Ambo University @Woliso Campus

School of Technology and Informatics

Computer Science Department

Analysis of Algorithm Course Full hondout

Program: **Extension** Batch: **4th** Semister: **II**

Course Code:**CoSc 3131**

Prepared and Compiled by: ***Yoobsan Bechera (BSc)***

May/25/2020, AU, Oromia, Ethiopia

Table of Contents

Chapter 1 1

Introduction to Analysis of Algorithm 1

Introduction 1

Properties of an algorithm 1

Euclid's algorithm for computing gcd(m,n) 4

Alternative Euclid's algorithm 4

Algorithm Evaluation 5

Correct Algorithm 5

How to prove the correctness of an algorithm? 5

The sorting problem 6

Insertion Sort Algorithm 6

Insertion sort 6

Proof correctness of the algorithm 7

Review Exercise 8

Chapter 2 8

Why we study algorithm? 8

Algorithm Design & Analysis Process 9

Analysis of Algorithm 9

Analysis of Insertion Sort 10

Best Case Analysis of Insertion Sort 11

Worst Case Analysis of Insertion Sort 11

Average Case Analysis of Insertion Sort 11

Running time of an algorithm 12

How to analyze 'for loops' 13

How to Analyze 'Nested for loops' 13

How to Analyze 'Consecutive statements' 13

How to Analyze 'Conditionals' 14

Asymptotic analysis 15

Asymptotic notations: Big-Oh (O) 15

Order of growth of functions 16
Asymptotic notations: Big-Omega (Ω) 17
Asymptotic notations: Theta (Θ) 17
Algorithmic efficiency 18
Recursive Algorithm 18
The factorial function 19
Analysis of Recursive algorithm using substitution: 19
Recursive algorithm for reversing an Array 20
Review Exercise 21
Chapter 3 21
Disjoint sets and Graph 21
Disjoint sets 21
Algorithm for Disjoint set 22
Graph 26
Graph Interpretations 26
Graph Definitions and Types 26
Adjacency multilist 29
Graph Traversals 30
Depth-First Traversal 31
Breadth-First Search (BFS) 32
Time complexity 33
Review Exercise 34
Chapter 4 34
The divide-and-conquer strategy 34
Divide-and-Conquer Strategy 35
Divide-and-Conquer Technique 36
Solving Recurrence Relation 36
Binary Recursive Method 38
Binary search 38
Binary Search Recursive Algorithm 39
Binary Search Iterative Algorithm 40

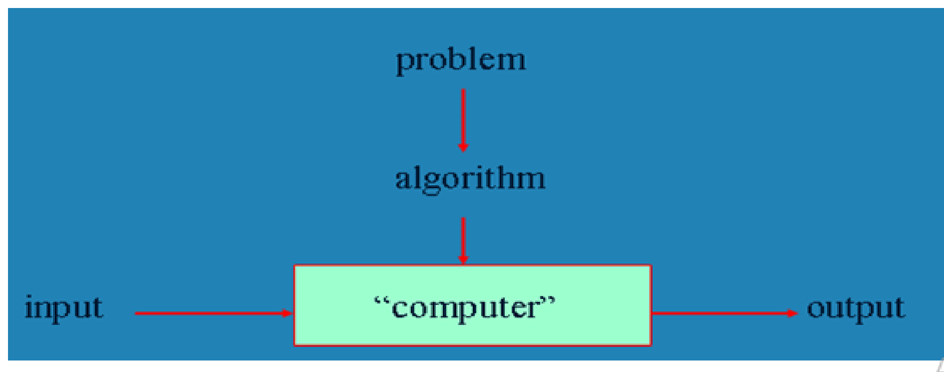
Chapter 5 46
Greedy Algorithms 46
The greedy method 46
Feasible vs. optimal solution 46
Greedy Choice Property 47
Knapsack problem 48
Analysis of Knapsack problem 49
Minimum Spanning Trees 50
Prim’s algorithm 51
Kruskal’s algorithm 52
Correctness of Kruskal 54
Dijkstra’s shortest-path algorithm 54
Huffman Coding 55
Chapter 6 56
Dynamic programming 56
Divide & Conquer vs. Dynamic Programming 56
Greedy vs. Dynamic Programming 56
Dynamic programming approaches 57
0/1 Knapsack: Brute-force approach 58
The shortest path in multistage graphs 60
String editing 62
Chapter 7 63
Backtracking Algorithm 63
Backtracking 63
Backtracking approach 63
The Queens Problem 63
4 - Queens 64
Traveling Salesperson Problem (TSP) 65
Branch and Bound Algorithm 66

Chapter 1

Introduction to Analysis of Algorithm

Introduction
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. The definition can be illustrated by the following diagram.



- An algorithm is a **clearly** specified set of simple instructions to be followed to solve a problem. Any **well-defined** computational procedure that *takes some value (or set of values) as* an **input** and *produces some value (or set of values*) as an **output**. A **sequence** of computational steps that *transforms* the **input** into the **output**

- A set of **well-defined**, **finite** rules used for problem solving. A **finite** set of instructions that, if followed, accomplish a particular task. It is a **precise**, systematic method for producing a specified **result.**

Properties of an algorithm

- From the above definitions, algorithm has the following properties**: Sequence, Unambiguous, Input, Output, Finite**

Sequence

- It is a step-by-step procedure for solving a given problem
- Every algorithm should have a beginning (start) and a halt (end) step
- The first step (start step) and last step (halt step) must be clearly noted
- Between the two every step should have preceding and succeeding steps

- That is, each step must have a uniquely defined preceding and succeeding step

Unambiguous

- Define rigorously the sequence of operations performed for transforming the inputs into the outputs
- No ambiguous statements are allowed: Each step of an algorithm must be clearly and precisely defined, having one and only one interpretation.
- At each point in computation, one should be able to tell exactly what will happen next
- Algorithms must specify every step. It must be composed of concrete steps
- Every detail of each step must be spelled out, including how to handle errors
- This ensures that if the algorithm is performed at different times or by different systems using the same data, the output will be the same.

Input specified

- The inputs are the data that will be transformed during the computation to produce the output
- An input to an algorithm specifies an instance of the problem the algorithm solves
- Every algorithm should have a specified number (zero or more) input values (or quantities) which are externally supplied
  - ✓ We must specify **the type of data** and **the amount of data**
- Note that, correct algorithm is not one that works most of the time but one that works correctly for all **legitimate inputs**

Output specified

- The output is the data resulting from the computation
  - ✓ It is the intended result
- Every algorithm should have **one or a sequence of output values**
  - ✓ There must be one or more result values
- A possible output for some computations is a statement that there can be no output, i.e., no solution is possible
- The algorithm can be proved to produce the correct output given a valid input.

Finiteness: It must terminate

- Every valid algorithm must complete or terminate after a finite number of steps.
- If you trace out the instructions of an algorithm, then **for all cases the algorithm must terminate** after a finite number of steps.
- It must eventually stop either with the right output or with a statement that no solution is possible.
- **Finiteness** is an issue for computer algorithms because
  - ✓ Computer algorithms often repeat instructions
  - ✓ If the algorithm doesn't specify when to stop, the computer will continue to repeat the instructions forever.

Middle-school algorithm for computing gcd(m,n)

Step1: Find the prime factors of m

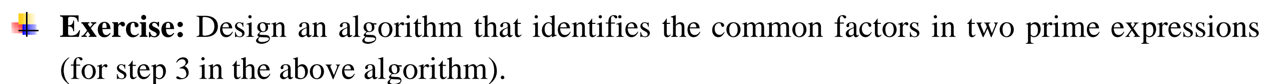
Step2: Find the prime factors of n

Step3: Identify all the common factors in the two prime expressions found in Step 1 and Step 2.

- If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min(p_m,p_n)$ times.

Step4: Compute the product of all the common divisors as the GCD for the given inputs, m and n

Step5: Return this value as GCD of m and n


Exercise: Design an algorithm that identifies the common factors in two prime expressions (for step 3 in the above algorithm).

Consecutive integer checking algorithm for computing gcd (m, n)

Step1: k = min (m, n) - find the minimum of m & n

Step2: Divide m by k.

- If the remainder of the division is 0, go to Step 3;
- otherwise, go to Step 4

Step3: Divide n by k.

- If the remainder of the division is 0, proceed to Step 5
- otherwise, proceed to Step 4

Step4: Decrease the value of k and go to step 2

Step5: Return the value of k as gcd of m and n and stop

Euclid's algorithm for computing gcd(m,n)

Step1: if n = 0

- return the value of m as the answer and stop
- Otherwise proceed to step 2.

Step2: divide m by n and assign the remainder to r.

Step3: assign the value of n to m and the value of r to n.

Step4: go to Step 1

Alternative Euclid's algorithm

- We can also express the same algorithm in a better way as follows:

Algorithm Euclid (m,n)

//Input: two nonnegative, not-both-zero integers' m & n

//Output: gcd of m and n

while n $\neq$ 0 do

r ← m mod n

m ← n

n ← r

end while

return m

end algorithm

<u>Exercise</u>

1) Write an algorithm (that satisfies the properties) for computing gcd(m,n) using either **Middle school** or **Consecutive integer checking** algorithm

2) Write a program to implement one of the above algorithms for computing gcd(m,n) and test your program using inputs

   a. gcd(60,24)

   b. gcd(31415,14142)

   c. gcd(60,0)

3) Justify why the algorithm will eventually stop in (i) consecutive integer checking algorithm (ii) Euclid's algorithm for computing gcd(m,n)

Algorithm Evaluation

- Which one is an efficient algorithm? Compare consecutive integer checking and Euclid algorithm. How much iteration required solving gcd (60, 24)? What about gcd(31415,14142)

- Consecutive integer checking procedure is much more complex and slower than Euclid algorithms. Euclid's algorithm is less complex and faster to compute. Which one is the correct algorithm? How can we know?

Correct Algorithm

- A correct algorithm solves the given computational problem. If the algorithm is not doing what it is supposed to do, it is worthless. An algorithm is said to be correct if, for every input instance, **it halts with the correct output**

- An **incorrect** algorithm might not halt at all on some input instances, or might halt with a wrong answer. In order to show that an algorithm is **incorrect**, you need just one instance of its input for which the algorithm fails

How to prove the correctness of an algorithm?

- Common techniques are by mathematical **induction & contradiction**

Proof by Induction:

- **The induction base**: is the proof that the statement is true for initial value (e.g. n =1)

   - **The induction hypothesis**: is the assumption that the statement is true for an arbitrary values 1, 2, …, n

   - **The induction step**: is the proof that if the statement is true for n, it must be true for n+1

   - **Example 1**: show that, for all positive integers n,

     Answer: $$1+2+...+n=\frac{n(n+1)}{2}$$

Exercise: proof by induction that:

$$r^n + r^{n-1} + \dots + r^1 + r^0 = \frac{r^{n+1} - 1}{r - 1}$$

The sorting problem

- One might need to sort a sequence of numbers into non-increasing order or into non-decreasing order.

- Statement of the sorting problem:

  Input: a sequence of n number $\langle a_1, a_2, \dots, a_n \rangle$

  Output: a permutation (reordering) $\langle a_1', a_2', \dots, a_n' \rangle$ such that $a_1' \leq a_2' \leq \dots \leq a_n'$.

Example:

- Given an input sequence <31, 41, 59, 26, 41, 58>, a sorting algorithm that arranges in non-decreasing order returns as an output the sequence <26, 31, 41, 41, 58, 59>

Insertion Sort Algorithm

- It is an efficient algorithm for sorting a smaller number of elements. It is similar to sorting a hand of playing cards.

- **Idea**: Every time, take one card and insert the card to correct position in already sorted cards.

- We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table & insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left as shown in the figure.

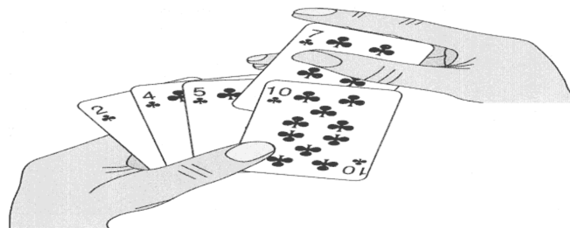


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion sort

- **Principle**: starting from the beginning sort each member of the input by putting one by one in its proper position

- **Input**: a sequence of n number <a_1, a_2, …,a_n>. We are required to sort the array in increasing order.
- **Output**: a permutation (reordering) <a_1', a_2', …,a_n'> such that $a_1' \le a_2' \le \ldots \le a_n'$.

