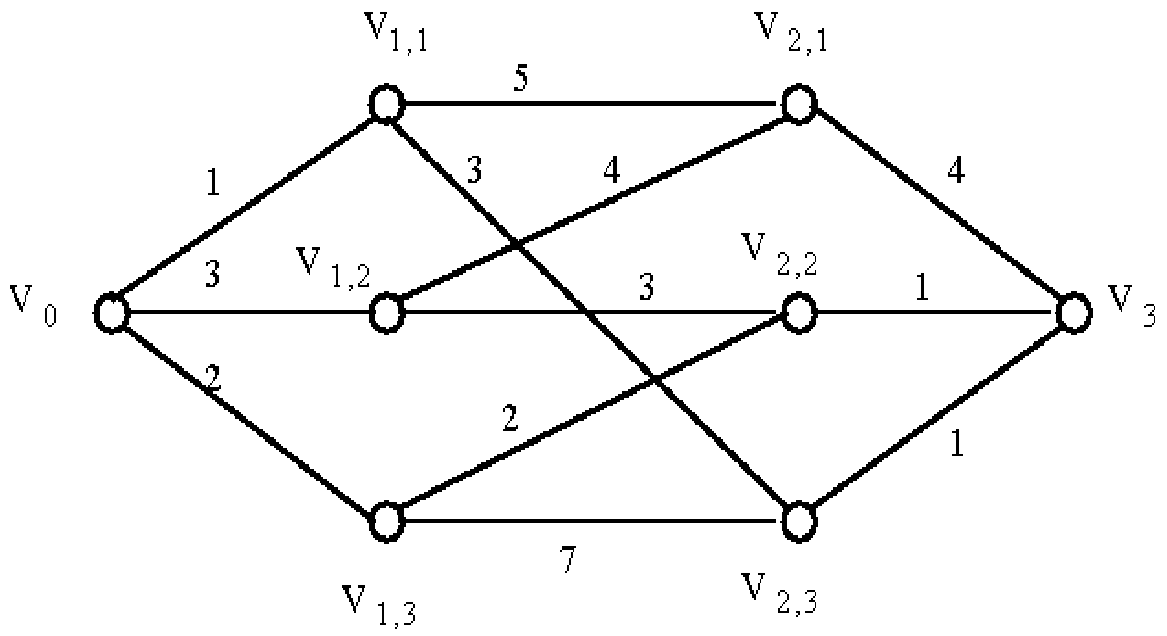
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many braches can be terminated

Branch-and-bound strategy

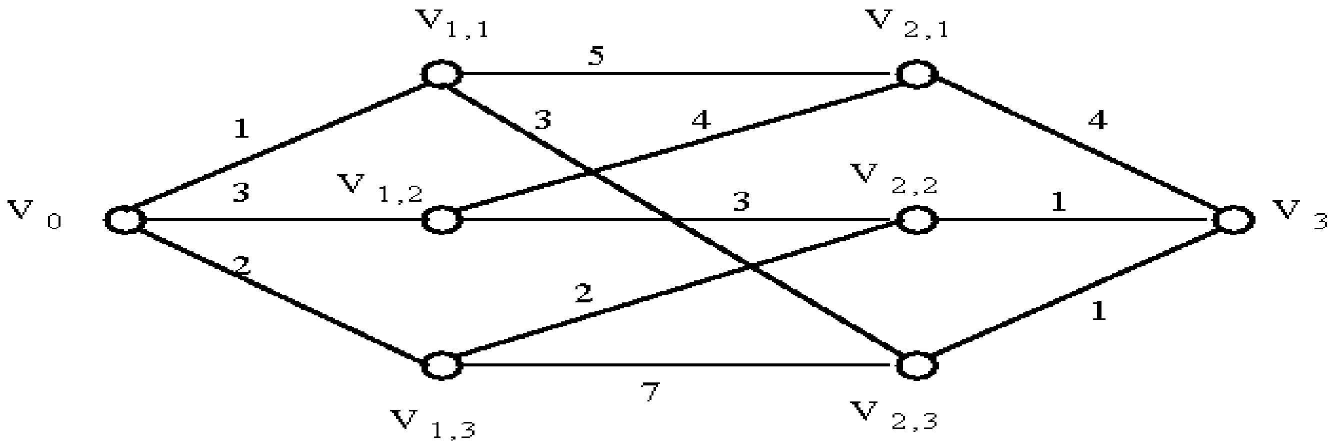
- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

