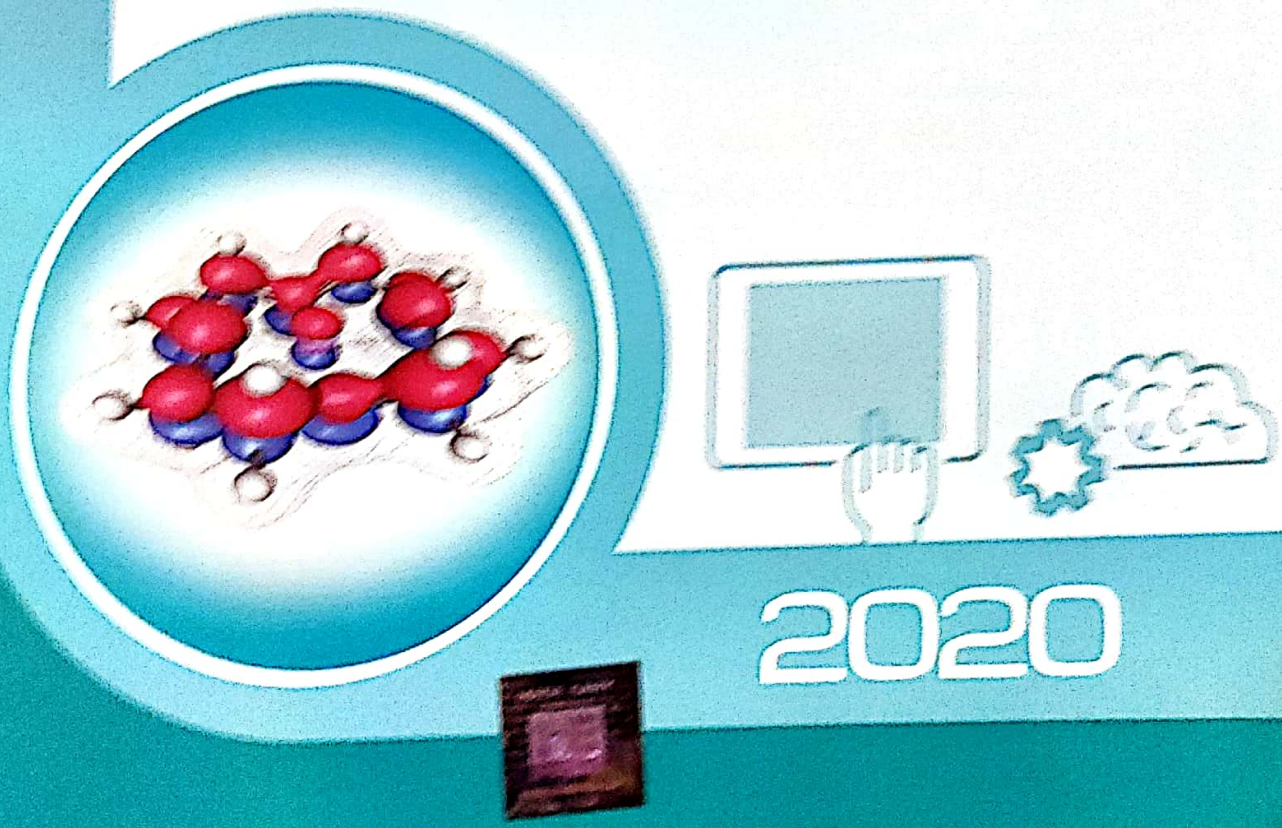
GATE PSUs

MADE EASY Publications

POSTAL STUDY PACKAGE

COMPUTER SCIENCE & IT



2020

THEORY BOOK

Algorithms

Well illustrated theory with solved examples

Computer Science & IT

Algorithms

Comprehensive Theory

with Solved Examples and Practice Questions

MADE EASY Publications

Contents

Algorithms

■■■■

CHAPTER 01

Asymptotic Analysis of Algorithms

1.1 Need for Performance Analysis

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple; we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

Choosing the Best Algorithm

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2. It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

ASYMPTOTIC ANALYSIS is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the **order of growth** of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000 $n \log n$ and 2 $n \log n$ time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is $n \log n$). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis. Also, in asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower always performs better for your particular situation. So, you may end up choosing an algorithm that is asymptotically slower but faster for your software.