  Example: show how the insertion sort algorithm sorts in increasing order the sequence

  A = <5, 2, 4, 6, 1, 3>

- An algorithm to sort in non-decreasing order

Algorithm INSERTION-SORT(A, n)

//input: array of A[1..n]

//output: sorted array of A

for *j* = 2 to length[A] **do**

key ← A[*j*]

//insert A[*j*] to sorted sequence A[1..*j*-1]

i ← *j*-1

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

A[*i*+1] ← A[*i*] //move A[*i*] one position right

i ← *i*-1

end while

A[*i*+1] ← *key*

end for

end algorithm

Proof correctness of the algorithm

- The index j indicates the "current value" being inserted into the sorted array.
  - Array element A[1..j-1] constitute the currently sorted element.
  - Elements A[j+1...n] correspond to the other values still not sorted
- At each iteration of the outer for loop, the element A[j] is picked out of the array (line 2). Then, starting in position j-1, elements are successively moved one position to the right until

the proper position for A[j] is found (while loop from lines 4-7), at which point it is inserted (line 8)

Proof by induction

Example to finding the maximum element problem

- The Input is an array A storing n elements and the output is the maximum one in A. Given array A= [31, 41, 26, 41, 58], max algorithm returns 58.

```
        Algorithm findMax(A, n)
        //Input: An array A[1..n].
        //Output: The maximum element in A.
        currentMax ← A[0]
        for  i← 1 to n -1 do
            if currentMax < A[i] then
                currentMax ← A[i]
        end for
        return currentMax
    end algorithm
```

Review Exercise

1) Write an algorithm for finding minimum element(s) from the given sequence.
   a) Is your algorithm **correct** and **efficient**?
   b) For your answer of "a" above **justify your reason.**

2) We can write an algorithm either by flow chart or pseudo code. Write an algorithm which finds the average of n numbers by both ways.

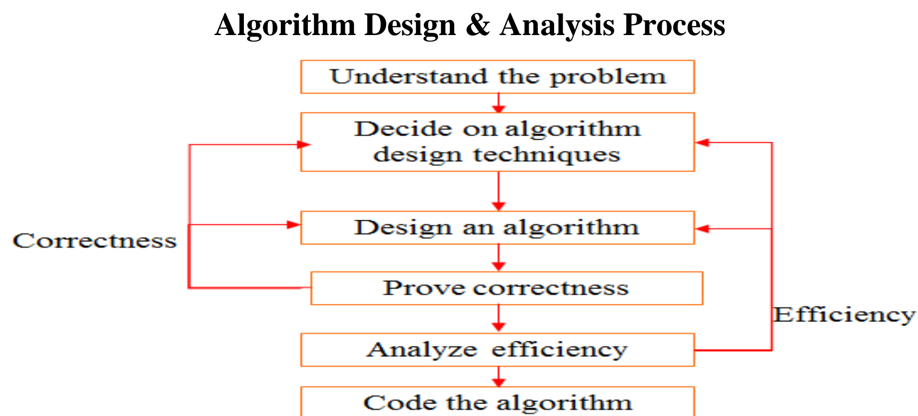
Chapter 2

Why we study algorithm?

- Suppose computers are infinitely fast and computer memory was free. Would you have any reason to study algorithm?

- Yes, because we want to demonstrate that the algorithm is correct; it terminates with the intended solution for all inputs given.
- However, the reality shows the following facts:
    - Computers may be fast, but they are not infinitely fast
    - Memory may be cheap, but it is not free
    - Computing time and resources are therefore a **bounded resources**

Algorithm Design & Analysis Process



Analysis of Algorithm

- As you may have noted, there are often multiple algorithms one can use to solve the same problem.
    - In solving GCD problem, we can use techniques such as middle school, consecutive integer checking or Euclid
    - In searching from a sequence of list, one can use linear search, binary search…
    - You can come up with your own variants.

How do we choose which algorithm is the best?

- The fastest/most efficient algorithm.
- The one that uses the fewest resources.
- The clearest.
- The shortest, ...

- Analysis of algorithm is the analysis of resource usage of a given algorithm. It means predicting the resources that the algorithm requires. The main resources are **running time**

and **memory usage**. An algorithm that solves a problem but requires **a year** and **GBs of main memory** is hardly of any use.

- The **objective** of algorithm analysis is:
  - to measure the resources (e.g., time, space) requirements of an algorithm so as to determine how quickly (with less memory) an algorithm executes in practice.
- An algorithm should make **efficient** use of computer resources. Most frequently, we look at efficiency:
  - how long does the algorithm take to run
  - What is the best way to represent the running time of an algorithm?

Efficiency

- An algorithm must solve a problem with the least amount of computational resources such as time and space. An algorithm should run as fast as possible using as little memory as possible.
- Two types of algorithmic efficiency evaluation:

  Time efficiency - indicates how fast the algorithm runs

  Space efficiency - indicates how much memory the algorithm needs

What to analyze?

- To keep things simple, we will concentrate on the running time of algorithms and will not look at the space (the amount of memory) needed or required.
- So, efficiency considerations of algorithm usually focus on the amount of time elapsed (called **running time of an algorithm**) when processing data.

Analysis of Insertion Sort

algorithm INSERTION-SORT(A)	***cost***	***times***
for $j \leftarrow 2$ to length[A] **do**	c_1	n
$key \leftarrow A[j]$	c_2	n-1
$i \leftarrow j-1$	c_3	n-1
while $i>0$ and $A[i]>key$ **do**	c_4	$\sum_{j=2}^{n} t_j$
		$\sum_{j=2}^{n} t_j - 1$

A[i+1] ← A[i] c_5

i ← i-1 c_6 $\sum_{j=2}^{n} t_j - 1$

A[i+1] ← key c_7 n-1

- (t_j is the number of times the while loop test in line 4 is executed for that value of j)

- The running time, T(n), of the insertion algorithm is the sum of running times for each statement executed, i.e.: **$= c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=2}^{n} t_j + c_5\sum_{j=2}^{n} (t_j-1) + c_6\sum_{j=2}^{n}(t_j-1) + c_7(n-1)$**

Best Case Analysis of Insertion Sort

- Occurs if the array contains already sorted values. For each j = 2, 3, 4… n, we then find that **A[i] ≤ key** in line 4 when i has its initial value of j – 1. Thus t_j=1 for j = 2, 3,…, n, and line 5 and 6 will be executed 0 times

- The best case running time is

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

- This running time can be expressed as ***an + b*** for constants a and b that depends on the statement cost c_i;

  - it is thus a **linear function** of n

Worst Case Analysis of Insertion Sort

- Occurs if the array contains values that are in reverse sorted order, which is in decreasing order. We must compare each element A[j] with each element in the entire sorted sub array A[1..j-1]. So, t_j = j for j = 2, 3… n.

- Therefore the worst case running time of INSERTION-SORT is $T(n)$. This worst case running time can be expressed as $\boldsymbol{an^2 + bn + c}$ for constants a, b, c, it is thus a **quadratic function** on n

Average Case Analysis of Insertion Sort

- Suppose that we randomly choose **n** numbers and apply insertion sort. How long does it take to determine where in sub array A [1...j-1] to insert the element A[j]?

  - On average, half the elements in A [1...j-1] are less than A[j], and half the elements are greater. Therefore, we check half the sub array A [1...j-1], so $t_j = j/2$ and $T(n)$ will still be in the order of n^2.

- This average case running time can then be expressed as **quadratic function, $an^2 + bn + c$** for constants a, b, c, which is the same as worst case.
- In summary, the running time of insertion sort for
    - ✓ Best case: an – b
    - ✓ Worst case: $an^2 + bn - c$
    - ✓ Average case: $an^2 + bn - c$

Running time of an algorithm

- Several factors affect the **running time** of an algorithm:
    - ✓ Compiler used (quality of compiler)
    - ✓ Computer used (speed of machine): The same operation may take different times on different machines.
    - ✓ The algorithm used (or quality of source code): Not all operations take the same time. For example, addition is typically quicker than multiplication, and integer addition is typically quicker than floating point addition.
    - ✓ The input to the algorithm (size and characteristics of input): Different inputs lead to different running times.
        - The first two are beyond the scope of theoretical model. The last two are the main factors that we deal
    - For most algorithms, the running time depends on:
        - **characteristics of the input:** An already sorted sequence is easier to sort than unsorted one for sorting algorithms
        - **size of the input:** Short sequences are easier to sort than long ones
    - Thus, the running time of most algorithms varies with the **characteristics** and **size of input.**
- Running time is expressed as T(n) for some function T of input size n.

Example: Find the running time T(n) for the algorithm

```
int x = 0

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

```
x++;
```

T(n) = 0.6*n + 3 = an + b

- To find running time, T(n), we have two options:

  1. Count the number of times each of the algorithm's step-by-step instructions are executed. This method is excessively difficult and usually unnecessary

  2. Count the number of times the most important operations of the algorithm is executed. Which are the basic operations for an algorithm? The most time consuming operations are found inside the inner most loop.

How to analyze 'for loops'

- In general, a ***for*** loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the '***for'*** loop.

```
for i = 1 to N do
  sum = sum + i;
end for loop
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence *N* additions in total

How to Analyze 'Nested for loops'

- Nested for loops translate into multiple summations, one for each for loop.

```
for i = 1 to N do
    for j = 1 to M do
        sum = sum + i + j ;
    end inner for
end outer for
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Suppose again we count the number of additions. The outer summation is for the outer for loop.

How to Analyze 'Consecutive statements'

- Add the running times of the separate blocks of your algorithm.

```
for i = 1 to N do
    sum = sum + i;
        end for
 for i = 1 to N do
for j = 1 to N do
        sum = sum + i + j;
            end inner for
                end outer for
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

$T(n) = n + n^2$

How to Analyze 'Conditionals'