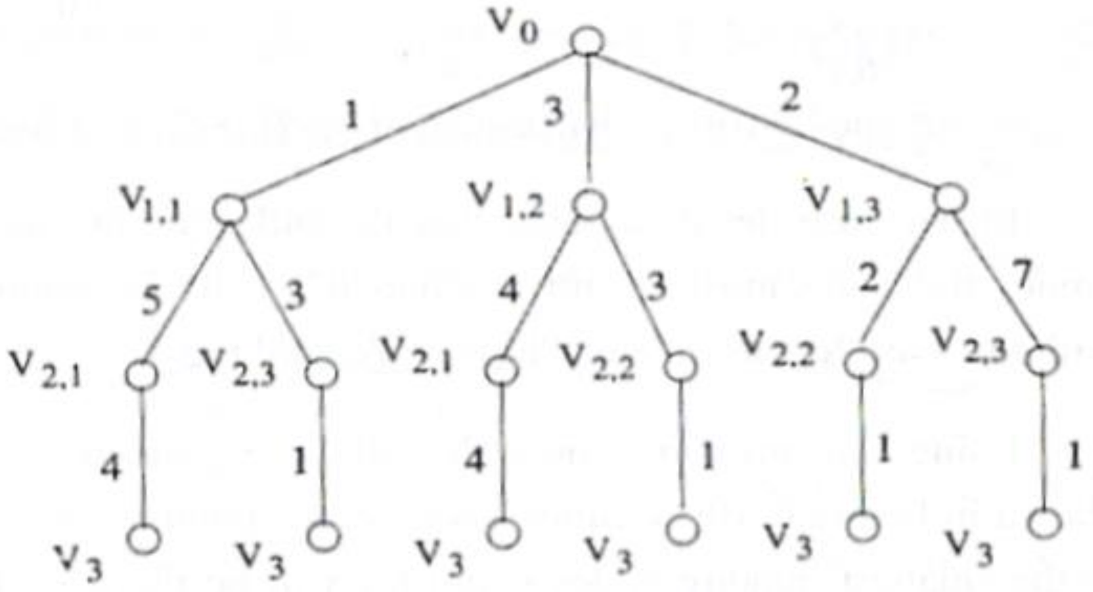
A Multi-Stage Graph Searching Problem.

