

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq

Class: Third

Branch :A.I

1st course 2020

Lecture (1)

INTRODUCTION

The term algorithm is universally used in computer science to describe problem-solving methods suitable for implementation as computer programs they are central objects of study in many, if not most, areas of the field.

INTRODUCTION

- Definition
- Data structure :they are objects created during complicated methods of organizing the data involved in the computation such as list, stack, queue, tree, graph ,and double ended queue and so on .

The Role of Algorithms in Computing

- Definition
- an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. An ***algorithm*** is thus a sequence of computational steps that transform the input into the output. We can also view an ***algorithm*** as a tool for solving a well-specified computational problem.

The Role of Algorithms in Computing

- Example
- For example, one might need to sort a sequence of numbers into increasing order
- formally define the ***sorting problem***:
- • **Input:** A sequence of n numbers (a_1, a_2, \dots, a_n).
- • **Output:** A permutation (reordering) (a_1', a_2', \dots, a_n') of the input sequence such that $a_1' \leq a_2' \leq \dots \leq a_n'$.

The Role of Algorithms in Computing

- ***instance of a problem*** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.
- An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output. We say that a correct algorithm ***solves*** the given computational problem
- . An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one.

Algorithm Properties

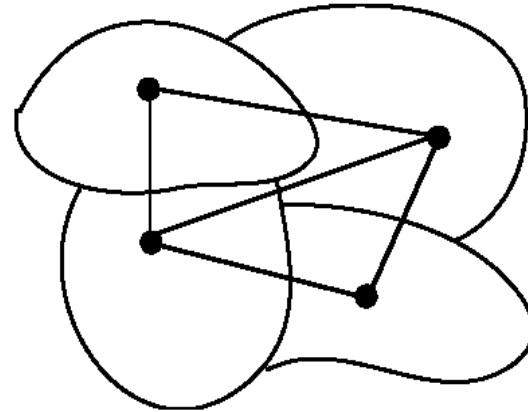
- • An algorithm possesses the following properties:
- 1– It must be correct.
- 2– It must be composed of a series of concrete steps.
- 3– There can be no ambiguity as to which step will be performed next.
- 4– It must be composed of a finite number of steps.
- 5– It must terminate
- A **computer program** is an instance, or concrete representation, for an algorithm in some programming language.

Easy and Hard Problems

- **1- Eulerian Tour vs. Hamiltonian Tour**
- • **Eulerian Tours (Easy)**
 - – INPUT: A graph $G = (V, E)$.
 - – DECIDE: Is there a path that crosses every edge exactly once and returns to its starting point?
- • **Hamiltonian Tours (Hard)**
 - – INPUT: A graph $G = (V, E)$.
 - – DECIDE: Is there a path that visits every vertex exactly once and returns to its starting point?

Easy and Hard Problems

- **2- Map Colorability**
 - **Map 2-colorability (Easy)**
 - INPUT: A graph $G=(V, E)$.
 - DECIDE: Can this map be colored with 2 colors so that no two adjacent countries have the same color?
 - **Map 3-colorability (Hard)**
 - INPUT: A graph $G=(V, E)$.
 - DECIDE: Can this map be colored with 3 colors so that no two adjacent countries have the same color?
 - **Map 4-colorability (Easy)**
 - INPUT: A graph $G=(V, E)$.
 - DECIDE: Can this map be colored with 4 colors so that no two adjacent countries have the same color?



Easy and Hard Problems

- **3- Longest Path vs. Shortest Path**
- **• Longest Path (Hard)**
 - – INPUT: A graph $G = (V, E)$, two vertices u, v of V , and a weighting function on E .
 - – OUTPUT: The longest path between u and v .
 - No one is able to come up with a polynomial time algorithm yet.
- **Shortest Path (Easy)**
 - – INPUT: A graph $G = (V, E)$, two vertices u, v of V , and a weighting function on E .
 - – OUTPUT: The shortest path between u and v .
 - A greedy method will solve this problem easily

Easy and Hard Problems

- **4- Multiplication vs. Factoring**
- • **Multiplication (Easy)**
 - – INPUT: Integers x, y .
 - – OUTPUT: The product $x \times y$.
- • **Factoring (Un-multiplying) (Hard)**
 - – INPUT: An integer n .
 - – OUTPUT: If n is not prime, output two integers x, y such that $1 < x, y < n$ and $x \times y = n$.

Example:

- **Algorithm** *fibonacci* (n):
 - **Input:** a nonnegative integer n .
 - **Output:** fib, the n th term of the fibonacci sequence.
1. **if** $n \geq 1$ **then**
 2. $n = n - 1$
 3. Else
 4. $F1 = 0$
 5. $F2 = 1$
 6. For $i = 2$ to n
 7. $Fib = F1 + F2$
 8. $F1 = F2$
 9. $F2 = Fib$
 10. End for
 11. End if
 12. Return fib

References

- Introduction to algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, 2001
- Algorithms design technique and analysis, M.H. Alsuwaiyel, 2002
- some websites

University of Technology
Computer science Department



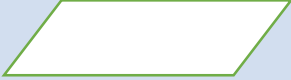


Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Second lecture

Why data structure and algorithms

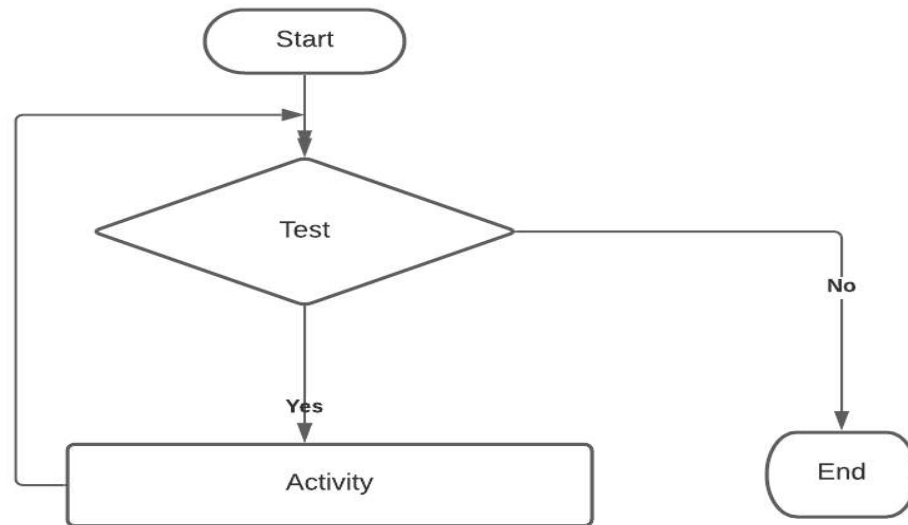
- When programming you are an engineer and each engineer has a bag of tools and tricks and the knowledge of which tool is the right one for a given problem
- Data structure + Algorithms = program
- Flowchart : A graphical representation of an algorithm often used in the design phase of programming to work out the logical flow of a program

Symbols and meaning

Name	Symbol	Use in flowchart
Oval		Is Used in flowchart denotes the beginning or end of the program
Flow line		Denotes the direction of logic flow in a program
Parallelogram		Denotes either input or output operation
Rectangle		Denotes a process to be carried out (e.g. addition)
Diamond		Denotes a decision or branch to be made that a program should continue along one of two routes (e.g. if/then/else)

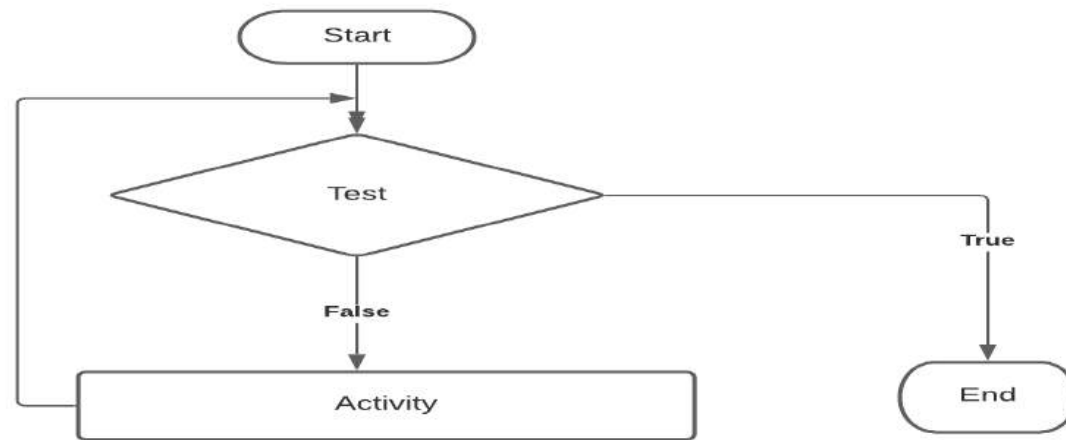
Some control structure

- Control structure like :sequencing ,looping , selecting



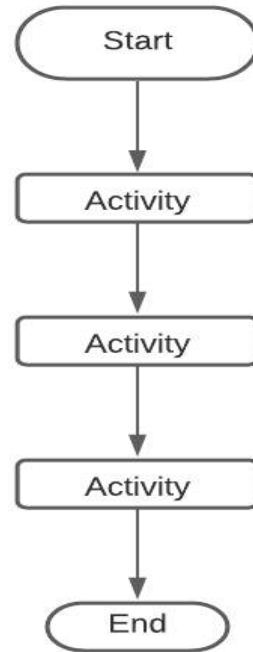
A.Do while loop

Some control structure



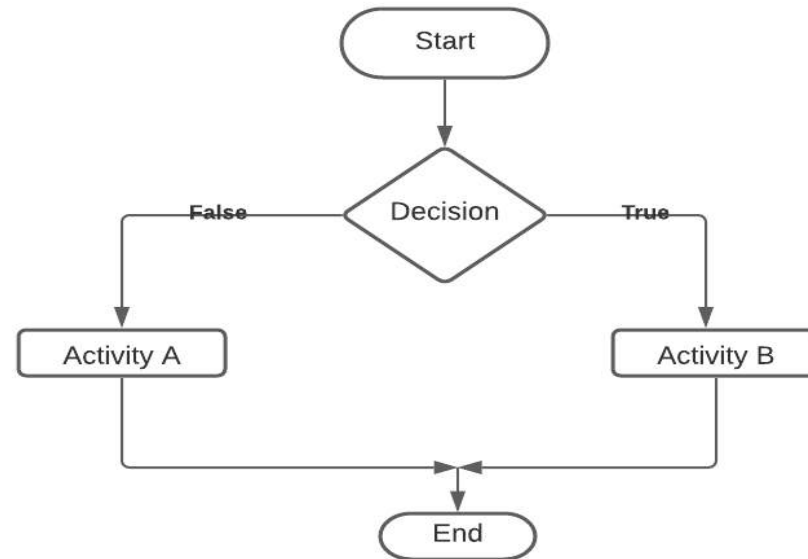
B. do until loop

Some control structure



C. Sequence

Some control structure



D.Selection

Pseudo code

Pseudo code is basically short English phrases used to explain specific tasks within a program's algorithm , it should not contain any specific computer languages.