- if (test) s1 else s2:
  - Compute the maximum of the running time for s1 and s2.

```
            if (test == 1)
    for (int i = 1; i <= N; i++)
            sum = sum + 1;
                end for
          else
    for (int i=1;i<=N;i++)
        for (int j = 1; j <= N; j++)
                sum = sum + i + j;
                    end inner for
                        end outer for
                            end if
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) = \max\left(N, 2N^2\right) = 2N^2$$

Asymptotic analysis

- There are five notations used to describe a running time function: Big-Oh, Big-Omega, Theta, Little-o, little-omega. Demonstrating that a function T(n) is in big-O (or others) of a function f(n) requires that we find specific constants C and n_o for which the inequality holds.
- The following points are facts that can be used for efficiency comparison.

$1 \leq n$ for all $n \geq 1$	$2^n \leq n!$ for all $n \geq 4$
$n \leq n^2$ for all $n \geq 1$	$\log_2^n \leq n$ for all $n \geq 2$
$n \leq n\log_2^n$ for all $n \geq 2$	

Asymptotic notations: Big-Oh (O)

Definition:

- Let f(n) and g(n) be functions mapping nonnegative integers to real numbers.
- A function f(n) = O(g(n)), if there is some positive constant c > 0 and a non-negative integer $n_o \geq 1$ such that

$$\mathbf{f(n) \leq c.g(n) \text{ for all } n \geq n_o}$$

- Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n
- An *upper bound* is the best algorithmic solution that has been found for a problem (**"what is the best thing that we know we can do?"**)
- In simple words, f(n) = O(g(n)) means that the ***growth rate of f(n)*** is less than or equal to g(n). The statement f(n) = O(g(n)) states only that c.g(n) is un upper bound on the value of f(n) for all n, $n \geq n_0$

Big-Oh theorems

- **Theorem 1**: If k is a constant, then k is O(1)

  Example: $f(n) = 2^{100} = O(1)$

- **Theorem 2:** If f(n) is a polynomial of degree k, then

  $f(n) = O(n^k)$

- If $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k$, where a_i and k are constants, then f(n) is in $O(n^k)$
- Polynomial's growth rate is determined by the leading term

  Example: $f(n) = 7n^4 + 3n^2 + 5n + 1000$ is $O(n^4)$

➢ ***Theorem* 3:** Constant factors may be ignored

If g(n) is in O(f(n)), then k * g(n) is O(f(n)), k >0

Example:

✓ $T(n) = 7n^4 + 3n^2 + 5n + 1000$ is $O(n^4)$

✓ $T(n) = 28n^4 + 12n^2 + 20n + 4000$ is $O(n^4)$

➢ **Theorem 4 (Transitivity)**

- If T(n) is O(f(n))and f(n) is O(g(n)), then T(n) is O(g(n)).

➢ **Theorem 5**

- If T(n) is in O(f(n)) and g(n) is in O(h(n)), then T(n) + g(n) is in O(T(n) + g(n))

➢ **Theorem 6**

- If T(n) is in O(f(n)) and g(n) is in O(h(n)), then T(n) . g(n) is in O(T(n) . g(n))
- product of upper bounds is upper bound for the product

➢ **Theorem 7**

- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))$

Order of growth of functions

Typical orders: Here is a table of some typical cases. It shows that the typical order is:

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$

N	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1024	1	10	1,024	10,240	1,048,576	1,073,741,824

<u>Example: Big-Oh (O)</u>

- Find O(f(n) for the given functions:

  a) $f(n) = 2n + 6$

  b) $f(n) = \mathbf{13n^3} + 42n^2 + 2n \log n$

  c) If $f(n) = 3n^2 + 4n + 1$ then show that $f(n) = O(n^2)$

  d) If $f(n) = 10n + 5$ and $g(n) = n$, then show that f(n) is O(g(n))

Asymptotic notations: Big-Omega (Ω)

Definition:

- Let f(n) and g(n) be functions mapping nonnegative integers to real numbers.
- A function $f(n) = \Omega(g(n))$, if there is some positive constant $c > 0$ and a negative integer $n_o \geq 1$ such that

$$\mathbf{f(n) \geq c.g(n) \text{ for all } n \geq n_o}$$

- The statement $f(n) = \Omega(g(n))$ states only that c.g(n) is a lower bound on the value of f(n) for all n, $n \geq n_0$. In simple terms, $f(n) = \Omega(g(n))$ means that the growth rate of f(n) is greater than or equal to g(n)

Big-Omega- Example

- Show that the function $T(n) = 5n^2 - 64n + 256 = \Omega(n^2)$
  - We need to show that for non-negative integer n_0 and a constant $c > 0$, $T(n) \geq c.n^2$ for all integers $n \geq n_0$
  - we have that for c=1 and $n_0 = 0$, $T(n) \geq cn^2$ for all integers $n \geq n_0$
  - What if $c = 2$ and $n_0 = 16$?
  - Show if $f(n) = 10n^2 + 4n + 2$ and $g(n) = n^2$, then $f(n) = \Omega(n^2)$
- Show that $3n^2 + 5 \neq \Omega(n^3)$

Asymptotic notations: Theta (Θ)

Definition:

- Let f(n) and g(n) be functions mapping nonnegative integers to real numbers.
- A function $f(n) = \mathbf{\Theta}(g(n))$, if there exist some positive constant c_1 and c_2 and a negative integer constant $n_o \geq 1$ such that $c_1\mathbf{.g(n) \leq f(n) \leq c_2.g(n) \text{ for all } n \geq n_o}$

- The Theta notation is used when the function f can be bounded both from above and below by the **same function.** When we write $f(n) = \boldsymbol{\Theta}(g(n))$, we mean that f lies between c_1 times the function g and c_2 times the function g except possibly when n is smaller than n_0
- Another way to view the θ-notation is that the function:
    - $f(n) = \theta(n)$ if and only if
    - $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Asymptotic Tightness

- The theta notation is more precise than both the Big-O and Big-Ω notations. The function $f(n) = \boldsymbol{\Theta}(g(n))$ iff g(n) is both an upper and lower bounds on f(n). Big-Oh does not have to be asymptotically tight:
    - $f(n) = ½n$ is O(n) with c=1, n_0=1, but is also in $O(n^{100})$…
- Big-Ω isn't tight either
    - $f(n) = n^5$ is Ω(n) with c=1, $n_0 = 1$…
    - Theta (Θ) is tight…
    - f(n) must be in same growth classes to meet definition.
    - Can you prove this assertion? Prove that $f(n)=3n^3+2n^2+1$ is $\Theta(n^3)$.
    - Show that f(n) is $O(n^3)$, and also, f(n) is $\Omega(n^3)$

Algorithmic efficiency

- What is the best, worst and average case time complexity of the following algorithms:
    - Insertion sort?
    - Linear search?

Recursive Algorithm

- **Recursive algorithm**: calls itself again and again until exit condition is satisfied

To define a recursive algorithm:

Specify a base case: there should be one or more base cases

- Begin by testing for a set of base cases.

- Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.

Recur once: rule to perform a single recursive call each time

- Define a recursive call that makes one recursive call each time and progress towards a base case.

Classic example: give recursive definition of the factorial function:

- Recursive definition of the factorial: n! = 1· 2· 3· ··· · (n-1)· n

  Base case: f(n) = 1 if n = 0

  Recursive case: f(n) = n . f_{n-1} if n > 0

The factorial function

Recursive algorithm:

```
algorithm recursiveFactorial(n)
  if (n = 0) then
   return 1;          // base case
    else
   return n * recursiveFactorial(n- 1);    // recursive case
     endif
 end algorithm
```

Analysis of Recursive algorithm using substitution:

- **Backward substitution** starts with some *n* and work backward, substituting repeatedly for each occurrence of the function T (n), until a clear pattern emerges; then substitute for the base case.

$$T(n) = \begin{cases} 2 & \text{if n} = 0 \\ T(n-1) + 2 & \text{if n} > 0 \end{cases}$$

Running time T (n) = 2n + 2

Example of Recursion

Problem1: Write a recursive function to find the sum of the first n integers A[1…n] and output the sum. Example: given k = 3, we return sum = A[1] + A[2] + A[3], given k = n, we return A[1] + A[2] + … + A[n]. How can you define the problem in terms of a smaller problem of the same type?

$$1 + 2 + \ldots + n = [1 + 2 + \ldots + (n - 1)] + n$$

for n > 1, f(n) = f(n.1) + n

- How does each recursive call diminish the size of the problem? It reduces by 1 the number of values to be summed.
- What instance of the problem can serve as the base case? **n = 1.**
- As the problem size diminishes, will you reach this base case? **Yes**, as long as n is nonnegative. Therefore the statement **"n >= 1"** needs to be a precondition

Problem2: Write a recursive function to find the sum of the first n integers A[1…n] and output the sum

algorithm LinearSum(*A, n*)

// ***Input:*** an array *A with* n elements

// ***Output:*** The sum of the first *n* integers in *A*

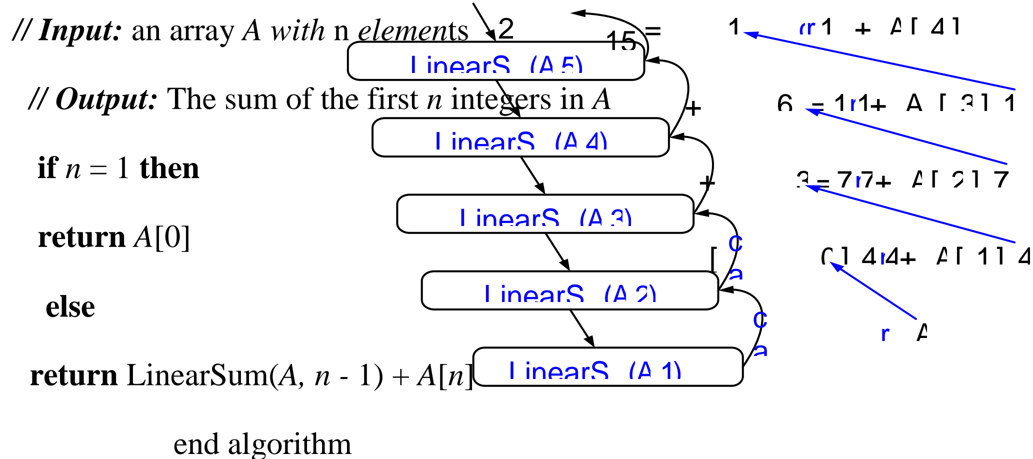
if *n* = 1 **then**

return *A*[0]

else

return LinearSum(*A, n* - 1) + *A*[*n*]

end algorithm



Recursive algorithm for reversing an Array

Algorithm ReverseArray(*A, i, j*)

Input: An array *A* and indices *i = 1* and *j = n*

Output: The reversal of the elements in *A*

if *i < j* **then**

Swap *A*[*i*] and *A*[*j*]

ReverseArray(*A, i* + 1, *j* - 1)

return

end algorithm

Algorithm IterativeReverseArray(*A, i, j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of elements in *A* starting at index *i* and ending at *j*

while $i < j$ **do**

swap ***A*[*i*] and *A*[*j*]**

***i* = *i* + 1**

***j* = *j* - 1**

return

end algorithm

Review Exercise

1) Write (i) an algorithm and (ii) a code that scans through the given sequence of inputs, A = <a_1, a_2… a_n>, finding for **one or more minimum value(s).**

2) Write algorithm and code for linear search, which scans through the given sequence of inputs, A = <a_1, a_2… a_n>, looking for **Key**.

   The output is either

   - One or more index i (the position of all values if **key** = A[i]) or
   - The special message "Not Found" if **Key** does not appear in the list.

Chapter 3

Disjoint sets and Graph

Disjoint sets

- Many times the efficiency of an algorithm depends on the data structures used in the algorithm. Choosing suitable data structure in solving a problem can **reduce the time of execution, the time to implement the algorithm** and **the amount of memory used**.

The problem

- Let's consider the following problem: In a room are N persons. Thus, two persons are friends if they are directly or indirectly friends. If A is a friend with B, and B is a friend with C, then A is a friend of C too. A group of friends is a group of persons where any two persons in the

group are friends. Given the list of persons that are directly friends, find the number of groups of friends and the number of persons in each group. For example, N = 9 & the list of friends are: 1-2, 3-4, 5-1, 6-4, 7-9 and 8-2.

- Some applications involve grouping **n** distinct elements into a collection of disjoint sets.
  - Two sets are disjoint if their intersection is NULL: $S_1 \Lambda S_2 = Ø$
- In disjoint set data structure:
  - Each set is represented by a tree, so that each element points to a parent in the tree.
  - Every set contains a representative (root), which is also one of the member of the set

Applications

- Maintain the connected components of a graph as new vertices and edges are added.
- To solve the problem of spanning tree (Kruskal algorithm).
- In both applications, we can use a disjoint-set data structure, where we keep a set for each connected component, containing that component's vertices.

Three operations of disjoint sets are:

CREATE (x):

- Creates a new set {x} containing the single element x.
- The element x must not appear in any other set in our collection.
- The root/leader of the new set is obviously x.

UNION(*x, y*):

- Combines/merges two disjoint sets containing root *x* and root *y* into one set.
- Replaces two sets, A and B with their union A U B.

FIND(*x*):

- Finds in which set a given node x belongs to and returns the root node of the set containing the element *x.*

Algorithm for Disjoint set

algorithm DisjointSet(x, n)

//input x[1,2, …,n]

```
FOR i = 1 to n DO
        CREATE(x_i)
end for
FOR (each pair of friends (x_i, x_j) ) DO
        IF (FIND(x_i) != FIND(x_j)) THEN
                UNION(x_i, x_j)
        end if
end for
end algorithm
```

- FIND() operation check if the two pairs, x_i & x_j are in the same group or not, before merging them using UNION() operation

Algorithms: Create, Union & Find operations

```
procedure CREATE(x)
  parent(x) = -1 //some negative number
    end
procedure UNION(x,y)
    parent(x) = y
      end
procedure FIND(x)
    y = x
  while parent(y) > 0 do
        y = parent(y)
end while
return y
end
```

- Determine time complexity of the above algorithms?

Improving UNION and FIND operations

- Analyze the total time required for performing the following operations:

  UNION(1,2), UNION(2,3), …, UNION(n-1,n), and FIND(1), FIND(2), …, FIND(n)

- There is a need to enhance the efficiency of UNION and FIND operations. Notice that the time to do a FIND operation on an element corresponds to its depth in the tree. Can we improve the performance of UNION and FIND? Improve means decrease the height of the trees. Hence our goal is to keep the trees as short as possible.

- Two heuristics for keeping the height of the disjoint trees short are **UNION by rank** and **Path Compression**

UNION by rank: ensures that when we combine two trees, we try to keep the overall depth of the resulting tree small.

  - If x and y are roots of two distinct trees, this technique makes the root of the smaller tree a child of the root of the larger tree.
  - Union by rank avoids creation of degenerate trees

Path compression: collapses all nodes to point to root node.

  - Is used during FIND operation so as to make each node on the find path point directly to the root.

UNION by rank

- Balances the height of a tree. The idea is that the rank of the root is associated with the depth of the tree so as to keep the depth small. **Weighting rule for UNION(x, y):** If the number of nodes in tree with root x is less than the number in tree with root y, then make y the parent of x; otherwise, make x the parent of y.