1.2 Worst, Average and Best Cases

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>
// Linearly search x in arr []. If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
      int i;
      for (i=0; i<n; i++)
      {
            if (arr[i] == x)
            return i;
      }
      return -1;
}
/* Driver program to test above functions*/
int main()
{
      int arr[] = {1, 10, 30, 15};
      int x = 30;
      int n = sizeof(arr)/sizeof(arr[0]);
      printf("%d is present at index %d", x, search(arr, n, x));
      getchar();
      return 0;
}
```

1.2.1 Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array (or) last element, the search functions compares it with all the elements of arr[ ] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

1.2.2 Average Case Analysis

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by $(n + 1)$. Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1}\Theta(i)}{n+1} = \frac{\Theta\left((n+1)\times\frac{n+2}{2}\right)}{n+1} = \Theta(n)$$

1.2.3 Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$.

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of fan algorithm which is good information.

The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

1.3 Asymptotic Notations

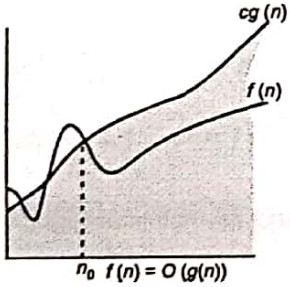
Let f be a nonnegative function. Then we define the three most common asymptotic bounds as follows.

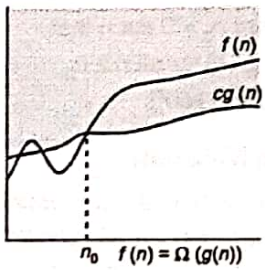
1.3.1 Big-Oh(O)

We say that $f(n)$ is Big-O of $g(n)$, written as $f(n) = O(g(n))$, iff there are positive constants c and n_0 such that

$0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

If $f(n) = O(g(n))$, we say that $g(n)$ is an upper bound on $f(n)$.





1.3.2 Big-Omega (Ω)

We say that $f(n)$ is Big-Omega of $g(n)$, written as $f(n) = \Omega(g(n))$, iff there are positive constants c and n_0 such that

$0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0$

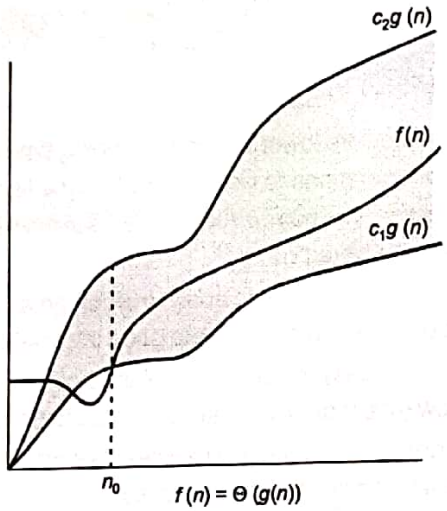
If $f(n) = \Omega(g(n))$, we say that $g(n)$ is a lower bound on $f(n)$.

1.3.3 Big-Theta (Θ)

We say that $f(n)$ is **Big-Theta** of $g(n)$, written as $f(n) = \Theta(g(n))$, iff there are positive constants c_1, c_2 and n_0 such that

$$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \text{ for all } n \ge n_o$$

Equivalently, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. If $f(n) = \Theta(g(n))$, we say that $g(n)$ is a **tight bound** on $f(n)$.



NOTE: Sometimes the notation $f(n) \le O(g(n))$ is used instead of $f(n) = O(g(n))$ (similar for Ω and Θ). These mean essentially the same thing, and the use of either is generally personal preference.

1.3.4 Small-Oh (o) Notation

The asymptotic upper bound provided by o-notation may not be asymptotically tight e.g. $2n^2 = O(n^2)$ is asymptotically tight but $2n = O(n^2)$ is not.

Hence we use o-notiation to devote an upper bound that is not asymptotically tight.

$o(g(n)) = \{f(n)\}$: for every positive constant $c > 0$

there exist a constant $n_0 > 0$ such that

$$0 \le f(n) < cg(n) \text{ for all } n \ge n_0$$

eg. $2n = \{f(n)\}$ but $2n^2 \ne o(n^2)$

1.3.5 Small-Omega (ω) Notation

ω-notation is used to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{f(n)\}$: for every positive constant $n_0 > 0$

there exist a constant $n_0 > 0$ such that

$$0 \le cg(n) < f(n) \text{ for all } n \ge n_0$$

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

eg. $\frac{n^2}{2} = \omega(n)$ but $\frac{n^2}{2} \ne \omega(n^2)$

1.4 Analysis of Loops

1. **O(1):** Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.
   // set of non-recursive and non-loop statements
   For example swap( ) function has O(1) time complexity. A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++)
{
    // some O(1) expressions
}
```