Find the shortest path from V_0 to V_3

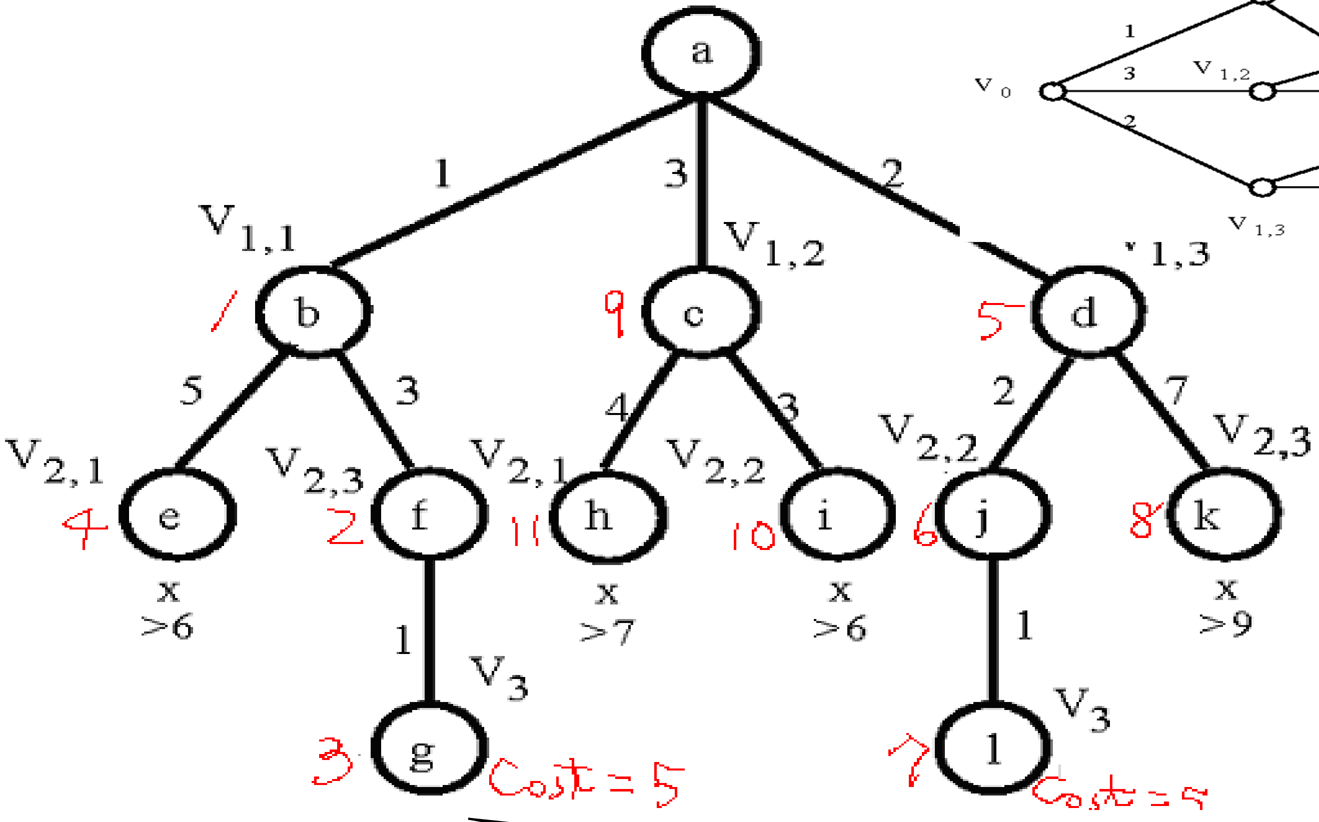


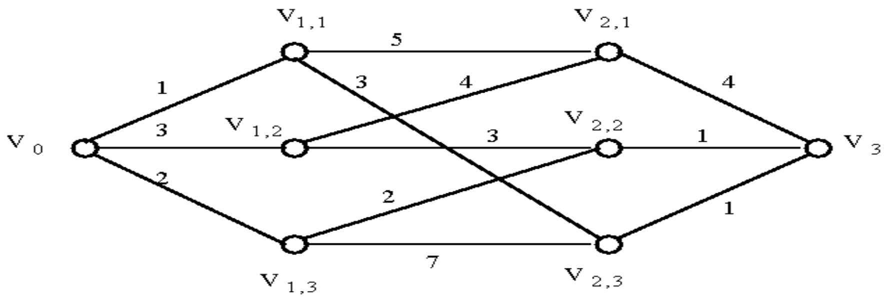
E.G.:A Multi-Stage Graph Searching Problem





Solved by branch-and-bound (hill-climbing with bounds)





A feasible solution is found whose cost is equal to 5.

An upper bound of the optimal solution is first found here.

The traveling salesperson optimization problem

- Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimal cost.
- It is NP-hard.
- We try to avoid n! exhaustive search by the branch-and-bound technique on the average case. (Recall that $O(n!)>O(2^n)$.)

The traveling salesperson optimization problem

- E.g. A Cost Matrix for a Traveling Salesperson Problem.

i \ j	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

The basic idea

- There is a way to split the solution space (branch)
- There is a way to predict a lower bound for a class of solutions. There is also a way to find a upper bound of an optimal solution. If the lower bound of a solution exceeds the upper bound, this solution cannot be optimal and thus we should terminate the branching associated with this solution.

The traveling salesperson optimization problem

- The Cost Matrix for a Traveling Salesperson Problem.

Step 1 to reduce: Search each row for the smallest value

to j

from i

i \ j	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

Step 2 to reduce: Search each column for the smallest value

The traveling salesperson optimization problem

- Reduced cost matrix:

i \ j	1	2	3	4	5	6	7	
1	∞	0	90	10	30	6	54	(-3)
2	0	∞	73	38	17	12	30	(-4)
3	29	1	∞	20	0	12	9	(-16)
4	32	83	73	∞	49	0	84	(-7)
5	3	21	63	8	∞	0	32	(-25)
6	0	85	15	43	89	∞	4	(-3)
7	18	0	7	1	58	13	∞	(-26)

reduced:84

A Reduced Cost Matrix.