```
procedure UNION(x,y)

    z = parent(x) + parent(y)

    if parent(x) > parent(y) then

        parent(x) = y

        parent(y) = z

    else
```

```
            parent(y) = x
            parent(x) = z
      end if
end UNION
```

- To implement weighting rule there is a need to keep the number of nodes in every tree in the parent field of the root as –ve numbers.

Path Compression

- Reduces the complexity of FIND algorithm. This is done by collapsing nodes in a tree. **Collapsing rule:** If x is a node on the path from y to its root and parent[y] ≠ root[y], then set parent[x] to root[y]. The idea is that, once we perform a FIND on some element, we should adjust its parent pointer so that it points directly to the root; that way, if we ever do another FIND on it, we start out much closer to the root.

```
function FIND(x)
    y = x
  while parent(y) > 0 do
        y = parent(y) //Find root
    z = x
    while z ≠ y do //collapse nodes from z to root y
            t = parent(z)
            parent(z) = y
            z = t
        return y
    end FIND
```

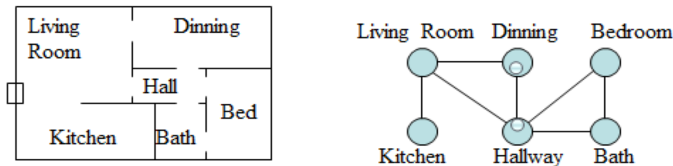
Exercise:

Given $S_1 = \{3,4,6\}$, $S_2 = \{1,7,8,9\}$ & $S_3 = \{2,5,10\}$, represent them using disjoint set tree?

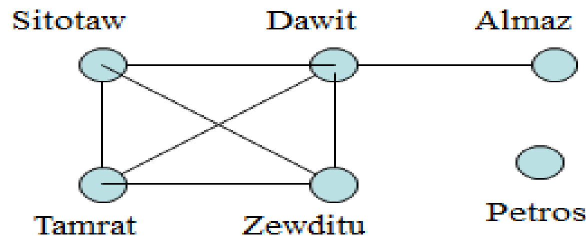
Graph

Graph Interpretations

- The use of a graph may be an easy simplification for a problem.
- Example: **Apartment Blueprint**: The vertices could represent rooms in a house, and the edges could indicate which of those rooms are connected to each other.



- **Friendship Graphs:** Each vertex represents a person, and each edge indicates that the two people are friends.



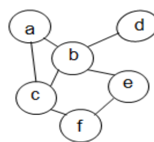
Graph Definitions and Types

- A **graph** *G* is a pair, G = (*V*, *E*), where *V* is the set of **vertices** and *E is* the set of **edges** that link together the vertices. The **degree** of a vertex is determined by the number of distinct edges that are incident to it.

Types of Graphs

Simple Graphs

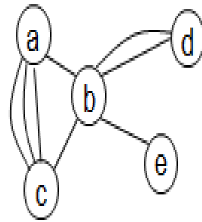
A simple graph G is a pair (V, E), where V is a set of vertices (representing the objects) and E is a set of edges, where each edge in E is a set of 1 or 2 vertices (representing the links between vertices).



G = (V,E), where
V = {a,b,c,d,e,f}
E = {{a,b},{a,c},{b,c},{b,d},{b,e},{c,f},{e,f}}

Multigraphs

A multigraph G is a pair (V, E), where V is a set of vertices (representing the objects) and E is a *bag* of edges, where each edge in E is a set of 1 or 2 vertices (representing the links between vertices).

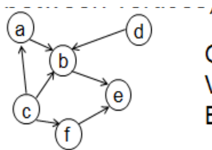


G = (V,E), where
V = {a,b,c,d,e}
E = {{a,b},{a,c},{a,c},{a,c},{b,c},{b,d},{b,d},
{b,e}}

- **A bag, or multi-set, is a set in which repeated elements are allowed**

Directed Graphs

A directed graph G is a pair (V, E), where V is a set of vertices (representing the objects) and E is a set of edges, where each edge in E is an ordered pair of vertices (representing the links between vertices).

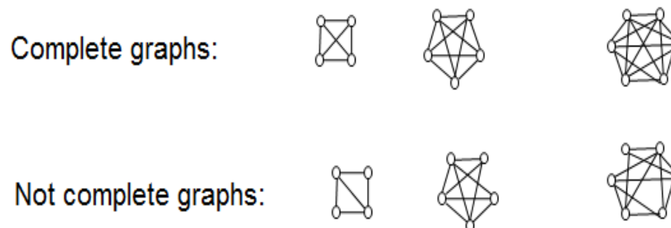


G = (V,E), where
V = {a,b,c,d,e,f}
E = {(a,b),(b,e),(c,a),(c,b),(c,f),(d,b),(f,e)}

Note: the order in the ordered pair matters.
(a,b) and (b,a) are different edges

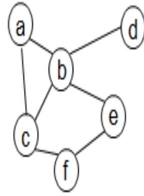
Complete graphs

A complete graph is one where every vertex is adjacent to every other vertex.



Vertex degree

The degree of a vertex, v, is the number of times edges are incident on v (where an edge from v to itself counts twice)



G = (V,E), where
V = {a,b,c,d,e,f}
E = {{a,b},{a,c},{b,c},{b,d},{b,e},{c,f},{e,f}}

- *The degree of vertex a is 2, and degree of vertex b is 4. The function deg:V→N returns the degree of any vertex.*

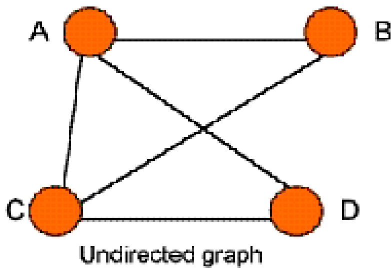
Forms of a Graph

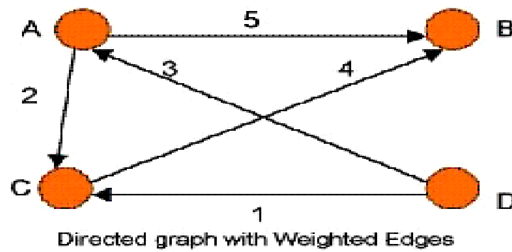
Directed and undirected graphs:

- A **directed graph** (**digraph**) is one in which the direction of any given edge is defined.
- In an **undirected graph**, G, one can move in both directions between vertices. The pairs (u, v) and (v, u) represent the same edge

Weighted or unweighted graphs:

- A graph is said to be ***weighted*** if each edge has an associated number (weight). Otherwise, it is ***unweighted*** graph.



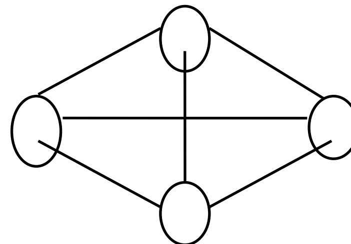


a) Undirected and un-weighted b) directed and weighted

Graph Representation

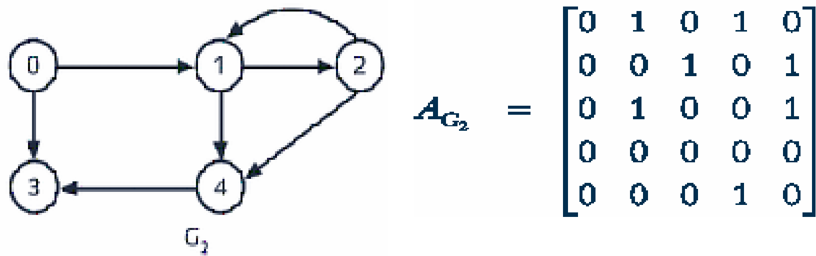
Adjacency Matrix

- Let G be a graph with n vertices, where $n > 0$.



- The adjacency matrix A_G is a two-dimensional $n \times n$ binary matrix such that matrix[i][j] storing whether there is an edge between the ith vertex and the jth vertex

- Matrix $A_G=[a_{ij}]$, where a_{ij} is 1 if there is an edge $\{v_i, v_j\}$, 0 otherwise

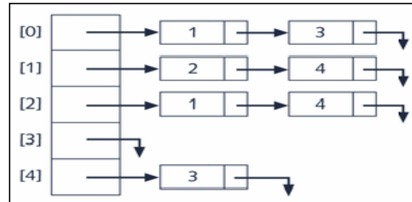


$$A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Exercise: what about for G_1

Adjacency Lists

- In adjacency list representation, corresponding to each vertex, v, is a linked list such that each node of the linked list contains the vertex u, such that $(v, u) \in E(G)$. Array, A, of size n, such that $A[i]$ is a pointer to the linked list containing the vertices to which v_i is adjacent. Each node has two components, (vertex and link)

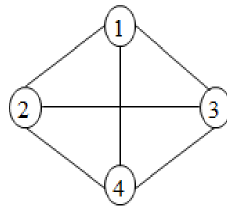


Exercise: (1) what about for G_1

(2) Can you suggest a better representation that solves drawback of Adjacency List?

Adjacency multilist

- In the adjacency multilist representation of a graph G
  - There will be one list for each vertex in G.
  - Each list has one node for each edge in G
  - The node represents the edge that has vertices Vi and Vj incident to a specific edge. Thus each node will be a member of two lists; one a member for vertex Vi and one for vertex Vj.
  - It has the following structure:



M	V1	V2	Link1 for V1	Link2 for V2

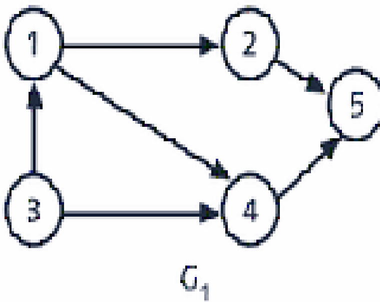
Vertex	Node	M	V1	V2	Link1	Link2	Edge
0	N1		0	1	N2	N4	(0,1)
1	N2		0	2	N3	N4	(0,2)
2	N3		0	3	NULL	N5	(0,3)
3	N4		1	2	N5	N6	(1,2)
	N5		1	3	NULL	N6	(1,3)
	N6		2	3	NULL	NULL	(2,3)

The adjacency lists are:
Vertex 0: N1-N2-N3
Vertex 1: N1-N4-N5
Vertex 2: N2-N4-N6
Vertex 3: N3-N5-N6

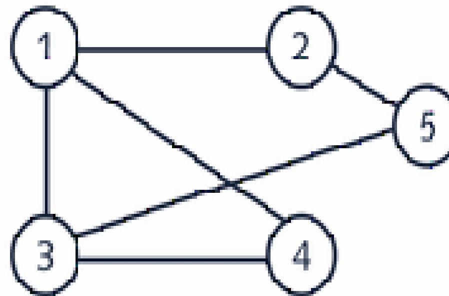
Exercise

Represent the given graph using the following techniques:

a) Adjacency matrix b) Adjacency list c) Adjacency multilist



i)



ii)

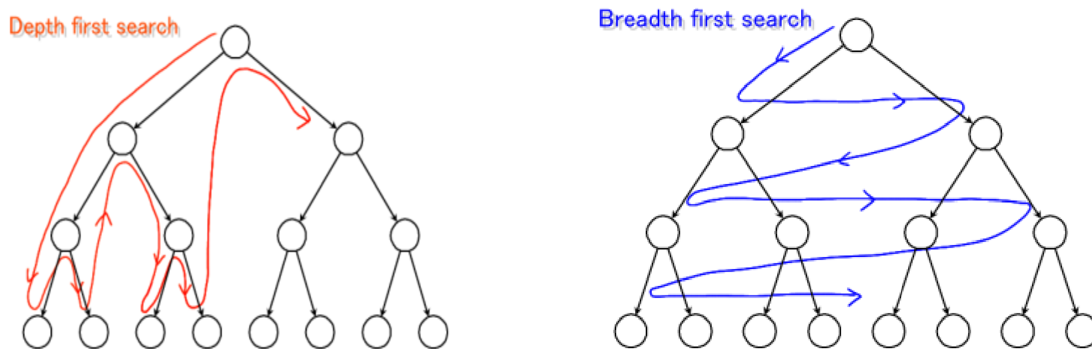
Graph Traversals

- Given a graph, one of the fundamental graph problems is to traverse every edge and vertex in a graph so as to:
  - count the number of edges
  - Identify connected components of a graph.
- **Goal**: visit all (or some) vertices and edges of the graph using some *strategy* (the order of visit is *systematic*)
- **Depth First Search** (DFS) and **Breadth First Search** (BFS) are examples of graph traversals. The order of exploring the vertices depends upon the data structure used:

  Queue: by storing the vertices in a queue, ***BFS*** explores vertices following FIFO order. Thus, it explores starting the initial vertex level by level.

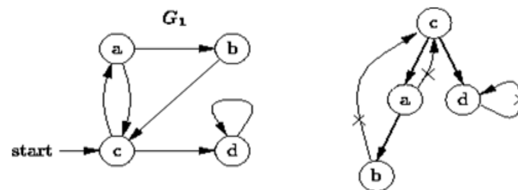
Stack: by storing the vertices in a stack, ***DFS*** explores vertices following LIFO order. Thus, it constantly visits a new neighbor depth-wise.

- Some shortest path algorithms and spanning tree algorithms have specific visit order.
- In DFS, as each vertex v is visited, all of it's children are added to front for immediate processing. Use of a stack leads to a *depth-first* visit order. Stack is used to keep track of: nodes to be visited next, or nodes that we have already visited.
- In BFS, as each vertex v is visited all of its unvisited children are kept in a waiting list. Use of a queue leads to a *breadth-first* visit order. Queue is used to keep track of: nodes to be visited next, or nodes that we have already visited.



Depth-First Traversal

- **Strategy**: Go as far as you can (if there is unvisited node depth-wise); otherwise, go back and try another way



- The depth-first traversal visits the nodes in the order - c, a, b, d

Remark:

- A depth-first traversal only follows edges that lead to unvisited vertices. If we omit the edges that are not followed, the remaining edges form a tree.

DFS: Algorithm

procedure DFS(v)

visited (v) = 1;   //Mark v as visited

For each vertex u adjacent to v do

If (visited (u) = 0) then   //if vertex u is unvisited

DFS(u)

end

procedure GraphTraversal()

for i = 1 to n   // n is the number of vertices v

visited(v_i) = 0;   //Mark v as unvisited

for i = 1 to n

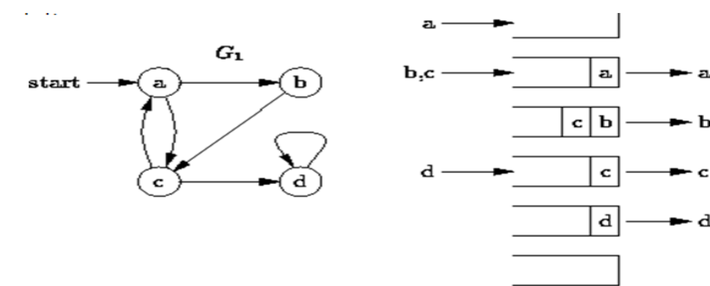
if (visited(v_i) = 0)

DFS(v_i)

end

Breadth-First Search (BFS)

- In DFS, we choose the most recently visited vertex to expand. Whereas, BFS explores the vertices in the order of their distance from the start vertex level-wise.
    - BFS examines every path of length i before going on to paths of length i+1.



BFS visits the nodes in the order: a, b, c, d

BFS: Algorithm

procedure BFS(v)

visited (v) = 1; //Mark v as visited

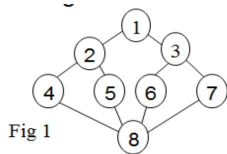
```
enqueue(v)