2. **O(*n*):** Time Complexity of a loop is considered as O(*n*) if the loop variables is incremented/decremented by a constant amount. For example following functions have O(*n*) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c)
{
    // some O(1) expressions
}
for (int i = n; i > 0; i -= c)
{
    // some O(1) expressions
}
```

3. **O(n^c):** Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n^2) time complexity.

```
for (int i = 1; i <= n; i += c)
{
    for (int j = 1; j <= n; j += c)
    {
        // some O(1) expressions
    }
}
    for (int i = n; i > 0; i += c)
{
    for (int j = i + 1; j <= n; j += c)
{
    // some O(1) expressions
}
}
```

   For example Selection sort and Insertion Sort have O(n^2) time complexity.

4. **O(Log*n*)** Time Complexity of a loop is considered as O(Log*n*) if the loop variables is divided / multiplied by a constant amount.

```
Main( )
{
    i = 1;
```

```
        while(i < n)
        {
            i = 2x i;
        }
    }
```

In above code the statement inside the code runs log$n$ times. Initially i value is 1 ($i = 1$). As the loop executes the value increases $2i$, $2^2 i$, $2^3 i$, and so all. And takes log n increment of i value to become equal to n.

5. **O(loglog$n$):** Time Complexity of a loop is considered as O(loglog$n$) if the loop variables is reduced/ increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <= n; i = pow (i, c))
{
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i))
{
    // some O(1) expressions
}
```

Consecutive Loops

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <= m; i += c)
{
    // some O(1) expressions
}
for (int i = 1; i <= n; i += c)
{
    // some O(1) expressions
}
```

Time complexity of above code is O(m) + O(n) which is O($m + n$)

Example-1.1 What is time complexity of fun ()?

```
Int fun (int n)
{
    Int count = 0;
    For (int i = n; i > 0; i /= 2)
    For (int j = 0; j < i; j++)
    Count += 1;
    Return count;
}
```

Solution:
For a input integer n, the innermost statement of fun () is executed following times.
So time complexity $T(n)$ can be written as

$$T(n) = n + n/2 + n/4 + \ldots 1 = O(n)$$

The value of count is also $n + n/2 + n/4 + \ldots + 1$

Example - 1.2 What is time complexity of fun ()?

```
Int fun (int n)
{
        Int count = 0;
        For (int i = 0; i < n; i++)
        For (int j = i; j > 0; j—)
        Count = count + 1;
        Return count;
}
```

Solution:
The time complexity can be calculated by counting number of times the expression "count = count + 1;" is executed.
The expression is executed $0 + 1 + 2 + 3 + 4 + \ldots + (n-1)$ times.
Time complexity = $0 + 1 + 2 + 3 + \ldots + n - 1 \equiv \Theta(n(n-1)/2) = \Theta(n^2)$.

Example - 1.3 In a competition, four different functions are observed. All the functions use a single for loop and within the for loop, same set of statements are executed. Consider the following for loops:

A. for($i = 0$; $i < n$; i++)
B. for($i = 0$; $i < n$; i += 2)
C. for($i = 1$; $i < n$; i *= 2)
D. for($i = n$; $i > -1$; i /= 2)

If n is the size of input (positive), which function is most efficient (if the task to be performed is not an issue)?

Solution:
The time complexity of first for loop is $O(n)$. The time complexity of second for loop ($n/2$), equivalent to $O(n)$ in asymptotic analysis. The time complexity of third for loop is $O(\log n)$. The fourth for loop doesn't terminate.

Example - 1.4 Consider the following functions:

$$f(n) = 2^n$$
$$g(n) = n!$$
$$h(n) = n^{\log n}$$

Which of the following statements about the asymptotic behavior of $f(n)$, $g(n)$, and $h(n)$ is true?

(a) $f(n) = O(g(n))$; $g(n) = O(h(n))$
(b) $f(n) = \Theta(g(n))$; $g(n) = O(h(n))$
(c) $g(n) = O(f(n))$; $h(n) = O(f(n))$
(d) $h(n) = O(f(n))$; $g(n) = \Omega(f(n))$

Solution: (d)

Example-1.5 In the following C function, let $n \geq m$.

int gcd (n, m) {if (n % m == 0) return m; n = n % m; return gcd (m, n);}

How many recursive calls are made by this function?

(a) $\Theta(\log n)$ (b) $\Theta(n)$

(c) $\Theta(\log \log n)$ (d) $\Theta(\text{sqrt}(n))$

Solution:

$\Theta(\log n)$

Example-1.6 Consider the following two functions. What are time complexities of the functions?