Why is Pseudo code necessary ?

Ans /because it will save your time and efforts during the construction and testing phase of program development

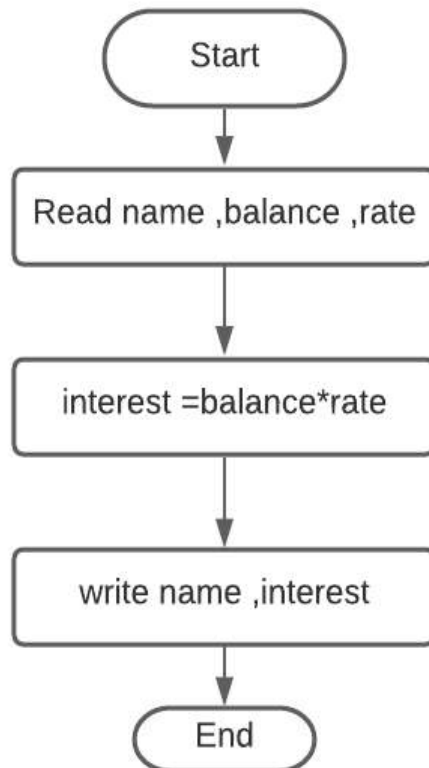
How do I write Pseudo code ?

Ans / consists mainly of executable statement

If you can not write it in Pseudo code you won't be able to write it in C++ or Java

Pseudo code

- A flowchart and it's equivalent Pseudo code



← read name, balance and interest

← compute interest as $\text{balance} * \text{rate}$

← write (display) name and interest

Pseudo code

Example : original program specification

- Write a program that obtains two integer numbers from the user .it will print out the sum of these numbers

Variables required (names and types):

Int1: (integer) to store first no.

Int2: (integer) to store second no.

Sum : (integer) to store the sum of the numbers

Pseudo-code:

Prompt the user to insert first integer int1

Prompt the user to insert second integer int2

Compute the sum of the two inputs

Sum=int1+int2

Display an output prompt that explain the answer

Display the result

Algorithm Analysis

- Algorithms: A clearly specified finite set of instructions a Computer follows to solve a problem
- Algorithm Analysis: a process of determining the amount of time resource, etc. required when executing an algorithm .

Why we need algorithm analysis?

- Writing a working program is not good enough
- The program may be inefficient If the program is run on a large data set then the running time becomes an issue

Algorithm Analysis

Algorithmic efficiency explained in a nutshell

we could say "Algorithm A is twice as fast as Algorithm B" but in fact this sort of statement isn't too meaningful, why ?

Because the proportion can change radically as the number of items are changed Perhaps you increase the number of items by 50%, and now A is three times as fast as B. Or you have half as many items and A and B are now equal

What you need is a comparison that tells how an algorithm's Speed is related to the number of items

Algorithm Analysis

- We only analyse Correct algorithms
- An algorithm is correct,
 - if for every input instance it halts with the correct output
- Incorrect algorithms
 - Might not halt at all on some input instance
 - Might halt with other than the desired answer
- Analyzing an algorithm:
- Predicting the resources that the algorithms requires
- Resources include
 - Computational time (usually most important)

Running Time

- Running Time most algorithms transform input objects into output objects
- The running time of an algorithm typically grows with the input size
- Average case time is often difficult to determine
- we focus on the worst case (upper - bound) running time.
 - Easier to analyze
 - crucial to applications such as games, finance and robotics
 - occurs more often .

Algorithm analysis

Worst / average / Best case

- Worst-case running time of an algorithm
- The longest running time for any input of size n
- An upper bound on the running time for any input guarantee that the algorithm will never take longer

Example: Sort a set of numbers in increasing order and the data is in decreasing order

The worst case can occur fairly often E.g. in searching a database for a particular piece of information

Best case running time

- sort a set of numbers in increasing order and the data is already in increasing order

Algorithm analysis

Average case running time: May be difficult to define what average means

Experimental studies

- write a program implementing the algorithm
- run the program with inputs of varying Size and composition
- Use a method to get an accurate -measure of the actual running time
- Plot the results

limitations of Experiments

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

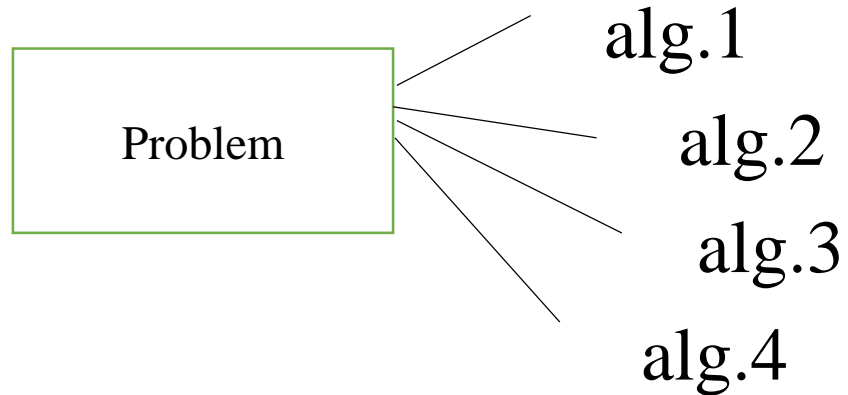
University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Third lecture

Algorithm Analysis

Why do we analyze an algorithm



Analysis of an algorithm helps us to determine which alg. Is the best to solve a problem

Algorithm analysis is based upon two factors :

1- CPU time (time complexity)

2- Main memory space (Space complexity) for a problem we can have more than one algorithm (solution) and we want to decide the best one

Time & Space Complexity

(TC) Time Complexity: Amount of time taken by an algorithm to run till its Completion

(SC) Space Complexity: Amount of space or, memory takes by an alg. to run till its completion

Time Complexity of an alg. Can be Calculated by 2 Methods:

- 1) Posterior Analysis.
- 2) priori Analysis

TC & SC are dependent upon various things such as Hardware , Processor, OS,etc

Time Complexity of an algorithm (equation)

$$T(P) = C(P) + R(P)$$

C(P) depends on : Compiler , software , language of Compiler (thing that is used for Compile time *and decide compile time*)

R(P) depends on : Processor , Hardware , Type of Hardware is use of (thing that is used in Calculating run time and decide run time)

C(p) = Compile time of program

R(p) = run time of program

Compiler is basically software

Processor is : is basically hardware

The most important things:

- language of compiler

- Type of Hardware

we can use any type of them

Priori and Postriori Analysis

- Priori Analysis: Here we determine time complexity of an algorithm by just analyzing the statements inside it rather than running it on any particular system
- Postriori Analysis: Here we determine time complexity of an algorithm after running it on a specific system

Posteriori and Priori Analysis Features

Posteriori Analysis:

- 1) analysis of an alg. after running it on a specific system
- 2) it is dependent on language of Compiler and type of how It gives exact answer
- 3) It gives exact answer
- 4) Answer changes from System to System
- 5) Relative analysis
- 6) Super Computer

Posteriori and Priori Analysis Features

Priori Analysis:

- 1) Analysis of an alg. Prior to running it on any system
- 2) It is independent on language of compiler & type of H.w
- 3) it gives an approximate answer
- 4) Same answer in any system
- 5) absolute Analysis
- 6) Super logic

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Fourth lecture

ASYMPTOTIC NOTATIONS

Algorithmic Complexity:

complexity :is a numerical function $T(n)$ - time versus the input size n .

"Algorithmic Complexity", also called "Running Time" or "Order of Growth", refers to the number of steps a program takes as a function of the size of its inputs.

A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc.

Algorithmic Complexity

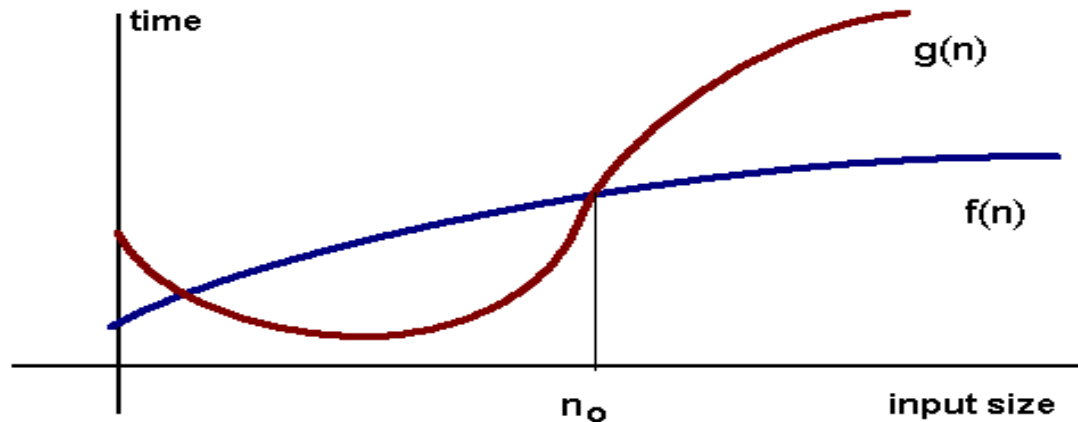
- *Asymptotically*: The way around is to estimate efficiency of each algorithm.
- Example :addition of two integers
- We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. the total computational time
- $T(n) = c * n$

Algorithmic Complexity

- where c is time taken by addition of two bits
- $N = \text{steps}(\text{no of bits})$
- The goal of computational complexity is to classify algorithms according to their performances
- $T(n) = O(n^2)$ says that an algorithm has a quadratic time complexity.

Definition of "big O"

- For any monotonic functions $f(n)$ and $g(n)$ from the positive integers to the positive integers, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that :
- $f(n) \leq c * g(n)$, for all $n \geq n_0$



- Algorithmic complexity is usually expressed in 1 of 2 ways. The first is the way used in lecture - "logarithmic", "linear", etc. The other is called Big-O notation. This is a more mathematical way of expressing running time, and looks more like a function. For example, a "linear" running time can also be expressed as $O(n)$. Similarly, a "logarithmic" running time can be expressed as $O(\log n)$.