do{

for all vertices u adjacent to v do

If visited(u) = 0

visited(u) =1    enqueue(u)

v = dequeue()

}while queue is not empty

end


procedure  GraphTraversal()

initialize visited(v) = 0;          //Mark v as unvisited

for i = 1 to n // n is the number of vertices v

        if (visited(vi) = 0)

        BFS(vi)

 end
```

Time complexity

- **Proof:** Depending on the graph representation, time complexity for DFS and BFS is the same:
  - If Adjacency matrix : $O(V^2)$
  - If Adjacency list : $O(V + E)$

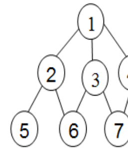
Exercise

1) Show the order of traversing the following graphs using DFS and BFS starting from node 1

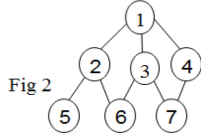


DFS: 1→2→4→8→5→6→3→7
BFS: 1→2→3→4→5→6→7→8

a)

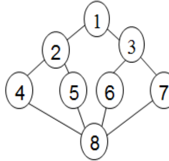


DFS: 1→2→5→6→3→7→4
BFS: 1→2→3→4→5→6→7



DFS: 1→2→5→6→3→7→4
BFS: 1→2→3→4→5→6→7

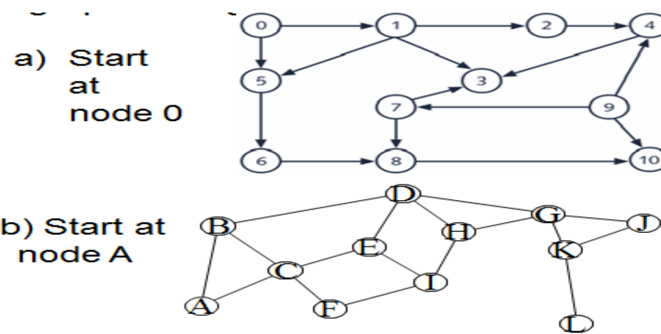
b)



DFS: 1→2→4→8→5→6→3→7
BFS: 1→2→3→4→5→6→7→8

Review Exercise

1) Find the order of traversing the following graphs using DFS and BFS



Chapter 4

The divide-and-conquer strategy

Open problem

- Given a sequence of n numbers, A = $<a_1, a_2 \dots a_n>$ and a value Key, write a code for linear search, which scans through the sequence, looking for Key, and returns one or more position i such that Key = A[i]?

Floor and Ceiling Functions

- The floor function, also called the greatest integer function or integer value gives the largest integer less than or equal to $\lfloor x \rfloor$. Ceiling function $\lceil x \rceil$, gives the smallest integer $\geq$ x, For all real x, $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$

Techniques for the Design of Algorithms:

- General approaches to the construction of *efficient* solutions to problems. Such methods are of interest because:

- They provide templates suited to solving a broad range of diverse problems which can be precisely analyzed.
- They can be translated into common control and data structures provided by most high-level languages.

- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.
  - The choice of design paradigm is an important aspect of algorithm analysis

Divide-and-Conquer Strategy

- ***Divide and Conquer*** is a general algorithm design paradigm that has created such efficient algorithms as **Merge Sort, Binary Search…**
- This method has three distinct steps:

  Divide: If the input size is too large, divide the input into two or more sub-problems. That is, divide **P** **P_1, …, P_k**, If the input size of the problem is small, it is solved directly

  Recur: Use divide and conquer to solve the sub-problems associated with each one-k^{th} of the data subsets separately, That is, find solution for **S(P_1), …, S(P_k)**

  Conquer: Take the solutions to the sub-problems and combine ("merge") these solutions into a solution for the original problem. That is, Merge **S(P_1), …, S(P_k) → S(P)**

Implementation: suppose we consider the divide-and-conquer strategy when it splits the input into two sub-problems of the same kind as the original problem.

- If the input size of the problem is small, it is solved directly.
- If the input size of the problem is large, apply the strategy:
  - Divide: divide the input data S in two disjoint subsets $S_1$and S_2
  - Recur: Solve each half of the sub-problems associated with S_1 and S_2
  - Conquer: combine the solution for $S_1$and S_2 into a solution for S

General Algorithm

procedure *DCS* (P)

if small(P) **then**

return S(P)

else

divide P into smaller instances $\mathbf{P_1, P_2 \ldots, P_k}$

apply DCS to each of these sub-problems

return (combine(DCS($\mathbf{P_1}$), *DCS(*$\mathbf{P_2}$), …, DCS($\mathbf{P_k}$))

end if;

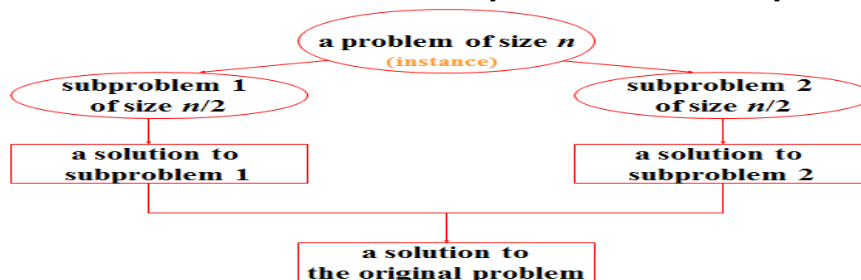
end *DCS*;

Complexity: f(n) { f(n) → n small

aT(n/b) + g(n) → otherwise, where:

- *b* be the ways we divide the problem at each step
- a be the number of sub-problems we solve at each step; i.e. n/b.
- *T*(*n*) be the time needed to solve the problem with input of size n
- *g*(*n*) be the time for dividing the problem and for combining solutions to sub-problems
- *f(n) be the time to compute the answer directly for small inputs*

Divide-and-Conquer Technique



- In general it leads to a recursive algorithm with complexity T(*n*) = 2 T(*n*/2) + g(*n*)

Solving Recurrence Relation

- One of the methods for solving recurrence relation is called the **substitution** method. This method repeatedly makes substitutions for each occurrence of the function T(n) until all such occurrences disappear.

Exercise: solve the following recurrence by substitution

1. $T(n) = \begin{cases} 2 & n = 1 \\ 2T(n/2)+n & n > 1 \end{cases}$

2. $T(n) = \begin{cases} 1 & n \leq 2 \\ 2T(n/2)+1 & n > 2 \end{cases}$

Example of Recursion: SUM A[1…n]

Problem: Write a recursive function to find the sum of the first n integers A[1…n] and output the sum

- Example: given k = 3, we return sum = A[1] + A[2] + A[3]

  given k = n, we return A[1] + A[2] + … + A[n]

- How can you define the problem in terms of a smaller problem of the same type?

  $$1 + 2 + \ldots + n = [1 + 2 + \ldots + (n - 1)] + n$$
  $$\text{for } n > 1,\ f(n) = f(n.1) + n$$

- How does each recursive call diminish the size of the problem? It reduces by 1 the number of values to be summed. What instance of the problem can serve as the base case? n = 1

- As the problem size diminishes, will you reach this base case? Yes, as long as n is nonnegative. Therefore the statement "n >= 1" needs to be a precondition

algorithm LinearSum(*A*, *n*)

// ***Input:*** an array *A with* n *element*s

// ***Output:*** The sum of the first *n* integers in *A*

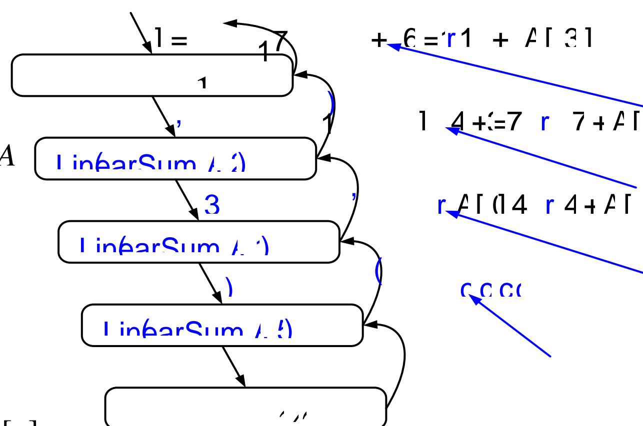
if *n* = 1 **then**

return *A*[0]

else

return LinearSum(*A*, *n* - 1) + *A*[*n*]

end algorithm



Binary Recursive Method

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.

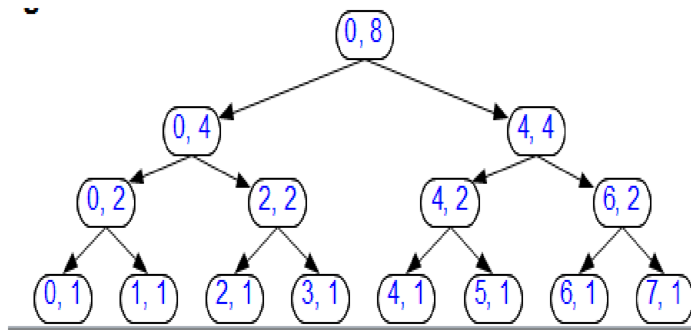
Algorithm BinarySum(A, i, n):

if $n = 1$ **then**

return $A[i]$

return (BinarySum(A, i, $n/2$) + BinarySum(A, $i + n/2$, $n/2$))

end algorithm



Binary search

- Binary Search is an algorithm to find an item in a sorted list.
  - very efficient algorithm for searching in **sorted array**
  - **Limitations**: must be a sorted array

Problem: *determine whether a given element* **K** *is present in the given list or not*

Input: *Let* A = <a_1, a_2, ... a_n> *be a list of elements that are sorted in non-decreasing order.*

Output: If K is present output its position. Otherwise output "Not Found".

Implementation:

- Pick the pivot item in the middle: Split the list in two halves (size n/2) at m so that

A[1], … A[m], … A[n].

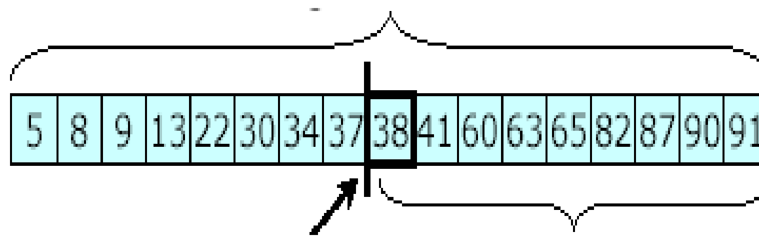
  - If K = A[m], stop (successful search);

- Otherwise, until the list has shrunk to size 1 narrow our search recursively to either the top half of the list : A[1..*m*-1] if $K < A[m]$ or the bottom half of the list: A[*m*+1..*n*] if $K > A[m]$

Example

- Binary Search for 64 in the given list A[] = {5 8 9 13 22 30 34 37 41 60 63 65 82 87 90 91}

1. Looking for 64 in this list.
2. Divide the list into two (1+17)/2 = 9
3. Pivot = 38. Is 64 < 38? No.
4. Recurse looking for 64 in the list > 38.
5. etc.



Binary Search Recursive Algorithm

- Four Questions in designing recursive algorithm :
  - How can you define the problem in terms of a smaller problem of the same type? Look at the middle of the list. Then recursively search the top or bottom half, as appropriate.
  - How does each recursive call diminish the size of the problem? It cuts the size of the list in half (roughly).
  - What instance of the problem can serve as the base case? Base case = 1.
  - As the problem size diminishes, will you reach this base case? Yes, A list cannot have negative size.