```
int fun1(int n)
{
    if (n ≤ 1) return n;
    return 2*fun1(n – 1);
}
    int fun2 (int n)
{
    if (n <= 1) return n;
    return fun2(n – 1) + fun2(n – 1);
}
```

Solution:

Time complexity of fun1( ) can be written as $T(n) = T(n-1) + C$ which is $O(n)$ Time complexity of fun2( ) can be written as $T(n) = 2T(n-1) + C$ which is $O(2^n)$

Example-1.7 Consider the following C-program fragment in which i, j and n are integer variables.

for ($i = n$, $j = 0$; $i > 0$; $i = 2$, $j+ = i$);

Let val(j) denote the value stored in the variable j after termination of the for loop. Which one of the following is true?

(a) val(j) = $\Theta(\log n)$ (b) val(j) = Θ (sqrt(n))

(c) val(j) = $\Theta(n)$ (d) val(j) = Θ (n log n)

Solution:

semicolon after the for loop, so there is nothing in the body. The variable j is initially 0 and value of j is sum of values of i. i is initialized as n and is reduced to half in each iteration.

$$j = n/2 + n/4 + n/8 + \ldots + 1 = \Theta(n)$$

Example-1.8 Consider the following pseudo code. What is the total number of multiplications to be performed?

```
D = 2
    for i = 1 to n do
        for j = i to n do
            for k = j + 1 to n do
                D = D x 3
```

Solution:

The statement "D = D x 33 is executed $n^*(n+1)^*(n-1)/6$ times. Let us see how.

For $i = 1$, the multiplication statement is executed $(n-1) + (n-2) + .. 2 + 1$ times.

For $i = 2$, the statement is executed $(n-2) + (n-3) + ... 2 + 1$ times

For $i = n-1$, the statement is executed once.

For $i = n$, the statement is not executed at all

So overall the statement is executed following times

$$[(n-1) + (n-2)+...2 + 1] + [(n-2) + (n-3)+...2 + 1] + ...+ 1 + 0$$

The above series can be written as

$$S = [n^*(n-1)/2 + (n-1)^*(n-2)/2+...+ 1]$$

The sum of above series can be obtained by trick of subtraction the series from standard Series S1 = $n^2 + (n-1)^2 +...1^2$. The sum of this standard series is $n^*(n+1)^*(2n+1)/6$

$$S1 - 2S = n + (n-1) + ... 1 = n^*(n+1)/2$$
$$2S = n^*(n+1)^*(2n+1)/6 - n^*(n+1)/2$$
$$S = n^*(n+1)^*(n-1)/6$$

Example-1.9 Let A[1, . . . , n] be an array storing a bit (1 or 0) at each location, and $f(m)$ is a function whose time complexity is $\Theta(m)$. Consider the following program fragment written in a C like language:

```
counter = 0;
for (i = 1; i < = n; i++)
{   if (A[i] == 1)
        counter++;
    else
    {
        f(counter);
        counter = 0;
    }
}
```

Find true statements

Solution:

a. All 1s in A[ ]: Time taken is $\Theta(n)$ as only counter++ is executed n times.

b. All 0s in A[ ]: Time taken is $\Theta(n^2)$ as $f(0)$ is called n times each call $\Theta(n)$.

c. Half 1s, then half 0s: Time taken is $\Theta(n^2)$. Time = $n/2 + n/2 \times n = \Theta(n^2)$.

1.5 Comparisons of Functions

Many of the relational properties apply to asymptotic comparisons as well i.e. functions can be compared asymptotically.

1. Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

2. **Reflexivity:**

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

   $f(n) = O(f(n))$,

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

3. **Symmetry:**

   $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

4. **Transpose Symmetry:**

   $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$,

   $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

5. We can say that $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is asymptotically larger than g(n) if $f(n) = \omega(g(n))$
6. $o(g(n)) \wedge \omega(g(n))$ is the empty set.
7. Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Then maximum $(f(n), g(n)) = \Theta(f(n) + g(n))$.
8. For any real constant a and b, where $b > 0$.
   $(n + a)^b = \Theta(n^b)$
9. **Monotonicity:**
   A function is monotonically increasing if $m \le n$ implies $f(m) \le f(n)$
   A function is monotonically decreasing if $m \le n$ implies $f(m) \ge f(n)$
   A function is strictly increasing if $m < n$ implies $f(m) < f(n)$
   A function is strictly decreasing if $m < n$ implies $f(m) > f(n)$

1.6 Asymptotic Behaviour of Polynomials

$p(n) = \sum_{i=0}^{d} a_i n^i$, where $a_d > 0$ be a degree d polynomial in n, and let K be constant then:

(a) If $K \ge d$ then $p(n) = O(n^k)$

(b) If $K \le d$ then $p(n) = \Omega(n^k)$

(c) If $K = d$ then $p(n) = \Theta(n^k)$

(d) If $K > d$ then $p(n) = o(n^k)$

(e) If $K < d$ then $p(n) = \omega(n^k)$

Polynomially Bounded Functions

Given a non negative integer d, a polynomial in n of degree d is a function $p(n)$ of the form $p(n) = \sum_{i=0}^{d} a_i n^i$, where the constant $a_0, a_1 \ldots a_d$ are the coefficients of the polynomial and $a_d \neq 0$. A polynomial is asymptotically +ve iff $a_d > 0$.

For an asymptotically +ve polynomial $p(n)$ of degree d we have $p(n) = \Theta(n^d)$ we say that a function $f(n)$ is polynomially bounded if $f(x) = O(n^k)$ for some constant $k > 0 \equiv f(n) = n^{\text{constant}}$.

NOTE: $n^b = O(a^n)$, since $\lim_{x\to\infty} \frac{n^b}{a^n} = 0$. Thus any exponential function with base strictly greater than n^b, grows faster than any polynomial function.

Poly Logarithmically Bounded Functions

We say that the function is polylogarithmically bounded if $f(n) = O(\lg^k n)$ for some constant k.

NOTE: $\log^b n = O(n^a) \Rightarrow \frac{\log^b n}{n^a} = 0$ for any constant $a > 0$.

Commonly used Rate of Growths

Time complexity	Name
$O(1)$	Constant
$\log n$	Logarithmic
n	Linear
$n\log n$	Linear logarithmic
n^2	Quadratic
n^3	Cubic
c^n	Exponential

The decreasing order of growth of time complexity is

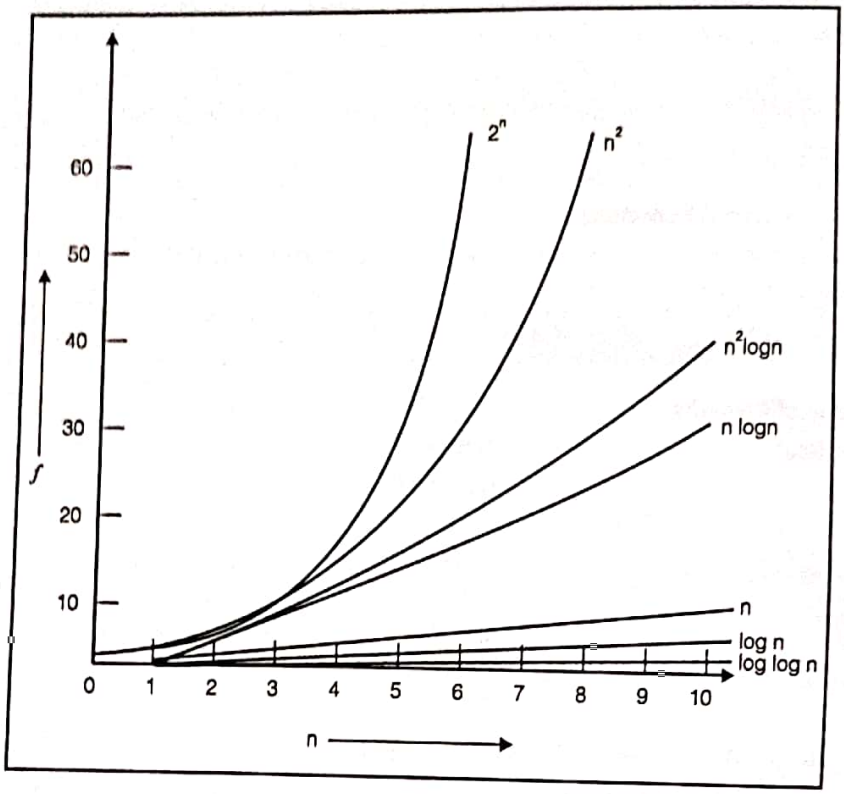
$$2^{2^n} > n! > 4^n > 2^n > n^2 > n\log n > \log(n!) > n > 2^{\log n} > \log^2 n > \sqrt{\log n} > \log\log n > 1$$

Remember