The traveling salesperson optimization problem

i \ j	1	2	3	4	5	6	7
1	∞	0	83	9	30	6	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	0	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞
			(-7)	(-1)			(-4)

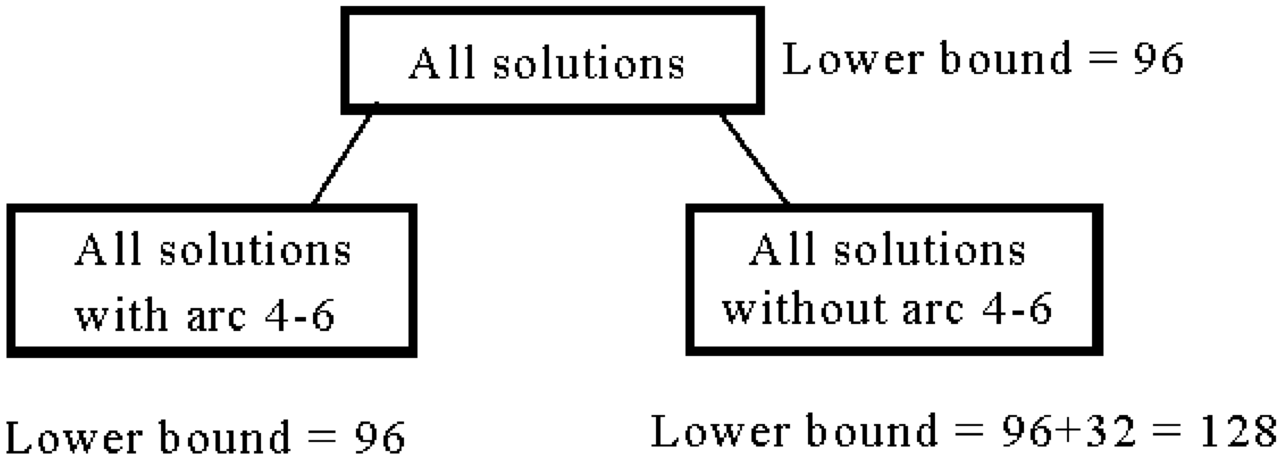
Table 6-5 Another Reduced Cost Matrix.

Lower bound

- The total cost of 84+12=96 is subtracted. Thus, we know the lower bound of feasible solutions to this TSP problem is 96.

The traveling salesperson optimization problem

- Total cost reduced: 84+7+1+4 = 96 (lower bound)
  decision tree:


All solutions
Lower bound = 96
All solutions with arc 4-6
All solutions without arc 4-6
Lower bound = 96
Lower bound = 96+32 = 128

The Highest Level of a Decision Tree.

- If we use arc 3-5 to split, the difference on the lower bounds is 17+1 = 18.

Heuristic to select an arc to split the solution space

- If an arc of cost 0 (x) is selected, then the lower bound is added by 0 (x) when the arc is included.
- If an arc <i,j> is not included, then the cost of the second smallest value (y) in row i and the second smallest value (z) in column j is added to the lower bound.
- Select the arc with the largest (y+z)-x

Searching algorithms

- Given a list, find a specific element in the list

- We will see two types
  - Linear search
    - a.k.a. sequential search
  - Binary search