```
procedure BSearch(A, low, high, key)
        // A is sorted array. Low =1, high = n
        if low = high  then
            if key = A[low] then return low
             else return "Not Found";
```

```
            end if
        else
            mid = (low + high)/2;
            if key > A[mid]
                return BSearch(A, mid+1, high, key);
            else
                return BSearch(A, low, mid-1, key);
            end if
        end if
    end algorithm
```

Binary Search Iterative Algorithm

```
Procedure BinarySearch(A, n, key)
    low ← 1;   high ← n;
    while low ≤ high do
            mid ← ⌊(low+high)/2⌋
    if  key = A[mid]  then
                return mid
    else if  key < A[mid]  then
                high ← mid-1
    else     low ← mid+1
    return "NotFound"
end
```

Analysis: considering the number of element comparison, the worst-case recurrence is:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

$T(n) = O(\log 2n)$. **Show?**

Finding the minimum & maximum

- Let there are n elements $< a_1, a_2, \ldots a_n >$. The problem is to find max and min elements in a set.

Straightforward algorithm

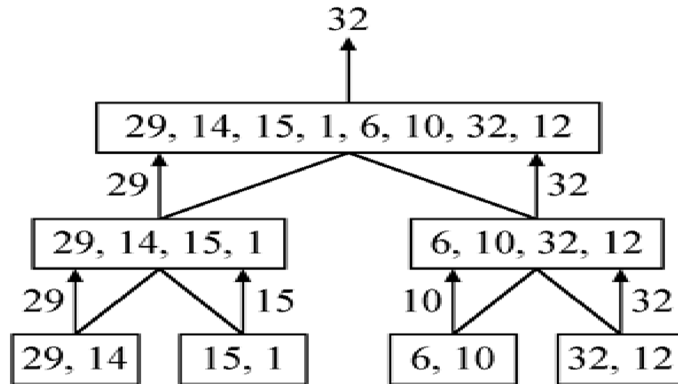
```
procedure max_min(A, n, max,min)
    max = min = A[1]
    for i = 2 to n do
        if A[i] > max then
            max = A[i]
        if A[i] < min then
            min = A[i]
end procedure
```

Analysis: there are **2 (n-1) numbers** of element comparisons in the best, worst and average cases. Can you suggest any improvement in the max_min algorithm?

Divide and conquer for finding min & max

- Recurrence relation:
  - If n =1, both max and min are the same. max = min = a[1]
  - If n =2 the problem can be solved by making one element comparison a[1] > a[2] or a[1] < a[2]
  - If n > 2 divide $< a_1, a_2, \ldots a_n >$ into two instances a[1..n/2] and a[n/2+1…n] and solve the sub-problems recursively.

Example: find the maximum of a given set of n numbers a[] = {29 14 15 1 6 10 32 12}

Solution: 

Recursive algorithm

```
procedure max_min(A, low, high, max, min)
    if low = high then
        max = min = A[1]
    else if low = high – 1 then
        if A[low] < A[high] then
            max = A[high];        min = A[low]
        else
            max = A[low];         min = A[high];
        end if
    else
        mid = ⌊(low+high)/2⌋
        max_min(A, low, mid, max, min);
        max_min(A, mid+1, high, max2, min2);
        if max < max2 then   max = max2;
        if min > min2 then        min = min2;
    end if
end procedure
```

Time complexity

- **Analysis**: considering the number of element comparison, the worst-case recurrence is:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

$T(n) = O(n)$. **Show?**

Merge sort

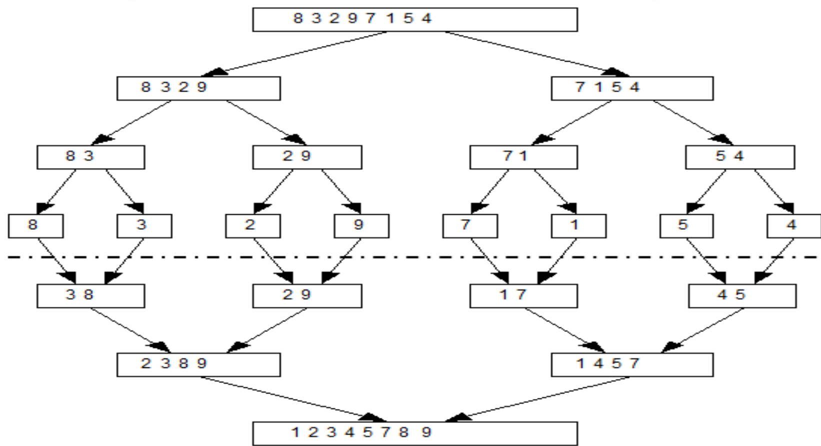
- Given a set of n elements A = < $a_1, a_2, \ldots a_n$>, design an algorithm which sort them in non-decreasing order.

Implementation:

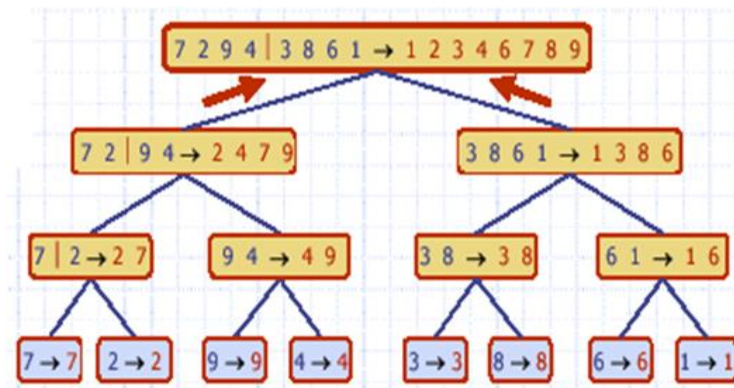
- Split the list in to two halves and sort each half
- Merge the two halves as follows:
  - ✓ Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - ✓ Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Example: consider array of ten elements: a[1:10]=(31,28,17,65,35,42,86,25,45,52)

Merge sort Execution Example



Merge-Sort Execution Example



Merge Algorithm

procedure merge(low, mid, high)

h ← *low*; i ← low; j ← mid+1;

while h ≤ mid and j ≤ high do

if A[h] ≤ A[j] then

B[i] = A[h]; h = h+1

else B[i] = A[j]; j = j+1

i = i+1

if h > mid then

for k = j to high do

```
                B[i] = A[k];    i = i+1
        else
            for k = h to mid do
                B[i] = A[k];    i = i+1
        endif
        for k = low to high do
            A[k] = B[k];
    end
```

Merge sort Algorithm

```
    procedure mergesort(low,high)
        if low < high then //if there are more than one element
            mid ← ⌊(low+high)/2⌋
        mergesort(low, mid)
        mergesort(mid+1, high)
            merge(low,mid,high)
        end if
    end
```

Analysis: considering the number of element comparison, the worst-case recurrence is:

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

$T(n) = O(n\log_2 n)$. **Show?**

Chapter 5

Greedy Algorithms

The greedy method

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution. A "greedy algorithm" works well for most optimization problems. Greedy method suggests that one can devise an algorithm that works in phases:
    - At each phase, take one input at a time (from the ordered input according to the selection criteria) and decide whether it is an optimal solution.

Feasible vs. optimal solution

- Greedy method solves problem by making a sequence of decisions.
    - Decisions are made one by one in some order.
    - Each decision is made using a greedy criterion.
    - A decision, once made, is (usually) not changed later.
- Given n inputs we are required to obtain a subset that satisfies some constraints
    - Any subset that satisfies the given constraints is called a **feasible solution**
    - A feasible solution that either maximizes or minimizes a given objective function is called an **optimal solution**.

Greedy algorithm

- To apply greedy algorithm
    - Decide optimization measure (maximization of profit or minimization of cost)
        - ✓ Sort the input in increasing or decreasing order based on the optimization measure selected for the given problem
    - Formulate a multistage solution
        - ✓ Take one input at a time as per the selection criteria
    - Select an input that is feasible and part of the optimal solution
        - ✓ from the ordered list pick one input at a time and include it in the solution set if it fulfills the criteria

Greedy Choice Property

- *Greedy algorithm* always makes the choice that looks best at the moment with the hope that a locally optimal choice will lead to a globally optimal solution. It says that a globally optimal solution can be arrived at by making a locally optimal choice

  - ✓ Locally optimal choice ⇒ globally optimal solution

The Problem of Making Coin Change

- Assume the coin denominations are: 25, 10, 5, and 1.
- **Problem:** Make a change of a given amount using the smallest possible number of coins

Example: make a change for x = 92.

- ✓ Mathematically this is written as $x = 25a + 10b + 5c + 1d$

  So that $a + b + c + d$ is minimum & $a, b, c, d \geq 0$.

Greedy algorithm for coin changing

- Order coins in decreasing order
- Select coins one at a time (divide x by denomination)
- **Solution:** contains a = 3, b = 1, c = 1, d = 2.

Algorithm

```
procedure greedy (A, n)
    Solution ← { };   // set that will hold the solution set.
    FOR i = 1 to n DO

            x = SELECT (A)
        IF FEASIBLE (Solution, x) THEN
            Solution = UNION (Solution, x)
        end if

    end FOR

    RETURN Solution

end procedure
```

SELECT function:

- ✓ Selects the most promising candidates from A[ ] and removes it from the list.

FEASIBLE function:

- ✓ a Boolean valued function that determines whether x can be included into the solution vector.

UNION function:

- ✓ combines x with the solution

Algorithm for Coin Change

- Make change for n units using the least possible number of coins.