- **$\log(n!) < (\log(n))!$:** This is true because applying the logarithmic operation after doing the factorial will drastically decrease the value compared to when we apply log and then apply factorial.
- **$\log(\log^* n) < \log^*(\log n)$:** $\log^* n$ specifies that how many times log should be applied on n repeatedly to make $n \le 1$. Example $\log^* 16 = 3$, $\log^* 64 = 4$, $\log^* 256 = 4$. So on.
- **$n < (\log n)^{\log n}$:** As long as power of $(\log n)$ is constant it is less than n for some constant C.
  $n > \log n$, $n > (\log n)^2 \ldots n > (\log n)^{100000}$
  But when the power of $(\log n)$ is the function of n, then n becomes less than $(\log n)^{f(n)}$.
- **$2^n < n! < n^n$**

Growth of Functions

$\log n$	n	$n\log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296



$O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^3) \ldots < O(n^k) < O(2^n)$

Summary

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We say that $f(n)$ is Big-O of $g(n)$, written as $f(n) = O(g(n))$
- We say that $f(n)$ is Big-Omega of $g(n)$, written as $f(n) = \Omega(g(n))$
- We say that $f(n)$ is **Big-Theta** of $g(n)$, written as $f(n) = \Theta(g(n))$
- $O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^3) \ldots < O(n^k) < O(2^n)$
- $\log(n!) < (\log(n))!$
- $\log(\log^* n) < \log^*(\log n)$
- $n < (\log n)^{\log n}$
- $2^n < n! < n^n$

Student's Assignments

Q.1 Which one of the following is true?

1. $an = o(n^2)\ a \geq 0$
2. $an^2 = O(n^2)\ a > 0$
3. $an^2 \neq o(n^2)\ a > 0$

(a) Only 1 and 2 are correct
(b) Only 1 is correct
(c) 1 and 3 are correct only
(d) All are correct

Q.2 Which of the following statements is true?

S1: $\frac{1}{2}n^2 = \omega(n)$ S2: $\frac{1}{2}n^2 = \omega(n^2)$

(a) S1 is correct
(b) S2 is correct
(c) S1 and S2 both are correct
(d) None of the above

Q.3 $f(n) = 3n^2 + 4n + 2$. Which will be the exact value for $f(n)$

(a) $\Theta(n^2)$ (b) $o(n^2)$
(c) $O(n^2)$ (d) $\Omega(n^2)$

Q.4 $f(n) = O(g(n))$ if and only if

(a) $g(n) = O(f(n))$ (b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ (d) None of these

Q.5 $f(n) = o(g(n))$ if and only if

(a) $g(n) = \Omega(f(n))$
(b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ and $g(n) = \omega(f(n))$
(d) None of these

Q.6 Which of the fallowing is not correct?

(a) $f(n) = O(f(n))$
(b) $c^* f(n) = O(f(n))$ for a constant C
(c) $(f(n) + g(n)) = o(g(n) + f(n))$
(d) None of the above

Q.7 $f(n) = \Theta(g(n))$ is

(a) $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(b) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $0 \leq f(n) \leq C\, g(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(d) None of the above

Q.8 $f(n) = O(g(n))$ implies

(a) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(b) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $0 \leq f(n) \leq C\, g(n)\ \forall n \geq n_0$ where C, n_0 are + integer
(d) $0 \leq C_1 g(n) \leq C_2 f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer

Q.9 $f(n) = \Theta(g(n))$ implies

(a) $f(n) = O(g(n))$ only
(b) $f(n) = \Omega(g(n))$ only
(c) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
(d) None of the above

Q.10 $f(n) = o(g(n))$ implies

(a) $0 \leq f(n) \leq Cg(n)$ such that there exists some positive constant C and $n_0 > \forall n \geq n_0$
(b) $0 \leq f(n) < Cg(n)$ for every +ve constant $C >$ 0 there exisxts $n_0 > 0$, $\forall n \geq n_0$
(c) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ such that C_1, C_2 and n_0 are +ve constants
(d) None of the above

Common Data for Questions (11 and 12):