Definition of "big O"

- **Examples:**
- $1 = O(n)$
- $n = O(n)$
- $\log(n) = O(n)$
- $2n + 1 = O(n)$
- The "big-O" notation is not symmetric: $n = O(n^2)$ but $n^2 \neq O(n)$.


Definition of "big O"

- **Standard Method to Prove Big-Oh :**
- 1. Choose $n_0 = 1$.
- 2. Assuming $n > 1$, find/derive a C such that
$$F(n)/g(n) \leq c * g(n)/g(n) = c$$
 - This shows that $n > 1$ implies $f(n) \leq C g(n)$.
- Keep in mind:
- $N > 1$ implies $1 < n, n < N^2, N^2 < N^3, \dots$
- • “Increase” numerator to “simplify” fraction.

Definition of "big O"

- **Exercise:** Let us prove $n^2 + 2n + 1 = O(n^2)$.
- Choose $n_0=1$.
- Assuming $n>1$ then
- $f(n)/g(n)=n^2 + 2n + 1/n^2 \leq n^2+2n^2+n^2/n^2=4$
- Choose $C =4$. Note that $2n<2n^2$ and $1<n^2$.
- Thus, $n^2 + 2n + 1$ is $O(n^2)$ because
- $n^2 + 2n + 1 \leq 4n^2$ whenever $n > 1$.
- H.W: Prove that $3n + 7 = O(n)$

Constant Time: $O(1)$

- An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:
- Given two numbers*, report the sum
- • Given a list, report the first element
- •  array: accessing any element

Linear Time: $O(n)$

- An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:
 - array: linear search, traversing, find minimum
 - ● Given a list of words, say each item of a list
 - ● Given a list of numbers, add each pair of numbers together
 - (item 1 + item 2, item 3 + item 4, etc.)

Logarithmic Time: $O(\log n)$

- An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size.
- Example:
- binary search
- Note, $\log(n) < n$. Algorithms that run in $O(\log n)$ does not use the whole input.

Quadratic Time: $O(n^2)$

- An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size. A typical pattern of quadratic time algorithms is performing a linear-time operation on each item of the input (n steps per item * n items = n^2 steps). Examples:
 - Compare each item of a list against all the other items in the list
- Examples :
 - bubble sort, selection sort, insertion sort

Cubic-Time Algorithms

- Cubic-Time Algorithms - $O(n^3)$ A cubic-time algorithm is one that takes a number of steps
- proportional to n^3 . In other words, if the input doubles, the number of steps is multiplied by 8. Similarly to the quadratic case, this could be the result of applying an n^2 algorithm to n
- items, or applying a linear algorithm to n^2 items.

Exponential-Time Algorithms - $O(2^n)$

- An exponential-time algorithm is one that takes time proportional to 2^n . In other words, if the size of the input increases by one, the number of steps doubles. Note that logarithms and exponents are inverses of each other. Algorithms in this category are often considered too slow to be practical, especially if the input is typically large. Examples:
- Generating *fibonacci series* .

Definition of "big Omega"

- To describe lower bounds we use the big-omega notation $f(n)=\Omega(g(n))$ usually defined by saying for some constant $c>0$ and all large enough n ,
- $f(n) \geq c g(n)$. This has a nice symmetry property, $f(n)=O(g(n))$ if $g(n)=\Omega(f(n))$.
- **Examples**
- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $2n + 1 = \Omega(n)$

Definition of "big Theta"

- To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- **Examples**
- $2n = \Theta(n)$
- $n^2 + 2n + 1 = \Theta(n^2)$

Analysis of Algorithms

- The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:
- 1- the *worst-case runtime complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size a .
- 2- the *best-case runtime complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size a .
- 3- the *average case runtime complexity* of the algorithm is the function defined by an average number of steps taken on any instance of size a .

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Fifth lecture

COMPLEXITY EXAMPLES

Running Time Functions

Most algorithms have a primary parameter N , usually the number of data items to be processed, which affects the running time most significantly.

The parameter N might be:

- The degree of a polynomial
- The size of a file to be sorted or searched
- the number of nodes in a graph

Running Time Functions

- 1. **(1)** Most instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is constant
- **(log N)** When the running time of a program is logarithmic, the program gets slightly slower as N grows. This running time commonly occurs in programs which solve a big problem by transforming it into a smaller problem by cutting the size by some constant fraction.
- 3. **(N)** When the running time of a program is linear, it generally is the case that a small amount of processing is done on each input element. When N is a million, then so is the running time.

Running Time Functions

- 4. (**$N \log N$**) This running time arises in algorithms which solve a problem by breaking it up into smaller sub problems, solving them independently, and then combining the solutions
- 5. (**N^2**) When the running time of an algorithm is *quadratic*, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms which process all pairs of data items (perhaps in a double nested loop). Whenever N doubles, the running time increases fourfold.

Running Time Functions

- **6. (N^3)** Similarly, an algorithm which processes triples of data items (perhaps in a triple-nested loop) has a *cubic* running time and is practical for use only on small problems. Whenever N doubles, the running time increases eightfold.
- **7. (2^N)** Few algorithms with *exponential* running time are likely to be appropriate for practical use, though such algorithms arise naturally as “brute-force” solutions to problems. Whenever N doubles, the running time squares.

Running Time Functions

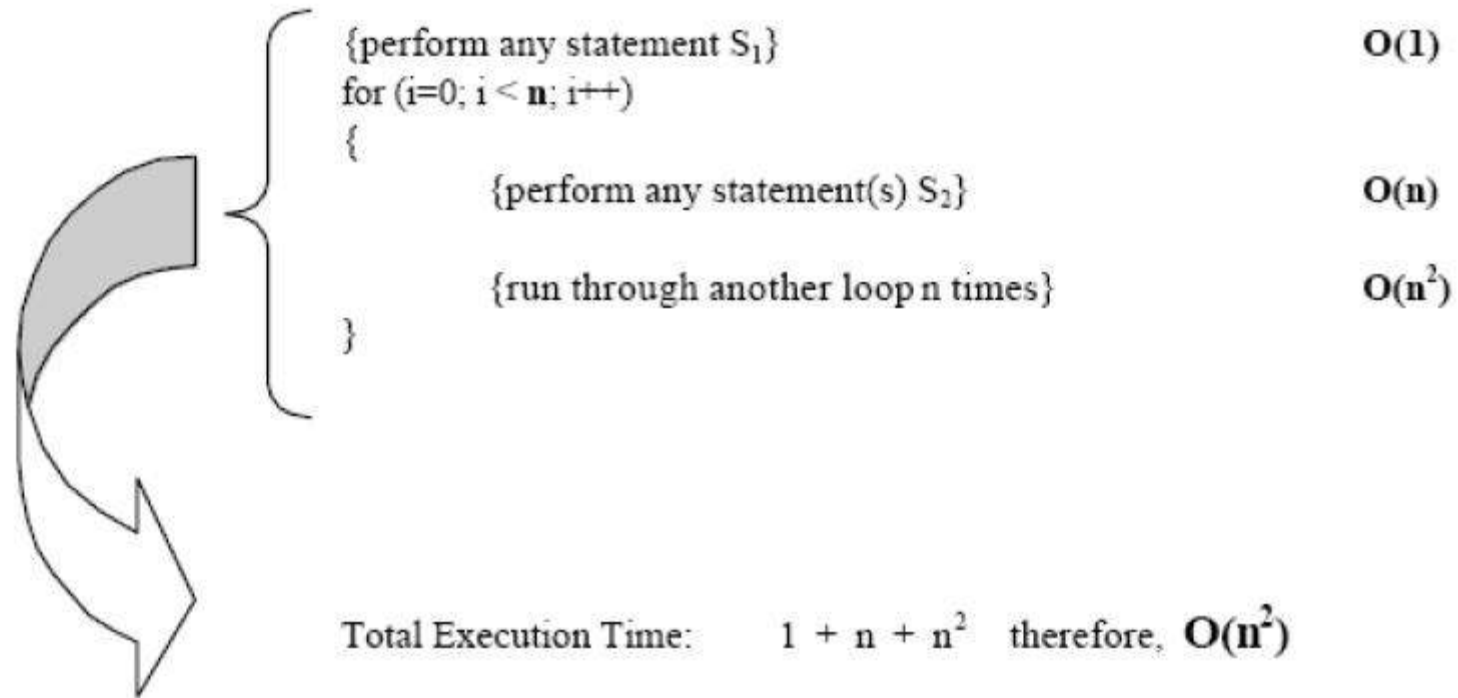
- **The O - Examples**

- $f(n) = 2n + 3$; $f(n) = O(n)$
- $f(n) = 6n^2 + 235$; $f(n) = O(n^2)$
- $f(n) = 6n + 567\ln(n)$; $f(n) = O(n)$
- $f(n) = 6n \times \ln(n)$; $f(n) = O(n)$
- $f(n) = 2\exp(n) + n\ln(n) + 456n$; $f(n) = O(\exp(n))$
- if we have a polynomial
- **$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$**

its growth is of the order n^k :

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

Example:



Big O and control structure: the following table can be used to assist in estimating the big O performance of algorithm

Control structure	Running time
Single assignment statement	$O(1)$
Simple expression	$O(1)$
The sequence Statement1 Statement 2	$\text{Max } [O(s1), O(s2), O(s3), \dots, O(s_n),]$
If condition : st1 Else: st2	$\text{Max } [O(\text{cond1}), O(\text{cond2})]$ worst case
For I in lista : statement	$O(n*s) = O(n) * O(s)$

Space complexity

- The (space) complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution. This number is computed with respect to the size n of the input data.
- **Note:**
- We thus make the assumption that each elementary object needs the same amount of space.
- The difference between space complexity and time complexity is that space can be reused.
- Amount of computer memory required during the program execution, as a function of the input size

How to Determine Complexities

- In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.
- **1.** Sequence of statements
- statement 1;----- $O(1)$
- statement 2;
- ...
- statement k;
- Total time = time (statement 1) + time (statement 2) + ... + time (statement k)
- If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: $O(1)$. In the following examples, assume the statements are simple unless noted otherwise.