```
Algorithm MAKE-CHANGE (C, n, A)
   //C ← {50, 25, 10, 5, 1}

//A is the amount to be changed
 Sol ← {}              // initialize Sol
 rem = A
WHILE rem > 0 & i < n DO

              Sol[k++] = rem / C[i++]
      rem = rem mod C[i++]

      end while
RETURN Sol

  end algorithm
```

Knapsack problem

- In a knapsack problem, we are given n items (where i =1, 2 . . . n) and a knapsack. Item i has weight $w_i > 0$ and the knapsack has a capacity of W.
- If a fraction x_i, $0 \le x_i \le 1$ of item i is placed in the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain the filling of the knapsack that maximizes the total profit earned. That is,

$$\text{maximize: } \sum_{i=1}^{n} p_i x_i \qquad \ldots\ldots (1)$$

subject to constraints:

$$\sum_{i=1}^{n} w_i x_i \le W \qquad \ldots\ldots (2)$$

$$\textit{and} \quad 0 \le x_i \le 1 \ \textit{for } i = 1, 2, \ldots, n \qquad \ldots\ldots (3)$$

- Thus a feasible solution is any set (x_i, *$i = 1 \ldots n$) satisfying (2) and (3). An* optimal solution is a feasible solution that maximizes (1).

Greedy Algorithm

Example: consider the following instances of the knapsack problem.

n=3, W=20, (p_1, p_2, p_3) = (25, 24, 15), and (w_1, w_2, w_3) = (18, 15,10)

- *Three measures can be applied to find the solution*
    - Profit of items
    - Weight of items
    - Ratio of profit to weight of items

```
procedure Greedy-knapsack (w, x, W, n)
    //w[i] is sorted array in decreasing order of pi/wi
    FOR i =1 to n do
          x[i] =0
  capacity = W;      i = 1
  WHILE w[i] < capacity & i ≤ n do
    IF capacity ≤ w[i] then        break;
       x[i] = 1
    capacity = capacity - w[i++]
    end while
  IF i ≤ n then
    x[i] = capacity / w[i]
end procedure
```

- If the items are already sorted into decreasing order of P_i / w_i, then the while-loop takes a time in *O (n)*. Complexity of greedy algorithm including the sort is in *O(n log n)*.

Analysis of Knapsack problem

- Is the algorithm correct?

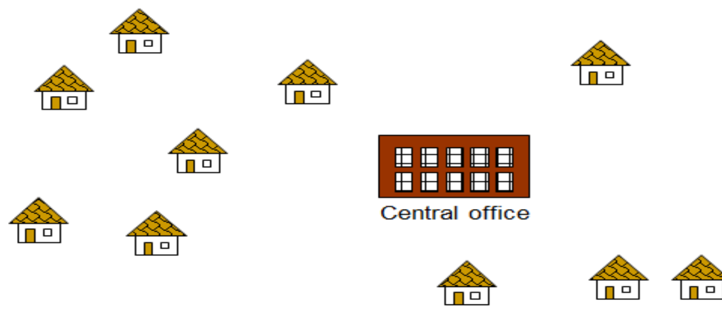
- If the algorithm is correct it halts with the right answer or optimal solution. The optimal solution to this problem is obtained by sorting items by the ratio of profit to weight of items.
- Optimal solution fills the knapsack completely. Because we can always increase the contribution of some object i by a fractional amount until the total weight is exactly W.

Proof (by contradiction):

- That greedy method generates an optimal solution to the given instance of the knapsack problem.

Minimum Spanning Trees

Problem: Laying Telephone Wire and Minimize the total length of wire connecting the customers.



- Assume you have an undirected graph G = (V,E) with weights assigned to edges. The objective is "use smallest set of edges of the given graph to connect everything together". How? A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes. It is a sub-graph of an undirected weighted graph ***G***, such that:
    - It is a tree (i.e., it is acyclic)
    - It covers all the vertices ***V***
    - contains ***/V/ - 1*** edges
    - The total cost associated with tree edges is the minimum among all possible spanning trees
- Applications of MST is:
    - Network design, road planning, etc.

How can we generate a MST?

- A MST is a least-cost subset of the edges of a graph that connects all the nodes. A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion.
- **Criteria**: to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included.

General procedure:

- Start by picking any node and adding it to the tree
- Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
- Stop when all nodes have been added to the tree

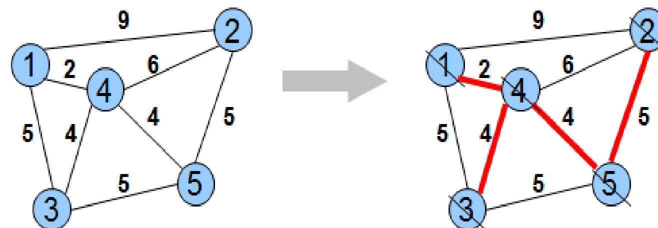
Prim's algorithm

Prim's: Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree.

- If A is the set of edges selected so far, then A forms a tree.
    - The next edge (u,v) to be included in A is a minimum cost edge not in A such that A υ {(u,v)} is also a tree, where ***u*** is in the tree & ***v*** is not.

Property: At each step, we add the edge (***u,v***) s.t. the weight of (***u,v***) is **minimum** among all edges. This spanning tree grows by one new node and edge at each iteration. Each step maintains a minimum spanning tree of the vertices that have been included thus far. When all vertices have been included, we have a minimum cost spanning tree for the graph

Example: find the minimum spanning tree using Prim algorithm



Prim's Algorithm

procedure primMST(G, cost, n, T)

```
Pick a vertex 1 to be the root of the spanning tree T
mincost = 0
for i = 2 to n do  near (i) = 1
near(1) = 0
 for i = 1 to n-1 do
   find j such that near(j) ≠ 0 and cost(j,near(j)) is min
   T(i,1) = j;   T(i,2) = near (j)
   mincost = mincost + cost(j,near(j))
   near (j) = 0
   for k = 1 to n do
       if near(k) ≠ 0 and cost(k,near(k) > cost(k,j)  then
              near (k) = j
   end for
 end for
 return mincost
end procedure
```

Correctness of Prim's

- If the algorithm is correct it halts with the right answer or optimal solution. Optimal solution is obtained if:
    - Prim algorithm adds *n-1* edges (with minimum cost) to the spanning tree without creating a cycle

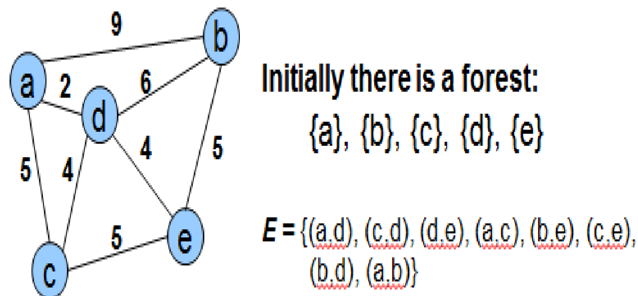
Kruskal's algorithm

- Kruskal algorithm: Always tries the *lowest-cost* remaining edge. It considers the edges of the graph in increasing order of cost. In this approach, the set T of edges so far selected for the

spanning tree may not be a tree at all stages in the algorithm. But it is possible to complete T into a tree.

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains. A "safe edge" is an edge of minimum weight which does not create a cycle

Example:



Kruskal Algorithm

```
  procedure kruskalMST(G, cost, n, T)
i = mincost = 0
while i < n – 1 do
        delete a minimum cost edge (u,v)
        j = find(u)
        k = find(v)
        if j ≠ k then
           i = i +1
           T(i,1) = u;   T(i,2) = v
           mincost = mincost + cost(u,v)
           union(j,k)
        end if
  end while
```

if i ≠ n-1 then return “no spanning tree”

return mincost

end procedure

- After each iteration, every tree in the forest is a MST of the vertices it connects
- Algorithm terminates when all vertices are connected into one tree
- Running time is bounded by sorting (or findMin): O($\mathbf{n^2}$)

Correctness of Kruskal

- If the algorithm is correct it halts with the right answer or optimal solution. Optimal solution is obtained if:
  - Kruskal algorithm adds *n-1* edges (with minimum cost) to the spanning tree without creating a cycle

Dijkstra's shortest-path algorithm

- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph. Always takes the *shortest* edge connecting a known node to an unknown node.
- **Initially**,
  - Mark the given node as *known* (path length is zero)
  - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
- Repeatedly (until all nodes are known),
  - Find an unknown node containing the smallest distance
  - Mark the new node as known
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - ✓ If so, also reset the predecessor of the new node

PROCEDURE shortestPath(v,COST,DIST,n)

start with V_1(s)

FOR i = 1 to n DO //initialize S and DIST

S(i) = 0; DIST(i) = COST(v,i);

```
END FOR
S(v1) = 1
FOR num = 2 to n-1 DO
        choose vertex u such that S(u) = 0 and DIST(u) is min
  S(u) = 1 //put u in S
        FOR each w adjacent to u with S(w) = 0 DO
            IF DIST[w] > DIST[u] + COST(u,w)  THEN
                DIST[w] = DIST[u] + COST(u,w);
   END FOR
 END FOR
END PROCEDURE
```

Huffman Coding

- **The problem**: Given a set of n messages and their weights (or frequencies), construct a set of code words so that the expected decoding time per symbol is minimized. Each code is a binary string that is used for transmission of the corresponding message.
  - At the receiving end the code is decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages.
  - The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node.

How to minimize decoding time?

- The Huffman encoding algorithm is a greedy algorithm
  - You always pick the two smallest numbers to combine
  - Example: given the following, apply the Huffman algorithm to find an optimal binary code:

Character:	b	e	c	a	d	t
Frequency:	5	10	12	16	17	25

Algorithm

```
procedure HuffmanCode(L,n)
    for i = 1 to n-1 do
        r = new Nodetype
        r→lchild = least(L)
        r→rchild = least(L)
        r→frequency = r→lchild→frequency + r→rchild→frequency
        insert(l,r)
    end for
  return (least(L))
end procedure
```

Chapter 6

Dynamic programming

Divide & Conquer vs. Dynamic Programming

- Both techniques split their input into parts, find sub-solutions to the parts, and combine solutions to sub-problems. In divide and conquer, solution to one sub-problem may not affect the solutions to other sub-problems of the same problem. In dynamic programming, sub-problems are not independent. Sub-problems may share sub-sub-problems

Greedy vs. Dynamic Programming

- Both techniques are an algorithm design technique for optimization problems (minimizing or maximizing), and both build solutions from a collection of choices of individual elements. The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution forward/backward by synthesizing them from smaller sub-solutions, and by trying many possibilities and choices before it arrives at the optimal set of choices. There is no a priori test by which one can tell if the Greedy method will lead to an optimal solution.
- By contrast, there is a test for Dynamic Programming, called **The Principle of Optimality**

The Principle of Optimality

- In DP an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.

Definition: A problem is said to satisfy the Principle of Optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems.

- In solving a problem, we make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the k decisions also be optimal

Examples: The **shortest path problem** satisfies the principle of optimality.

- This is because if a, $x_1, x_2, \ldots, x_n$, b is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j.
- DP reduces computation by:
    - Storing solution to a sub-problem the first time it is solved.
    - Looking up the solution when sub-problem is encountered again.
    - Solving sub-problems in a bottom-up or top-down fashion.

Dynamic programming (DP)

- DP is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Example: The solution to knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of x_i, $1 \leq i \leq n$. First we make a decision on x_1, then x_2 and so on.

- For some problems, an optimal sequence of decisions can be found by making the decisions one at a time using greedy method. For other problems, it is not possible to make step-wise decisions based on only local information. One way to solve such problems is to try all possible decision sequences. However time and space requirement is prohibitive.
- DP reduces those possible sequences not leading to optimal decision.

Dynamic programming approaches

- To solve a problem by using dynamic programming:
    - Find out the recurrence relations.
        - ✓ Dynamic programming is a technique for efficiently computing recurrences by storing partial results.
    - Represent the problem by a multistage graph.
    - In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming

Forward approach and backward approach:

- If the recurrence relations are formulated using the backward approach, then the relations is solved beginning with the last decision.
- If the recurrence relations are formulated using the forward approach, then the relations are solved starting from the beginning until we each to the final decision

Example: 0-1 knapsack problem

0-1 Knapsack problem

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected. Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.
- Given a knapsack with maximum capacity *W*, and a set of *n* items with weight $w_1, w_2, \ldots, w_n$ and benefit value $p_1, p_2, \ldots, p_n$ (all w_i, p_i & W are integer values), the problem is:
  - How to pack the knapsack to achieve maximum total value of packed items? That is:

$$\text{maximize} \rightarrow \sum_{1 \le i \le n} P_i x_i$$

$$\text{subject to: } \sum_{1 \le i \le n} W_i x_i \le W \text{ and } x_i = 0 \text{ or } 1, \ 1 \le i \le n$$

0/1 Knapsack: Brute-force approach

- In solving this problem with a straightforward algorithm, for the *n* items, we consider 2^n possible combinations of items. We go through all combinations and find the one with the best benefit/profit and with total weight less or equal to *W*
  - Running time - $O(2^n)$

Can we do better?

- Yes, with an algorithm based on dynamic programming approach such as backward or forward. Assuming $S_i(W)$ is the optimal solution, we can recursively obtain $S_1, \ldots, S_n$ following:

Backward approach: solves the recurrence relations beginning with the last decision. N.B. If $S_0(W) = 0$, "$_w \ge 0$ and $S_0(W) = -\text{inf}$, "$_w \le 0$

$$S_i(W) = \max \{S_{i-1}(W), S_{i-1}(W - w_i) + P_i\}$$

Forward approach: solves the recurrence relations starting from the beginning until we reach to the final decision.

$$S_i(W) = \max \{S_{i+1}(W), S_{i+1}(W - w_{i+1}) + P_{i+1}\}$$

Note: If $S_n(W) = 0$, "$_w \geq 0$ and $S_n(W) = -inf$, "$_w \leq 0$

Example: consider the case in which n=3, W=6, (w_1, w_2, w3) = (2, 3, 4), and (p_1, p_2, p_3) = (1, 2, 5).

0/1 Knapsack: DP approach

- To solve a knapsack problem we need to carefully identify the sub-problems. If items are labeled *1…n*, then a sub-problem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k} along with w*, which will represent the exact weight for each subset of items
- The sub-problem then will be to compute *S[k,w]*

$$S[k,w] = \begin{cases} S[k-1,w] & \text{if } w_k > w \\ \max\{S[k-1,w], S[k-1,w-w_k]+p_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight *w* is one of the two:

  First case: $w_k > w$. Item *k* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable

  Second case: $w_k <= w$. Then the item *k* can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
procedure DPknapsack(P[], w[], W, n)
  for w = 0 to W do
        S[0,w] = 0
  for i = 1 to n do
          S[i,0] = 0
        for w = 1 to W do
              if wi <= w then // item i can be part of the solution
                    if pi + S[i-1,w-wi] > S[i-1,w] then
                          S[i,w] = pi + S[i-1,w- wi]
                    else
```