```
void x (int A[ ]), int n)
{   int i, j;
    for (i = 0; i < n; i++)
    {   j = n – 1;
        while (j > i)
        {
            swap (A[j], A[j – 1]);
            j --;
        }
    }
}
```

Q.11 What will be time complexity of the above algorithm?

(a) $O(n)$ (b) $O(n^2)$
(c) $O(n \log_2 n)$ (d) $O(n^3)$

Q.12 Which of the following statements are True?

(a) $100\, n\log n = O(n\log n/100)$

(b) $\sqrt{\log n} = O(\log\log n)$

(c) If $0 < x < y$ then $n^x = O(x^y)$

(d) $2^n \neq O(n^c)$ where c is a constant and $c > 0$

Q.13 Which of the following statements (k, m are constants) are True?

(a) $(n + k)^m = O(n^m)$

(b) $2^{n+1} = O(2^n)$

(c) $2^{2^n} = O(2^n)$

(d) $f(n) = O(f(n)^2)$

Q.14 Which of the following statements are True?

(a) $n^2 \cdot 2^{3\log_2 n} = \Theta(n^5)$

(b) $\frac{4^n}{2^n} = \Theta(2^n)$

(c) $2^{\log_2 n} = \Theta(n^2)$

(d) if $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

Answer Key:

1. (d)	**2.** (a)	**3.** (a)	**4.** (c)	**5.** (c)
6. (c)	**7.** (a)	**8.** (c)	**9.** (c)	**10.** (b)
11. (b)	**12.** (a, c, d)			
13. (a, b)	**14.** (a, b)			

CHAPTER 02

Recurrence Relations

2.1 Introduction

A recurrence is an equation or inequality that describes a function in term of its value on smaller inputs. For instance we describe the worst-case running time $T(n)$ of the **Merge-Sort** procedure by the recurrence whose solution is $\Theta(n \log n)$. The recurrence relation for Merge-Sort is shown below.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \quad \ldots(2.1)$$

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3 to 1/3 split. If the divide and combine steps linear time, such an algorithm would give rise to the recurrence

$$T(n) = T(2n/3) + T(n/3) + \Theta(n)$$

This chapter offers three methods for solving recurrences–that is, for obtaining asymptotic "Θ" or "O" bounds on the solution.

- **Substitution Method:** This method consists of guessing an asymptotic (upper or lower) bound on the solution, and trying to prove it by induction.
- **Recursion Tree Method-Iteration Method:** These are two closely related methods for expressing $T(n)$ as a summation which can then be analyzed. A recursion tree is a graphical depiction of the entire set of recursive invocations of the function T. The goal of drawing a recursion tree is to obtain a guess which can then be verified by the more rigorous substitution method. Iteration consists of repeatedly substituting the recurrence into itself to obtain an iterated (i.e. summation) expression.
- **Master Method:** This is a cookbook method for determining asymptotic solutions to recurrences of a specific form.

Technicalities in Recurrences

We neglect certain technical details when we state and solve recurrences. For example, if we call Merge-Sort on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$ because $n/2$ is not an integer when n is odd. The actual recurrence describing the worst-case of Merge-Sort is given by

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