How to Determine Complexities

- **2-** if-then-else statements
- if (condition) :
 Sequence of statements 1
- else :
 sequence of statements 2
- Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: $\max(\text{time}(\text{sequence 1}), \text{time}(\text{sequence 2}))$. For example, if sequence 1 is $O(N)$ and sequence 2 is $O(1)$ the worst-case time for the whole if-then-else statement would be $O(N)$.

How to Determine Complexities

- **3-** for loops

for i in array :

 sequence of statements

- The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are $O(1)$, the total time for the for loop is $N * O(1)$, which is $O(N)$ overall.

How to Determine Complexities

- 4- Nested loops
- First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

for i in arrayA :

- for j in arrayB :
 sequence of statements

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of $N * M$ times $O(N*M)$

If the stopping condition of the inner loop is $j < N$ instead of $j < M$ (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

$O(2^n)$ –exponential time

Def Fibonacci(n) :

 if $n \leq 1$:

 return n

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

- ***Exercise***

- The following code is to find the biggest number in an array

- Def findBgstNum(ArrayS):

```
    bgstNum=ArrayS[0]-----O(1)
```

```
    for l in range(1,len(ArrayS)):-O(n)
```

```
        if ArrayS[i]> bgstNum:-O(1)
```

```
            bgstNum=ArrayS[i]-----O(1)
```

```
    print(bgstNum)-----O(1)
```

So time complexity = $O(n)$

- Ex:/ what is the run time of the below code ?

- Def PS(arrayA):

```
sum=0-----O(1)
product=1-----O(1)
for l in arrayA:-----O(n)
    sum+=1-----O(1)
for l in arrayA:-----O(n)
    product *=l-----O(1)
print("sum="+str(sum) + ", product =" + str(product))-----O(1)
```

time complexity = $O(n)$

- Def printarray(arrayA,arrayB):
 - for i in range(1,len(arrayA)):-----O(a)
 - for j in range(1,len(arrayB)):-----O(b)
 - if arrayA[i] > arrayB[j] :-----O(1)
 - print(str(arrayA[i])+',' +str(arrayB[i]))-----O(1)

a =length of array a

b=length of array b

time complexity = $O(a*b)$

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Sixth lecture

ALGORITHM TYPES AND CLASSIFICATIONS

Different types of algorithms

Every algorithm falls under a certain class. Basically they are

- 1) Brute force
- 2) Divide and conquer
- 3) Decrease and conquer
- 4) Dynamic programming
- 5) Greedy algorithm
- 6) Transform and conquer
- 7) Backtracking algorithm and so on

ALGORITHM TYPES AND CLASSIFICATIONS

- ***Brute force algorithm***

Brute force implies using the definition to solve the problem in a straightforward manner. Brute force algorithms are usually the easiest to implement, but the disadvantage of solving a problem by brute force is that it is usually very slow and can be applied only to problems where input size is small.

- ***Divide and conquer algorithm***

In divide and conquer method, we divide the size of a problem by a constant factor in each iteration. This means we have to process lesser and lesser part of the original problem in each iteration. Some of the fastest algorithms belong to this class. Divide and conquer algorithms have logarithmic runtime.

ALGORITHM TYPES AND CLASSIFICATIONS

- ***Decrease and conquer algorithm***

This kind of problem is same as divide and conquer, except, here we are decreasing the problem in each iteration by a constant size instead of constant factor.

- ***Dynamic programming***

Sometimes, a solution to the given instance of problem depends on the solution to smaller instance of sub-problems. It exhibits the property of overlapping sub-problems. Hence, to solve a problem we may have to recompute same values again and again for smaller sub-problems. Hence, computing cycles are wasted.

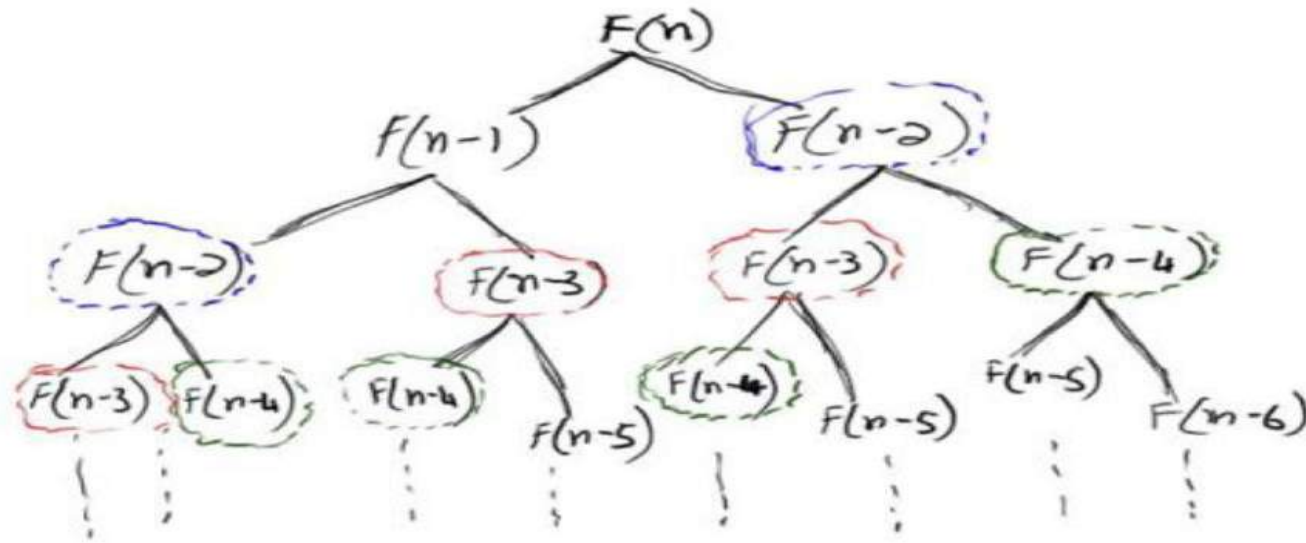
To remedy this, we can use dynamic programming technique. Basically, in dynamic programming, we “remember” the result of each sub-problem. Whenever we need it, we will use that value instead of recomputing it again and again.

we are using more space to hold the computed values to increase the execution speed drastically.

ALGORITHM TYPES AND CLASSIFICATIONS

A good example for a problem that has overlapping sub-problem is the relation for Nth Fibonacci number.

It is defined as $F(n) = F(n-1) + F(n-2)$.



ALGORITHM TYPES AND CLASSIFICATIONS

The solution to this is to store each value as we compute it and retrieve it directly instead of re calculating it. This transforms the exponential time algorithm into a linear time algorithm.

Hence, dynamic programming is a very important technique to speed up the problems that have overlapping sub problems.

ALGORITHM TYPES AND CLASSIFICATIONS

- ***Greedy algorithm***

For many problems, making greedy choices leads to an optimal solution. These algorithms are applicable to optimization problems. In a greedy algorithm, in each step, we will make a locally optimum solution such that it will lead to a globally optimal solution. Once a choice is made, we cannot retract it in later stages. Proving the correctness of a greedy algorithm is very important, since not all greedy algorithms lead to globally optimum solution.

For ex- consider the problem where you are given coins of certain denomination and asked to construct certain amount of money in minimum number of coins. Let the coins be of 1, 5, 10, 20 cents. If we want change for 36 cents, we select the largest possible coin first (greedy choice). According to this process, we select the coins as follows-

$$20 \quad 36 - 20 = 16$$

$$20 + 10 \quad 16 - 10 = 6$$

$$20 + 10 + 5 \quad 6 - 5 = 1$$

$$20 + 10 + 5 + 1 = 36.$$

For coins of given denomination, the greedy algorithm always works. But in general this is not true.

Consider the denomination as 1, 3, 4 cents. To make 6 cents, according to greedy algorithm the selected coins are $4 + 1 + 1 = 6$

But, the minimum coins needed are only 2 ($3 + 3$)

Hence, greedy algorithm is not the correct approach to solve the 'change making' problem.

ALGORITHM TYPES AND CLASSIFICATIONS

Transform and conquer algorithm

Sometimes it is very hard or not so apparent as to how to arrive at a solution for a particular problem. In this case, it is easier to transform the problem into something that we recognize, and then try to solve that problem to arrive at the solution. Consider the problem of finding LCM (least common multiple) of a number. Brute force approach of trying every number and seeing if it is the LCM is not the best approach. Instead, we can find the GCD (greater common divisor) of the problem using a very fast algorithm known as Euclid's algorithm and then use that result to find the LCM as

$$\text{LCM} (a , b) = (a * b) / \text{GCD} (a , b).$$

ALGORITHM TYPES AND CLASSIFICATIONS

- GCD in Euclidean
- Ex: compute gcd for (48,18) using Euclidean
- Sol: divide 48 by 18 as
- $(a=48 / b=18) = 2$ remainder 12 ($48 \bmod 18$)
- $(a=18 / b=12) = 1$ remainder 6 ($18 \bmod 12$)
- $12 / 6 = 2$ remainder 0
- When the remainder is 0 we find the GCD and its 6
- $\text{GCD}(a,b) = ?$ Write the relation as H.W. in recursion function

ALGORITHM TYPES AND CLASSIFICATIONS

- ***Backtracking algorithm:***

Backtracking approach is very similar to brute force approach. But the difference between backtracking and brute force is that, in brute force approach, we are generating every possible combination of solution and testing if it is a valid solution. Whereas, in backtracking, each time you generate a solution, you are testing if it satisfies all condition, and only then we continue generating subsequent solutions, else we will backtrack and go on a different path of finding solution.

A famous example to this problem is the N Queens problem. According to the problem, we are given a $N \times N$ sized chessboard. We have to place N queens on the chessboard such that no queens are under attack from any other queen.

An advantage of this method over brute force is that the numbers of candidates generated are very less compared to brute force approach. Hence we can isolate valid solutions quickly.

Lecture (7)

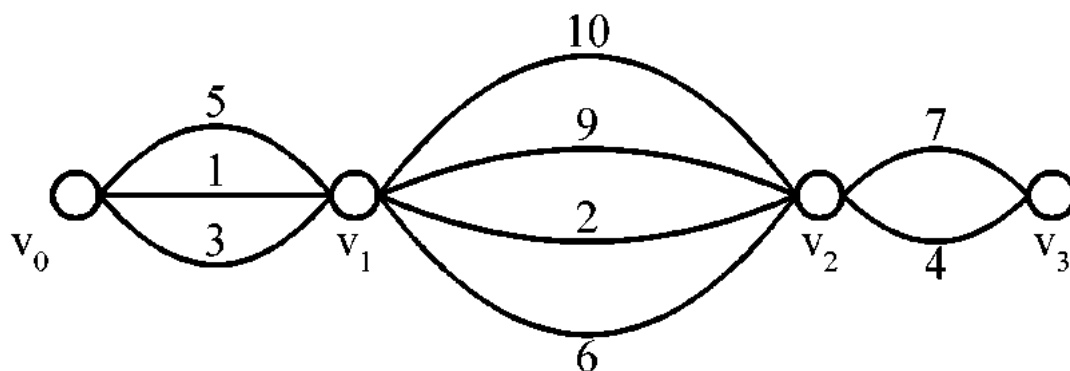
GREEDY ALGORITHM

Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.