Algorithm 2: Linear search

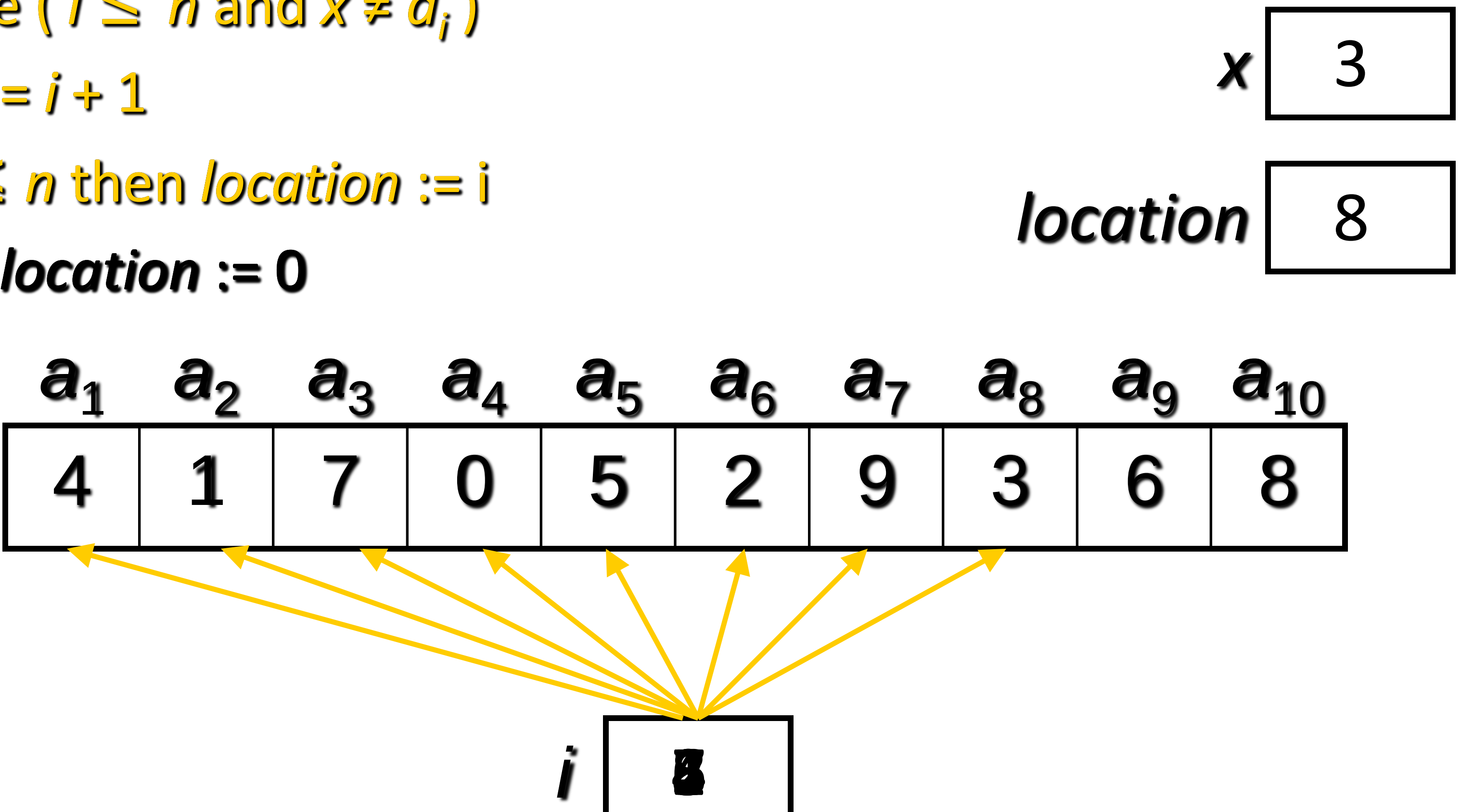
- Given a list, find a specific element in the list
  - List does NOT have to be sorted!

```
procedure linear_search ($x$: integer; $a_1, a_2, \ldots, a_n$: integers)
$i$ := 1
while ( $i \leq n$ and $x \neq a_i$ )
    $i$ := $i$ + 1
if $i \leq n$ then location := i
else location := 0
```

{*location* is the subscript of the term that equals x, or it is 0 if x is not found}