S[i,w] = S[i-1,w]

else S[i,w] = S[i-1,w] // $w_i > w$

end procedure

- Running time is reduced from *$O(2^n)$ (brute force approach)* to O(n*W)

Example:

- Let's run our algorithm on the following data: n = 4 (# of elements) W = 5 (max weight) Elements (weight, profit): (2,3), (3,4), (4,5), (5,6)

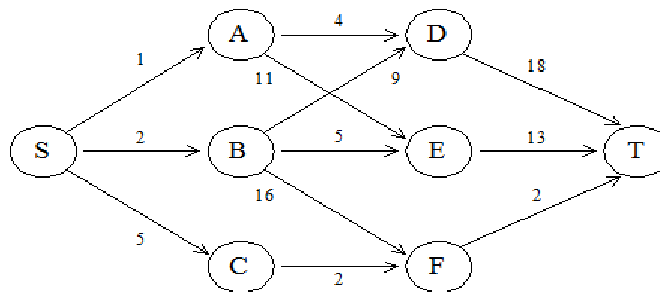
i \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

The shortest path in multistage graphs

- Find the shortest path in multistage graphs for the following example?



- The greedy method cannot be applied to this case: (S, A, D, T) 1+4+18 = 23. The real shortest path is: (S, C, F, T) 5+2+2 = 9.

The shortest path

- Given a multi-stage graph, how can I find a shortest path?
  - Let p(i,j) denote the minimum cost path from vertex j to the terminal vertex T. Let COST(i,j) denote the cost of p(i,j) path. Then using the
- **Forward approach** we obtain:

COST(i,j) = min { COST(i+1,k) + c(k,j)}

(j,k) Є E   k Є Vi+1

- **Backward approach**: Let p(i,j) be a minimum cost path from vertex S to a vertex j in V_i. Let COST(i,j) be the cost of p(i,j).

COST(i,j) = min { COST(i,k) + c(k,j-1)}

k Є Vi+1

(j,k) Є E

Note: If (i, j) is not element of E then COST(i, j) = inf.

Algorithm

procedure shortest_path (COST[], A[], *n*)

//cost[i,j] is the cost of edges[i,j] and A[i,j] is the shortest path from i to j

//cost[i,i] is 0.0

for *i = 1 to n* **do**

for *j = 1 to n* **do**

A(i, j) := COST(i, j) //copy cost into A

for *k = 1 to* **do**

for *i = 1 to* **do**

for *j = 1 to* **do**

*A(i, j) = min(*A*(i, j), A(i,k) + A(k,j))*;

end for

end for

end for

return A*(1..n,1..n)*

end *shortest_path*

- *This algorithm runs in time **O(n^3)***

String editing

- The problem is given two sequences of symbols, $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_m$, transform X to Y, based on a sequence of three operations: **Delete, Insert** and **Change,** so that for every operation $COST(C_{ij})$ is incurred.
- The objective of string editing is to identify a minimum cost sequence of edit operation that will transform X into Y.

Example: consider the sequences X = {a a b a b} and Y = {b a b b}. Identify a minimum cost sequence of edit operation that transform X into Y. Assume change costs 2 units, delete and insert 1 unit.

(a) apply brute force approach

(b) apply dynamic programming

Dynamic programming

- The minimum cost of any edit sequence that transforms $x_1 x_2 \dots x_i$ into $y_1 y_2 \dots y_j$ (for i>0 and j>0) is the minimum of the three costs: delete, change, or insert operations. The following recurrence equation is used for COST(i,j).

$$COST(i,j) = \begin{cases} 0 & \text{if } i=0, j=0 \\ COST(i-1,0) + D(x_i) & i>0, j=0 \\ COST(0,j-1) + I(y_j) & j>0, i=0 \\ COST'(i,j) & i>0, j>0 \end{cases}$$

$$\text{Where } COST'(i,j) = \min\{ COST(i-1,j) + D(x_i),\ COST(i-1,j-1) + C(x_i,y_j),\ COST(i,j-1) + I(y_j) \}$$

It takes O(n,m)

<u>**Example**</u>

- Transform the sequences X = {a a b a b} and Y = {b a b b} with minimum cost sequence of edit operation using dynamic programming approach, Assume that change costs 2 units, delete and insert 1 unit.

i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

- ✓ The value 3 at (5,4) is the optimal solution

- By tracing back one can determine which operations lead to optimal solution
  - Delete x_1, Delete x_2 and Insert y_4 Or, Change x_1 to y_1 & Delete x_4.

Chapter 7

Backtracking Algorithm

Backtracking

- It is a systematic way to search for a solution to a problem among all available options in a search space. It does so by assuming that the solutions are represented by vectors ($v_1, ..., v_m$) of values and by traversing, in a **depth first manner**, the domains of the vectors until the solutions are found.
  - Often the problem to be solved calls for finding **one vector** that maximizes (or minimizes or satisfies) a criterion function $P(x_1, ..., x_m)$.
- We build from a partial solution of length k, $v = (a_1, ..., a_k)$ and try to extend it by adding another element. After extending it, we will test whether what we have so far is still possible as a partial solution.
  - If it is still a candidate solution, great. If not, we delete a_k and try the next element from S_k:

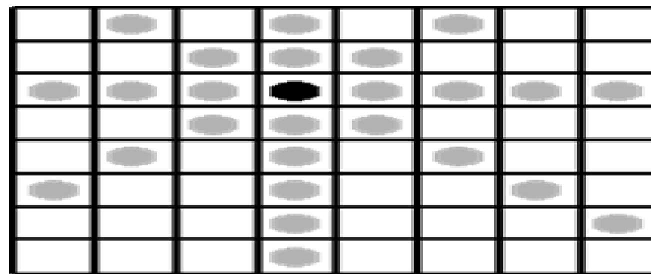
Backtracking approach

- An important requirement in backtracking is that there must be proper hierarchy in systematically searching for solutions so that sets of solutions that do not fulfill a certain requirement are rejected before the solutions are produced.
  - For this reason the examination and production of the solutions follows a model of non-cycle graph for which in this case we will consider as a *tree*.
  - It is easily understood that the tree (or any other graph) is produced during the examination of the solutions so that no rejected solutions are produced.
  - When a node is rejected, the whole sub-tree is rejected, and we backtrack to the ancestor of the node so that more *children* are produced and examined.

The Queens Problem

- Consider a *n by n* chess board, and the problem of placing *n* queens on the board without the queens threatening one another.

- The solution space is $\{1, 2, 3, \ldots, n\}^n$. The backtracking algorithm may record the columns where the different queens are positioned. Trying all vectors $(p_1, \ldots, p_n)$ implies n^n cases queens threatening one another.
- Noticing that all the queens must reside in different columns reduces the number of cases to $n!$
- For the latter case, the root of the traversal tree has degree n, the children have degree $n - 1$, the grand children degree $n - 2$, and so forth.

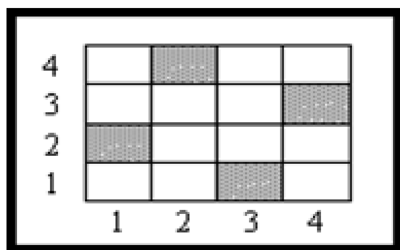


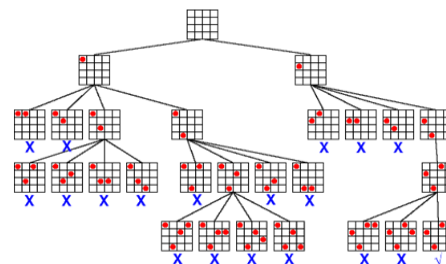
How backtracking works

- As a bounding function we use if $(x_1, \ldots, x_i)$ is the path to the current E-node (node being expanded), then all children nodes with parent-child labeling x_{i+1} are such that $(x_1, \ldots, x_{i+1})$ represents the chessboard configuration in which no two queens are attacking.
- Start with the root node as the only live node. This become the E-node and the path is (). Generate one child (say 2). The path is now (1), which corresponds to placing queen 1 on column 1.
- Node 2 becomes the E-node. Node 3 is then generated. If the node is attacked by the previous node, the path is immediately killed. Otherwise add to the path list (1, 2) and generate the next node. If the path cannot lead to an answer node then backtrack and try another path.

4 - Queens

- The problem is to place four queens on an 4 x 4 chessboard so that
    - No two attacks, i.e. no two of them are on the same row, column or diagonal.





Algorithm for n-queens

- Let (x_1, ..., x_n) represent a solution in which x_i is the column of the ith row where the ith queen is placed. All x_i's be distinct since no two queens can be placed in the same column.
    - Computing time = 0 + 1 + 2 + … + (n-

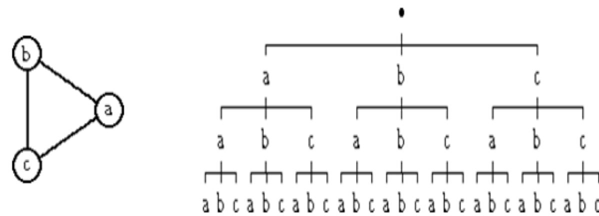
Algorithm

```
procedure NQueens(k,n)
for i = 1 to n do
    if place(k,i) then
        x[k] = i
        if k = n then print (x[1:n])
        else NQueens(k+1, n)
    end if
end for
end NQueens
procedure place(k,i) //returns true if a queen is placed at (k,i)
for j = 1 to k-1 do
    if (x[j] = i) or ((abs(x[j] - i) = (abs(j - k)) then
     //two in the same column or in the same diagonal
        return false
return true
end place
```

Traveling Salesperson Problem (TSP)

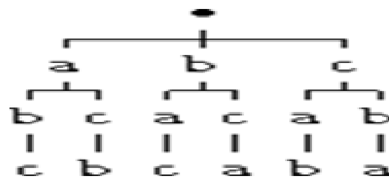
- The problem assumes a set of *n* cities, and a salesperson which needs to visit each city exactly once and return to the base city at the end. The solution should provide a route of minimal length. The traveling salesperson problem is an NP-hard problem, and so no polynomial time algorithm is available for it.

- Given an instance $G = (V, E)$ the backtracking algorithm may search for a vector of cities $(v1, ..., v|V|)$ which represents the best route.
- The validity criteria may just check for number of cities in the routes, pruning out routes longer than $|V|$. In such a case, the algorithm needs to investigate $|V|^{|V|}$ vectors from the solution space.

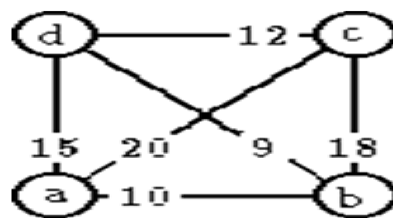


Traveling Salesperson

- On the other hand, the validity criteria may check for repetition of cities, in which case the number of vectors reduces to $|V|!$.
  - That is, complexity = n!



- Given the following problem, starting from city '**a**' apply backtracking algorithm to find the shortest path to visit all cities (and back to city a).



- The route (a, b, d, c) is the shortest one with length = 51. Can we reach to this decision using backtracking algorithm?

Branch and Bound Algorithm

Approach

- Track best current solution found
- Eliminate partial solutions that cannot improve upon best current solution

- Reduces amount of backtracking
  - Not guaranteed to avoid exponential time $O(2^n)$

Example: Travel Salesperson

- Branch and bound algorithm for TSP
  - Find possible paths using recursive backtracking
  - Track cost of best current solution found
  - Stop searching path
    - ✓ if cost > best current solution
  - Return lowest cost path
- ➢ If good solution found early, can reduce search
- ➢ May still require exponential time $O(2^n)$