Only a few optimization problems can be solved by the greedy method.

Shortest paths on a special graph

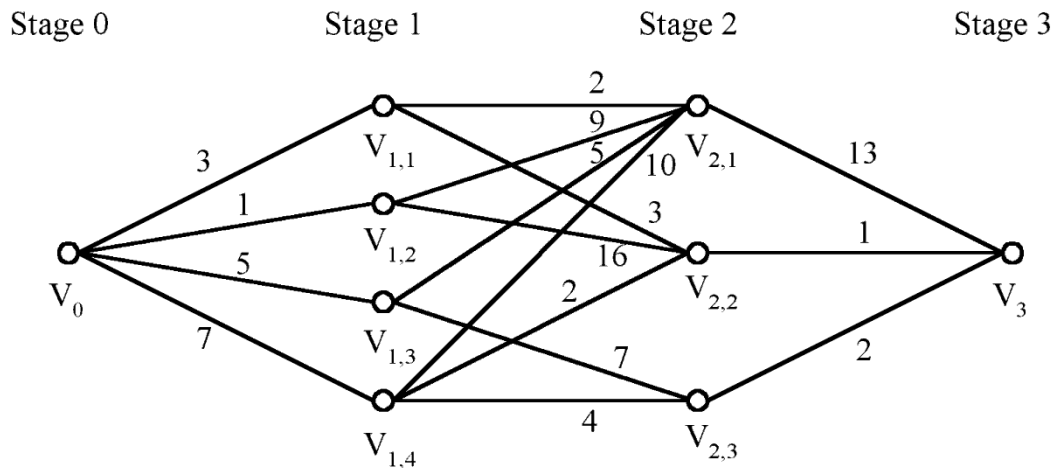
- Problem: Find a shortest path from v_0 to v_3 .



- The greedy method can solve this problem.
- The shortest path: $1 + 2 + 4 = 7$.

Shortest paths on a multi-stage graph

- **Problem:** Find a shortest path from v_0 to v_3 in the multi-stage graph.



- Greedy method: $v_0v_{1,2}v_{2,1}v_3 = 23$

- Optimal: $v_0v_{1,1}v_{2,2}v_3 = 7$

The greedy method does not work.

Solution of the above problem

- $d_{\min}(i,j)$: minimum distance between i and j .

$$d_{\min}(v_0, v_3) = \min \begin{cases} 3 + d_{\min}(v_{1,1}, v_3) \\ 1 + d_{\min}(v_{1,2}, v_3) \\ 5 + d_{\min}(v_{1,3}, v_3) \\ 7 + d_{\min}(v_{1,4}, v_3) \end{cases}$$

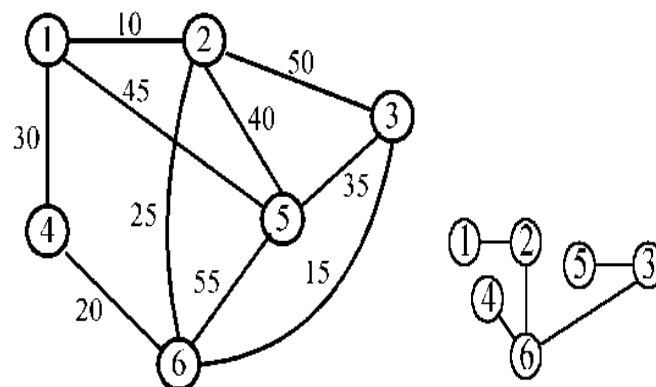
- This problem can be solved by the dynamic programming method.

Minimum spanning trees (MST)

- It may be defined on Euclidean space points or on a graph.
- $G = (V, E)$: weighted connected undirected graph
- Spanning tree : $S = (V, T)$, $T \subseteq E$, undirected tree
- Minimum spanning tree(MST) : a spanning tree with the smallest total weight.

An example of MST

- A graph and one of its minimum costs spanning tree



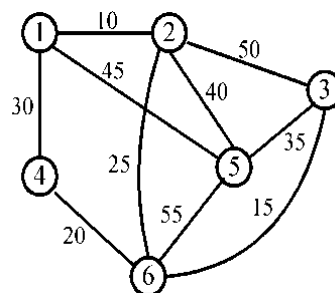
Kruskal's algorithm for finding MST:

Step 1: Sort all edges into nondecreasing order.

Step 2: Add the next smallest weight edge to the forest if it will not cause a cycle.

Step 3: Stop if $n-1$ edges. Otherwise, go to Step2.

An example for Kruskal's algorithm:



<u>Edge</u>	<u>Cost</u>	<u>Spanning Forest</u>
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

Prim's algorithm for finding MST:

Step 1: $x \in V$, Let $A = \{x\}$, $B = V - \{x\}$.

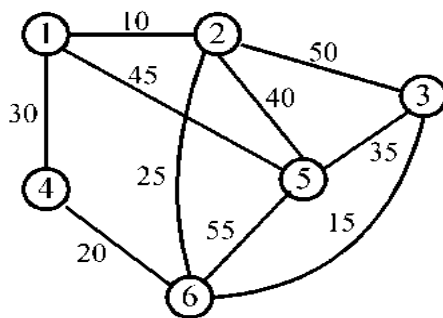
Step 2: Select $(u, v) \in E$, $u \in A$, $v \in B$ such that (u, v) has the smallest weight between A and B .

Step 3: Put (u, v) in the tree. $A = A \cup \{v\}$, $B = B - \{v\}$

Step 4: If $B = \emptyset$, stop; otherwise, go to Step 2.

Time complexity : $O(n^2)$, $n = |V|$.

An example for Prim's algorithm



Edge	Cost
------	------

(1,2)	10
-------	----

(2,6)	25
-------	----

(3,6)	15
-------	----

(6,4)	20
-------	----

(3,5)	35
-------	----

Spanning tree

① — ②

① — ②

⑥

① — ②

⑥

③

① — ②

④

⑥

③

① — ②

④

⑥

⑤

③

What are difference between Prim's algorithm and Kruskal's algorithm for finding the minimum spanning tree of a graph :

Prim's method starts with one vertex of a graph as your tree, and adds the smallest edge that grows your tree by one more vertex. Kruskal starts with all of the vertices of a graph as a forest, and adds the smallest edge that joins two trees in the forest.

Prim's method is better when

- You can only concentrate on one tree at a time
- You can concentrate on only a few edges at a time

Kruskal's method is better when

- You can look at all of the edges at once
- You can hold all of the vertices at once
- You can hold a forest, not just one tree

Basically, Kruskal's method is more time-saving (you can order the edges by weight and burn through them fast), while Prim's method is more space-saving (you only hold one tree, and only look at edges that connect to vertices in your tree).

Lecture (8)

DIVIDE AND CONQUER ALGORITHM

The *divide and conquer* strategy solves a problem by :

1. Breaking into *sub problems* that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub problems.
3. Appropriately combining their answers.

Two types of sorting algorithms which are based on this divide and conquer algorithm:

1. **Quick sort:** Quick sort also uses few comparisons. Like heap sort it can sort "in place" by moving data in an array.

2. **Merge sort:** Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

Quick sort

Quick sort is one of the fastest and simplest sorting algorithms, which uses partitioning as its main idea. It works recursively by a divide-and-conquer strategy.

Example: Pivot about 10.

17 12 6 19 23 8 5 10 - before

6 8 5 10 23 19 12 17 – after

Partitioning places all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array. The pivot fits in the slot between them.

Example: pivot about 10

| 17 12 6 19 23 8 5 | 10

| 12 6 19 23 8 5 | 17

| 6 19 23 8 5 | 12 17

6 | 19 23 8 5 | 12 17

6 | 23 8 5 | 19 12 17

6 | 8 5 | 23 19 12 17

6 8 | 5 | 23 19 12 17

6 8 5 || 23 19 12 17

6 8 5 10 23 19 12 17

Note that the pivot element ends up in the correct place in the total order.

Quick sort algorithm:

Algorithm quicksort(q)

var list less, pivotList, greater

if $\text{length}(q) \leq 1$

return q

else

select a pivot value from q

for each x in q except the pivot element

if $x < \text{pivot}$ then add x to less

if $x \geq \text{pivot}$ then add x to greater

add pivot to pivotList

return concatenate(quicksort(less), pivotList, quicksort(greater))

Time and Space Complexity of Quick Sort

Time complexity = $O(n \log n)$.

Space complexity = $O(n)$

When to use Quick sort :

- When average expected time is $O(n \log n)$

When to avoid Quick sort :

- When space is concerned
- When you need stable sort

Merge Sort

Merge Sort is a $O(n \log n)$ sorting algorithm. It is easy to implement merge sort such that it is stable meaning it preserves the input order of equal elements in the sorted output. It is a comparison sort.

Example:

A												
1	2	3	4	5	6	7	8	9	10	11	12	13
[31 23 01 17 19 28 09 03 13 15 22 08 29]												
1 : 13												
1 : 7							8 : 13					
1 : 4				5 : 7			8 : 10			11 : 13		
1 : 2	3 : 4	5 : 6	7 : 7	8 : 9			10 : 10	11 : 12	13 : 13			
31 23	01 17	19 28	09	03 13			15	22 08	29			
23 31	01 17	19 28	09	03 13			15	08 22	29			
01 17 23 31		09 19 28		03 13 15			08 22 29					
01 09 17 19 23 28 31							03 08 13 15 22 29					
01 03 08 09 13 15 17 19 22 23 29 31												

Merge sort algorithm:

Algorithm: mergesort (A, left, right)

Input: An array A of numbers , the bounds left and

Right for the elements to be sorted

Output: A [left...right] is sorted

Process:

If (left<right) { /*we have at least two elements to sort*/

Mid= [(left +right)/2]

Mergesort (A, left, mid) /*now A [left....mid] is sorted*/

mergesort(A,mid+1,right) /*now A [mid+1....right] is sorted */

Merge (A, left, mid, right) /* merge A [left ...mid] with A [mid+1...right]*/

}

Merging:

Merge(array A, int left, int mid, int right)

```
{  
    array B[left..right]  
    i = k = left           // initialize pointers  
    j = mid+1  
    while (i <= mid and j <= right) {    // while both subarrays are nonempty  
        if (A[i] <= A[j]) B[k++] = A[i++] // copy from left subarray  
        else               B[k++] = A[j++] // copy from right subarray  
    }  
    while (i <= mid) B[k++] = A[i++]    // copy any leftover to B  
    while (j <= right) B[k++] = A[j++]  
    for i = left to right A[i] = B[i]    // copy B back to A  
}
```

Time and Space Complexity of Merge Sort

Time complexity = $O(n \log n)$.