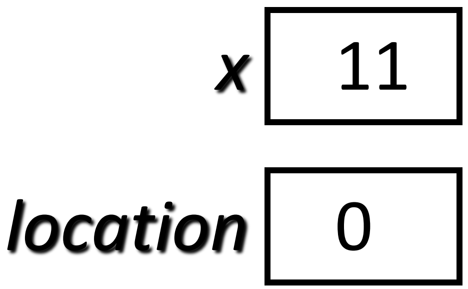
Algorithm 2: Linear search, take 1

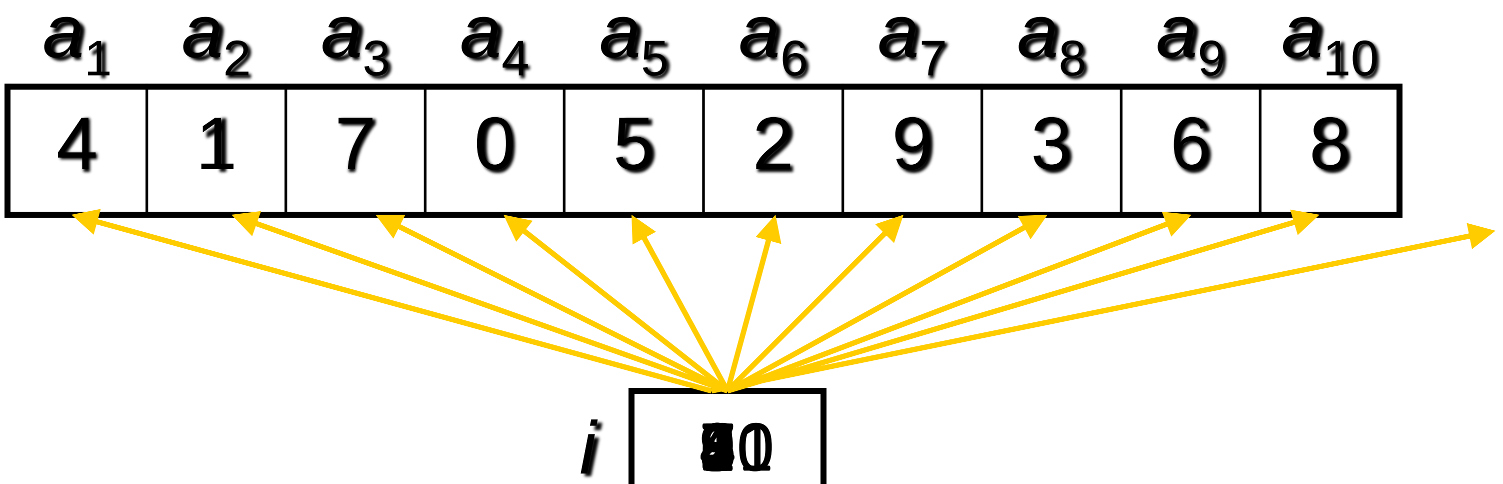
```
procedure linear_search ($x$: integer; $a_1, a_2, \ldots, a_n$: integers)
$i := 1$
while ( $i \leq n$ and $x \neq a_i$ )
    $i := i + 1$
if $i \leq n$ then $location := i$
else $location := 0$
```



Algorithm 2: Linear search, take 2

```
procedure linear_search ($x$: integer; $a_1, a_2, \ldots, a_n$: integers)
$i := 1$
while ( $i \leq n$ and $x \neq a_i$ )
    $i := i + 1$
if $i \leq n$ then $location := i$
else $location := 0$
```





a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
4 1 7 0 5 2 9 3 6 8
i

Linear search running time

- How long does this take?

- If the list has *n* elements, worst case scenario is that it takes *n* “steps”
  - Here, a step is considered a single step through the list

Algorithm 3: Binary search

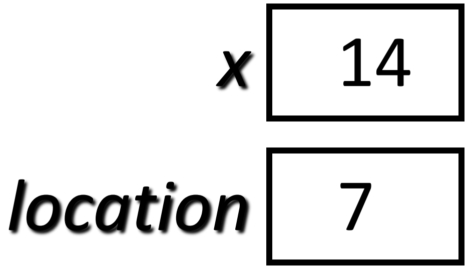
- Given a list, find a specific element in the list
  - List MUST be sorted!
- Each time it iterates through, it cuts the list in half