Space complexity = $O(n)$

When to use Merge sort :

- When average expected time is $O(n \log n)$
- When you need stable sort

When to avoid Merge sort :

- When space is concerned

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq
Class: Third
Branch :A.I
1st course 2020
Ninth lecture

Dynamic Programming

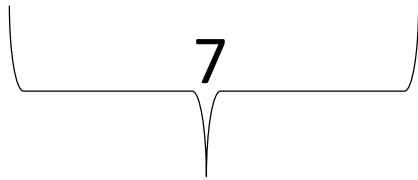
What is dynamic Programming ?

Definition: Dynamic programming (DP) is an algorithmic technique for Solving an optimization problem by breaking it down into simpler Subproblems and utilizing the fact that the optimal solution to the overall Problem depends upon the optimal solution to its subproblem

Simple example

$$1+1+1+1+1+1+1 = 7$$

$$1+1+1+1+1+1+1+2=9$$



Dynamic Programming

- The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored, the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.

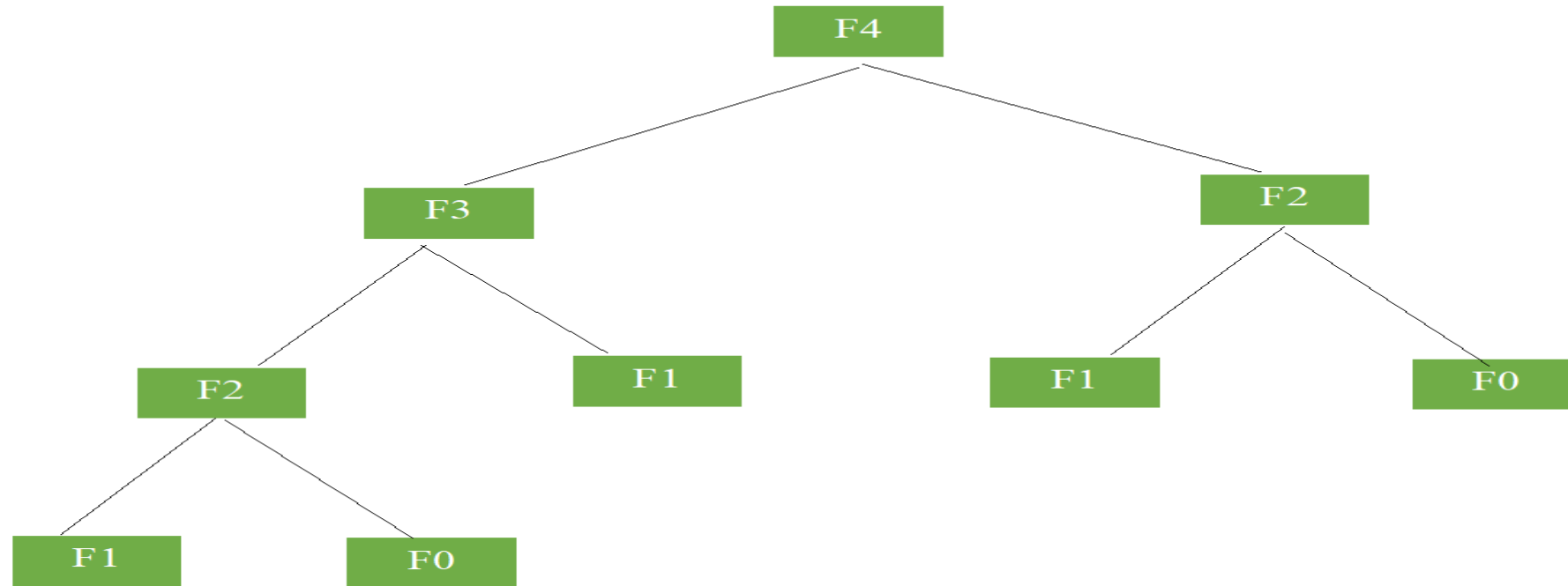
Dynamic Programming

- There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub problems. If a problem can be solved by combining optimal solutions to *non-overlapping* sub problems, the strategy is called "divide and conquer". This is why merge sort and quick sort are not classified as dynamic programming problems.
- *Optimal substructure* means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub problems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion

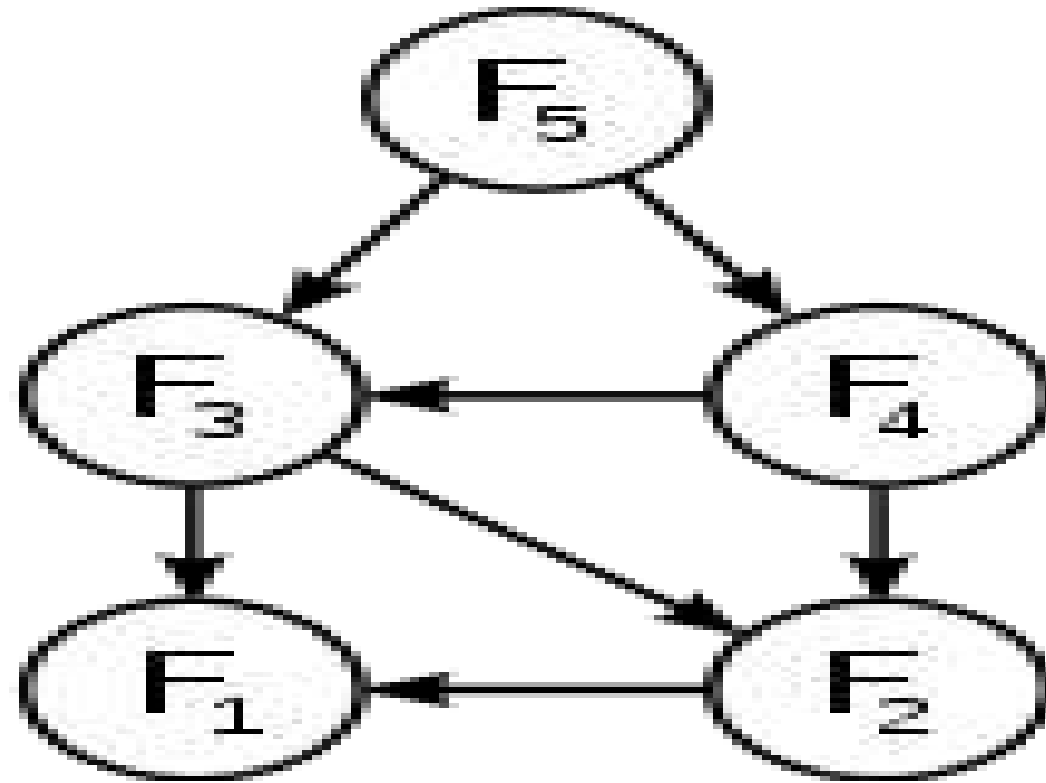
Dynamic Programming

- Ex: fibonacci
- $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
- *Overlapping* subproblems means that the space of subproblems must be small (subproblems are smaller version of the original problem), that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems.

Dynamic Programming



Dynamic Programming



Dynamic Programming

- Dynamic Programming is the optimization of Divide and conquer
- In DP we divide the main problem into smaller Problem and then solving this problems as in divide and Conquer but the difference if the smaller problems come again and again we use the solution instead of solving them multiple times and this method increase the efficiency of the algorithm .

Dynamic Programming

Top down with memorization.

Definition. Solving the bigger Problem by recursively finding the solution

- to smaller Subproblems. Whenever we solve a subproblem we Cache it's result So that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of already solved Subproblems is called Memoization The main idea we start from the top which is the main problem then Continue to the down by dividing the main problem into subproblem and while solving these subproblem we store the results so that if we face the same problem we are not solving it multiple times just use the result that we got before

Dynamic Programming

Ex :Fibonacci 0,1 1,2,3,5,8,13, 21, 34,55 ,

$\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$, in case of Divide & Conquer we use the following algorithm

fib (N):

If $n < 1$ return error message

If $n = 1$ return 0

If $n = 2$ return 1

Else

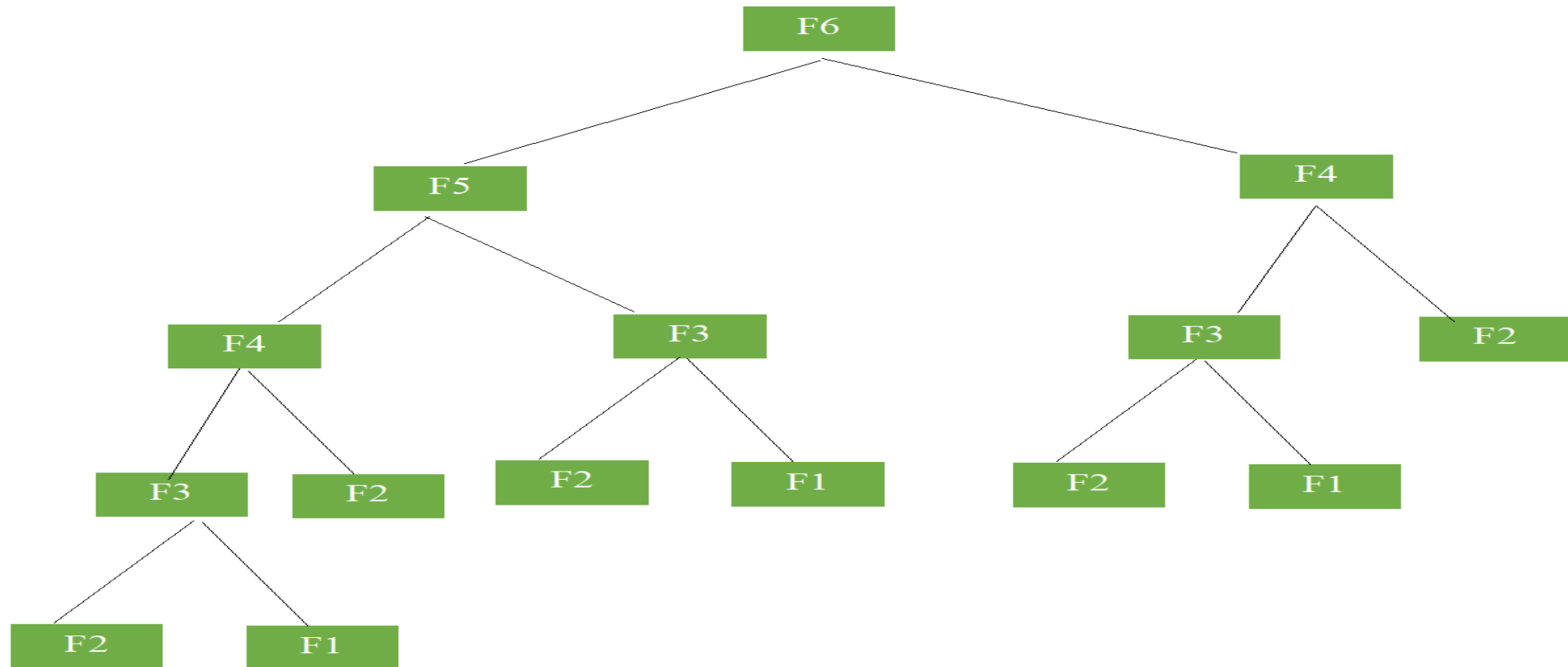
Return $\text{Fib}(N-1) + \text{Fib}(N-2)$

We have 2 recursive Calls

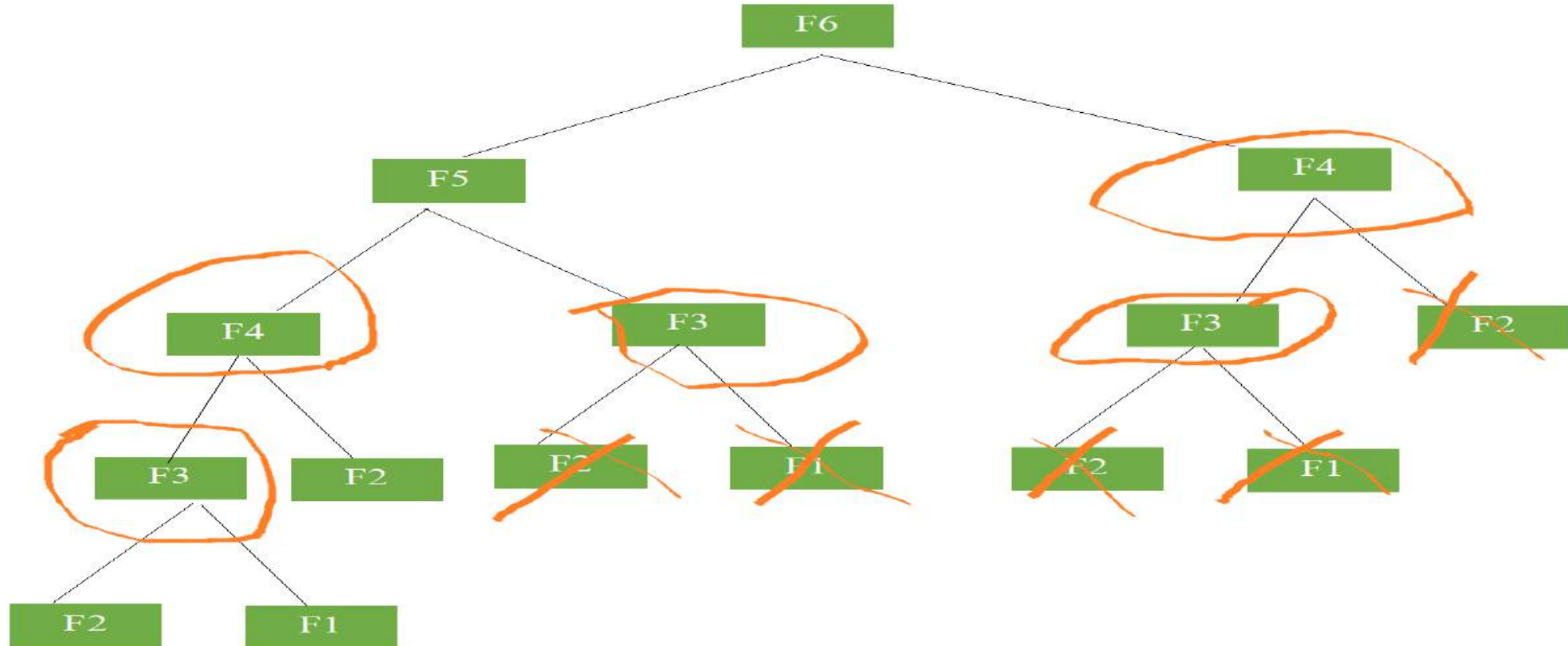
Time Complexity - $O(c^n)$ exponential

Space Complexity $O(n)$

Dynamic Programming



Dynamic Programming



Dynamic Programming

- Fibonacci in dynamic programming method

fib (N):

If $n < 1$ return error message

If $n = 1$ return 0

If $n = 2$ return 1

If not n in memo :

memo[n] = Fib (N-1, memo) + Fib(N-2, memo)

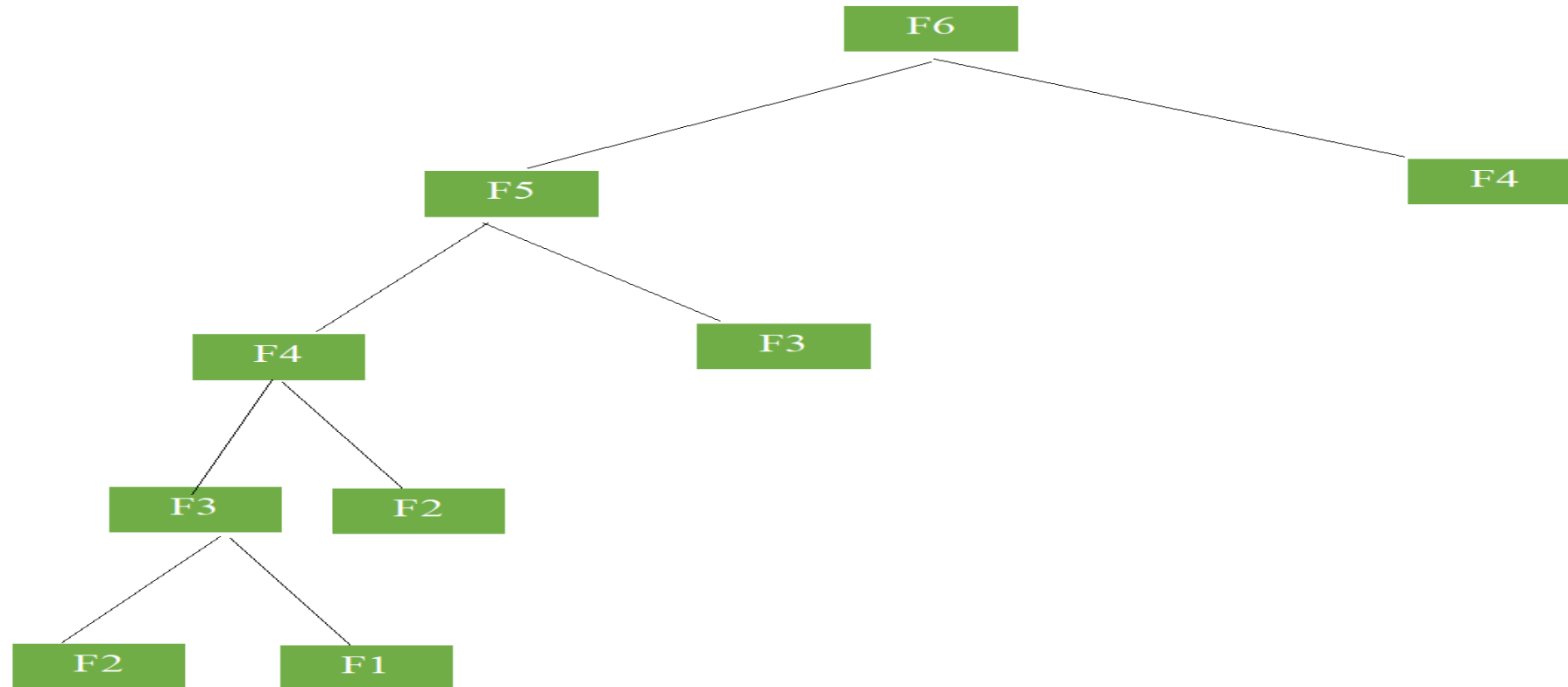
return memo[n]

Time complexity = $O(n)$

Space complexity = $O(n)$

So the previous tree is reduced to the following tree

Dynamic Programming



Dynamic Programming

- Filling memo in top down approach

F1	F2	F3	F4	F5	F6
0	1				$F4+f5$

F1	F2	F3	F4	F5	F6
0	1			$F3+F4$	$F4+F5$

F1	F2	F3	F4	F5	F6
0	1		$F2+F3$	$F3+F4$	$F4+f5$

F1	F2	F3	F4	F5	F6
0	1	$F1+F2$	$F2+F3$	$F3+F4$	$F4+F5$

F1	F2	F3	F4	F5	F6
0	1	1	$F2+F3$	$F3+F4$	$F4+f5$

F1	F2	F3	F4	F5	F6
0	1	1	2	$F3+F4$	$F4+F5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	$F4+f5$

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5

Dynamic Programming

Bottom up with Tabulation:

Tabulation is the opposite of the top down approach and avoids recursion. In this approach, we solve the problem "bottom-ups (ie. by solving all the related subproblems first). This is done by filling up a table. Based on the results in the table, the solution to the top /original

Dynamic Programming

- Filling memo in top bottom-up approach (solving all related sub-problems first)

F1	F2	F3	F4	F5	F6
0	1	F1+F2			

F1	F2	F3	F4	F5	F6
0	1	1			

F1	F2	F3	F4	F5	F6
0	1	1	F2+F3		

F1	F2	F3	F4	F5	F6
0	1	1	2		

F1	F2	F3	F4	F5	F6
0	1	1	2	F3+F4	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	