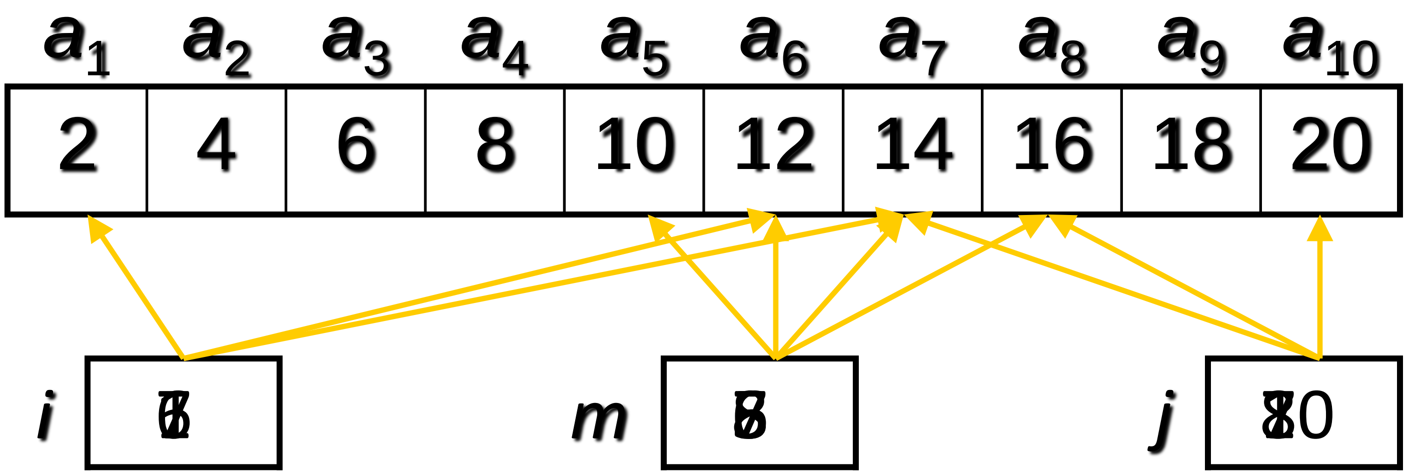
```
procedure binary_search ($x$: integer; $a_1, a_2, \ldots, a_n$: increasing integers)
$i$ := 1{ $i$ is left endpoint of search interval }
$j$ := $n${ $j$ is right endpoint of search interval }
while $i < j$
begin
    m := $\lfloor (i+j)/2 \rfloor$                    { $m$ is the point in the middle }
    if x > $a_m$ then $i$ := $m$+1
    else $j$ := $m$
end
if $x = a_i$ then location := $i$
else location := 0

{location is the subscript of the term that equals x, or it is 0 if x is not found}
```

Algorithm 3: Binary search, take 1

procedure binary_search (x: integer; $a_1, a_2, \ldots, a_n$: increasing integers)

```
i := 1
j := n
while i < j
begin
    m := ⌊(i+j)/2⌋
    if x > a_m then i := m+1
    else j := m
end
if x = a_i then location := i
else location := 0
```





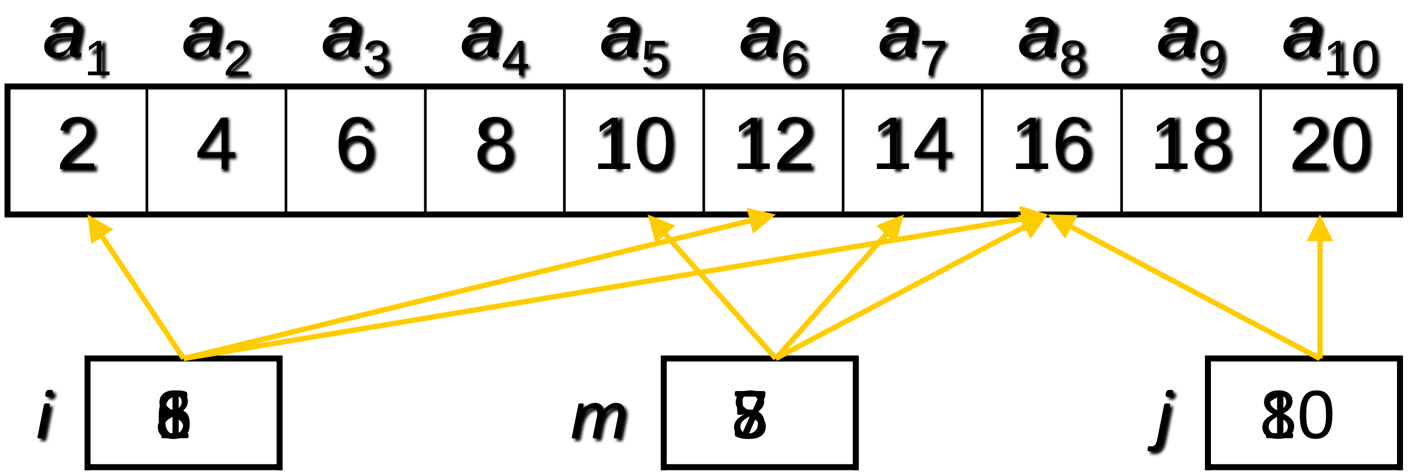
Algorithm 3: Binary search, take 2

procedure binary_search (x: integer; $a_1, a_2, \ldots, a_n$: increasing integers)

```
$i$ := 1
$j$ := $n$
while $i < j$
begin
    m := $\lfloor (i+j)/2 \rfloor$
    if x > $a_m$ then $i$ := $m$+1
    else $j$ := $m$
end
if $x = a_i$ then $location$ := $i$
else $location$ := 0
```

x 15

$location$ 0



a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
2 4 6 8 10 12 14 16 18 20
i 8
m 7
j 8