F1	F2	F3	F4	F5	F6
0	1	1	2	3	F4+f5

F1	F2	F3	F4	F5	F6
0	1	1	2	3	5

Dynamic Programming

- Top down Vs. bottom up

	Top down	Bottom up
Easiness	Easy to come up with solution as it is extensive of divide and conquer	Difficult to come up with solution
Runtime	Slow	Fast
Space efficiency	Unnecessary use of stack place	Stack is not used
When to use	Need a quick solution	Need an efficient solution

Longest common subsequence(LCS)

- A simple problem that illustrate the underlying principle of dynamic programming is the following problem given two strings A and B of lengths N and M find the longest common substring for both of them for the following example find the LCS between A="AABCAS"

And B = "CABSSBA"

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0					
A	0						
B	0						
S	0						
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0			
A	0						
B	0						
S	0						
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0						
B	0						
S	0						
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0						
S	0						
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0	1	1	2	2	2	2
S	0						
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0	1	1	2	2	2	2
S	0	1	1	2	2	2	3
S	0						
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0	1	1	2	2	2	2
S	0	1	1	2	2	2	3
S	0	1	1	2	2	2	3
B	0						
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0	1	1	2	2	2	2
S	0	1	1	2	2	2	3
S	0	1	1	2	2	2	3
B	0	1	1	2	2	2	3
A	0						

Longest common subsequence(LCS)

	0	A	A	B	C	A	S
0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1
A	0	1	1	1	1	2	2
B	0	1	1	2	2	2	2
S	0	1	1	2	2	2	3
S	0	1	1	2	2	2	3
B	0	1	1	2	2	2	3
A	0	1	2	2	2	3	3

University of Technology
Computer science Department

Algorithms Complexity

Lecturer : Farah Tawfiq

Class: Third

Branch :A.I

1st course 2020

Tenth lecture

Network Flow Problem

1. Introduction

The network flow problem is an example of a beautiful theoretical subject that has many important applications. It also has generated algorithmic questions that have been in a state of extremely rapid development in the past 20 years. Altogether, the fastest algorithms that are now known for the problem are much faster, and some are much simpler, than the ones that were in use a short time ago, but it is still unclear how close to the ‘ultimate’ algorithm we are.

Definition. *A network is an edge-capacitated directed graph, with two distinguished vertices called the source and the sink.*

capacity :in a network there is a positive real number associated with each directed edge e of the digraph .

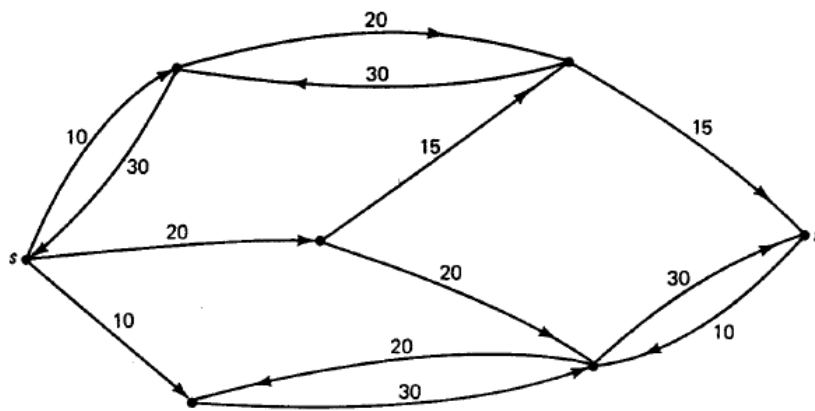


Figure (1) Network

Definition. A flow in a network \mathbf{X} is a function f that assigns to each edge e of the network a real number

$f(e)$, in such a way that

(1) For each edge e we have $0 \leq f(e) \leq \text{cap}(e)$ and

(2) For each vertex v other than the source and the sink, it is true that

$$\sum_{\text{Init}(e)=v} f(e) = \sum_{\text{Term}(e)=v} f(e).$$

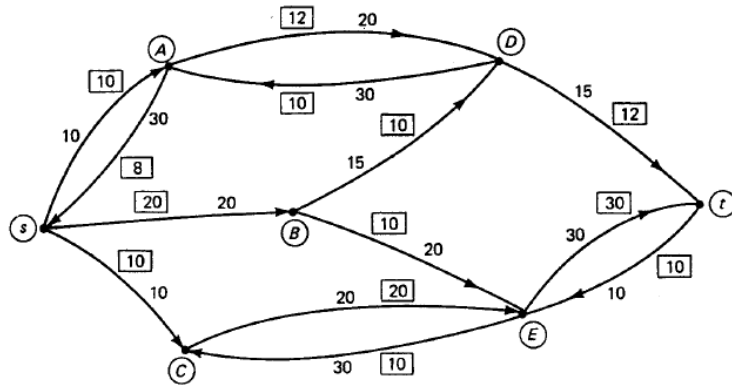


Figure (2)

2. Algorithms for the network flow problem

The first algorithm for the network flow problem was given by Ford and Fulkerson. They used that algorithm not only to solve instances of the problem, but also to prove theorems about network flow.

1. The algorithm of Ford and Fulkerson

The basic idea of the Ford-Fulkerson algorithm for the network flow problem is this: start with some flow function (initially this might consist of zero flow on every edge). Then look for a *flow augmenting path* in the network. A flow augmenting

path is a path from the source to the sink along which we can push some additional flow.

In Fig. (3) below we show a flow augmenting path for the network of Fig. (4). The capacities of the edges are shown on each edge, and the values of the flow function are shown in the boxes on the edges.

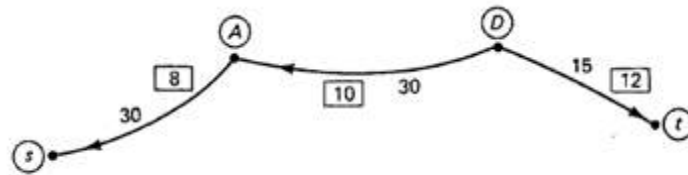
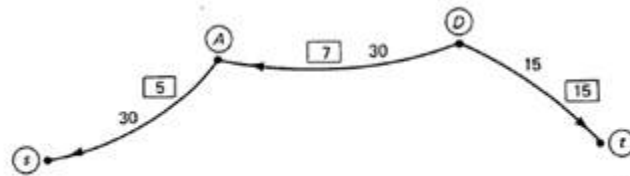


Figure (3)



Figure(4)

An edge can get elected to a flow augmenting path for two possible reasons. Either (a) the direction of the edge is *coherent* with the direction of the path from source to sink and the present value of the flow function on the edge is below the capacity of that edge, or (b) the direction of the edge is *opposed* to that of the path from source to sink and the present value of the flow function on the edge is strictly positive.

Indeed, on all edges of a flow augmenting path that are coherently oriented with the path we can increase the flow along the edge, and on all edges that are incoherently

oriented with the path we can decrease the flow on the edge, and in either case we will have *increased the value of the flow* (think about that one until it makes sense). It is, of course, necessary to maintain the conservation of flow, *i.e.*, to respect Kirchhoff's laws. To do this we will augment the flow on every edge of an augmenting path by the same amount. If the conservation conditions were satisfied before the augmentation then they will still be satisfied after such an augmentation. It may be helpful to remark that an edge is coherently or incoherently oriented only *with respect to a given path* from source to sink. That is, the coherence, or lack of it, is not only a property of the directed edge, but depends on how the edge sits inside a chosen path.

Thus, in Fig. (3) the first edge is directed towards the source, *i.e.*, incoherently with the path. Hence if we can *decrease* the flow in that edge we will have *increased* the value of the flow function, namely the net flow out of the source. That particular edge can indeed have its flow decreased, by at most 8 units. The next edge carries 10 units of flow towards the source. Therefore if we *decrease* the flow on that edge, by up to 10 units, we will also have *increased* the value of the flow function. Finally, the edge into the sink carries 12 units of flow and is oriented towards the sink. Hence if we *increase* the flow in this edge, by at most 3 units since its capacity is 15, we will have increased the value of the flow in the network.

Since every edge in the path that is shown in Fig. (3) can have its flow altered in one way or the

other so as to increase the flow in the network, the path is indeed a flow augmenting path. The most that we might accomplish with this path would be to push 3 more units of flow through it from source to sink.

We couldn't push more than 3 units through because one of the edges (the edge into the sink) will tolerate an augmentation of only 3 flow units before reaching its capacity.

To augment the flow by 3 units we would decrease the flow by 3 units on each of the first two edges and increase it by 3 units on the last edge. The resulting flow in this path is shown in Fig. (4). The flow in the full network, after this augmentation, is shown in Fig. (5). Note carefully that if these augmentations are made then flow conservation at each vertex of the network will still hold.

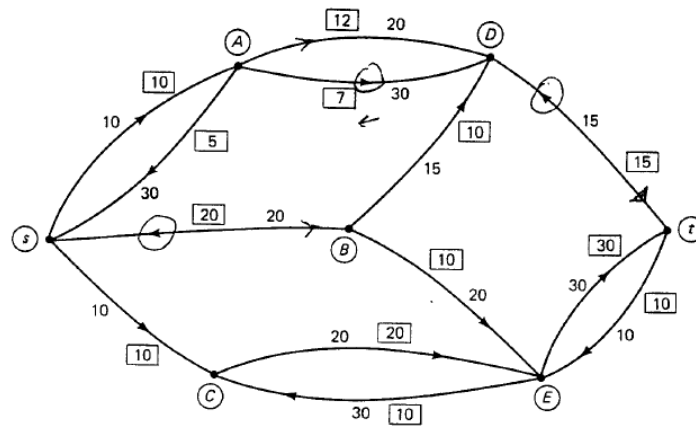


Figure (5)

After augmenting the flow by 3 units as we have just described, the resulting flow will be the one that is shown in Fig. (5). The value of the flow in Fig. (2) was 32 units. After the augmentation, the flow function in Fig. (5) has a value of 35 units. We have just described the main idea of the Ford-Fulkerson algorithm. It first finds a flow augmenting path. Then it augments the flow along that path as much as it can. Then it finds another flow augmenting path, etc. The algorithm terminates when no flow augmenting paths exist. We will prove that when that happens, the flow will be at the maximum possible value, *i.e.*, we will have found the solution of the network flow problem. We will now describe the steps of the algorithm in more detail.

Definition. Let f be a flow function in a network \mathbf{X} . We say that an edge e of \mathbf{X} is usable from v to w if either e is directed from v to w and the flow in e is less than the capacity of the edge, or e is directed from w to v and the flow in e is > 0 .