Algorithm 3: Binary search

- A somewhat alternative view of what a binary search does…

Binary search running time

- How long does this take (worst case)?

- If the list has 8 elements
  - It takes 3 steps
- If the list has 16 elements
  - It takes 4 steps
- If the list has 64 elements
  - It takes 6 steps

- If the list has *n* elements
  - It takes $\log_2 n$ steps

Sorting algorithms

- Given a list, put it into some order
  - Numerical, lexicographic, etc.

- We will see two types
  - Bubble sort
  - Insertion sort

Algorithm 4: Bubble sort

- One of the most simple sorting algorithms
  - Also one of the least efficient
- It takes successive elements and "bubbles" them up the list

```
procedure bubble_sort ($a_1$, $a_2$, …, $a_n$)
for $i$ := 1 to $n$-1
    for $j$ := 1 to $n$-$i$
      if $a_j > a_j$+1
                then interchange $a_j$ and $a_j$+1
{ $a_1$, …, $a_n$ are in increasing order }
```

Algorithm 4: Bubble sort

- An example using physical objects…

Bubble sort running time

- Bubble sort algorithm:

```
for $i$ := 1 to $n$-1
    for $j$ := 1 to $n$-$i$
        if $a_j > a_j$+1
            then interchange $a_j$ and $a_j$+1
```

- Outer for loop does n-1 iterations
- Inner for loop does
  - n-1 iterations the first time
  - n-2 iterations the second time
  - …
  - 1 iteration the last time
- Total: $(n-1) + (n-2) + (n-3) + \ldots + 2 + 1 = (n^2-n)/2$
  - We can say that's "about" n^2 time

Algorithm 5: Insertion sort

- Another simple (and inefficient) algorithm
- It starts with a list with one element, and inserts new elements into their proper place in the sorted part of the list

```
procedure insertion_sort ($a_1, a_2, …, a_n$)
    for $j$ := 2 to $n$                take successive elements in the list
    begin
        $i$ := 1
        while $a_j > a_i$              find where that element should be in the
                $i := i + 1$           sorted portion of the list
        $m := a_j$
        for $k$ := 0 to $j-i-1$        move all elements in the sorted portion of
                $a_{j-k} := a_{j-k-1}$ the list that are greater than the current
        $a_i := m$                     element up by one
    end { $a_1, a_2, …, a_n$ are sorted }
```

put the current element into it's proper place in the sorted portion of the list

Insertion sort running time

```
for $j$ := 2 to $n$ begin
    $i$ := 1
    while $a_j > a_i$
        $i$ := $i$ + 1
    $m$ := $a_j$
    for $k$ := 0 to $j-i-1$
        $a_{j-k}$ := $a_{j-k-1}$
    $a_i$ := $m$
end { $a_1, a_2, \ldots, a_n$ are sorted }
```

- Outer for loop runs n-1 times
- In the inner for loop:
  - Worst case is when the while keeps i at 1, and the for loop runs lots of times
  - If i is 1, the inner for loop runs 1 time (k goes from 0 to 0) on the first iteration, 1 time on the second, up to n-2 times on the last iteration
- Total is $1 + 2 + \ldots + n-2 = (n-1)(n-2)/2$
  - We can say that's "about" n^2 time

Comparison of running times

- Searches
  - Linear: n steps
  - Binary: $\log_2 n$ steps
  - Binary search is about as fast as you can get
- Sorts
  - Bubble: n^2 steps
  - Insertion: n^2 steps
  - There are other, more efficient, sorting techniques
    - In principle, the fastest are heap sort, quick sort, and merge sort
    - These each take take $n * \log_2 n$ steps
    - In practice, quick sort is the fastest, followed by merge sort