The running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. We shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant of small n. For example, we normally state recurrence (2.1) as $T(n) = 2T(n/2) + \Theta(n)$, without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

2.2 Substitution Method

Example-2.1 What is the time complexity of the following recurrence relation

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

Solution:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) + \ldots + n-2 + n-1 + n \\ &= T(1) + 2 + 3 + 4 + \ldots + n \\ &= 1 + 2 + 3 + \ldots + n \\ &= \frac{n(n+1)}{2} \Rightarrow O(n^2) \end{aligned}$$

Example-2.2 Find the complexity of the recurrence:

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

Solution:

$$\begin{aligned} T(n) &= 2T(n-1) - 1 \\ &= 2(2T(n-2) - 1) - 1 = 2^2 T(n-2) - 2 - 1 \\ &= 2^2 (2T(n-3) - 2 - 1) - 1 = 2^3 T(n-4) - 2^2 - 2^1 - 2^0 \\ &\vdots \\ &= 2^n T(n-n-1) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0 \\ &= 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots 2^2 - 2^1 - 2^0 \\ &= 2^n - (2^{n-1} - 1) \qquad \because 2^{n-1} - 2^{n-2} + \ldots + 2^0 = 2^n \\ T(n) &= 1 \end{aligned}$$

Time complexity is O(1).

Example - 2.3 Find the complexity of the recurrence:

$$T(n) = \begin{cases} 3T(n-1) & \text{If } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

Solution:

Let us try solving this function with substitution.

$$T(n) = 3T(n-1)$$
$$T(n) = 3\,(3T(n-2)) = 3^2\,T(n-2)$$
$$T(n) = 3^2(3T(n-3)$$
$$\vdots$$
$$T(n) = 3^n\,T(n-n) = 3^n\,T(0) = 3^n, \text{ therefore } O(3^n).$$

Example - 2.4 Find the complexity of the recurrence:

$$T(n) = \begin{cases} T(n-1) \times n & \text{If } n > 1 \\ 1 & \text{If } n = 1 \end{cases}$$

Solution:

$$T(n) = T(n-1)n$$
$$= [T(n-2) \times (n-1)]n \qquad \text{by putting } n = n-1$$
$$= T(n-3) \times (n-2) \times (n-1)n$$
$$\vdots$$
$$= T(n-(n-1)) \times (n-(n-2)) \ldots (n-2) \times (n-1) \times n$$
$$= 1 \times 2 \times 3 \times \ldots \times n = n!$$
$$= O(n^n)$$

Example - 2.5 What is the time complexity of the following recurrence relation (use substitution method)?

$$T(N) = 3T(n/4) + n \text{ for } n > 1$$

Solution:

Assume n to be a power of 4, i.e., $n = 4^k$ and $k = \log 4n$

$$T(n) = 3T(n/4) + n$$
$$= 3(3T(n/16) + 3(n/4) + n$$
$$= 9T\,(n/16) + 3(n/4) + n$$
$$= 27\,T(n/64) + 9(n/16) + n + 3(n/4) + n$$
$$= \ldots$$
$$= 27\,T(n/64) + 9(n/16) + n + 3(n/4) + n$$

With $n = 4^k$ and $T(1) = 1$

$$T(n) = 3^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n = 3^{\log_4 n}\,T(1) + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n$$

$$= n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)} \frac{3^i}{4^i} n$$

We use the formula $a^{\log_b n} = n^{\log_b a}$, n remains constant throughout the sum.

$$\sum_{i=0}^{m} X^i = \frac{X^{m+1}-1}{X-1}$$

In this case X = 3/4 and $m \log_4 n - 1$. We get

$$T(n) = n^{\log_4 3} + n\frac{(3/4)^{\log_4 n+1}-1}{(3/4)-1}$$

$$\left(\frac{3}{4}\right)^{\log_4 n} = n^{\log_4(3/4)} = n^{\log_4 3-\log_4 4} = n^{\log_4 3-1} = \frac{n^{\log_4 3}}{n}$$

If we plug this back, we get

$$T(n) = n^{\log_4 3} + n\frac{\frac{n^{\log_4 3}}{n}-1}{(3/4)-1} = n^{\log_4 3} + \frac{n^{\log_4 3}-n}{-1/4}$$

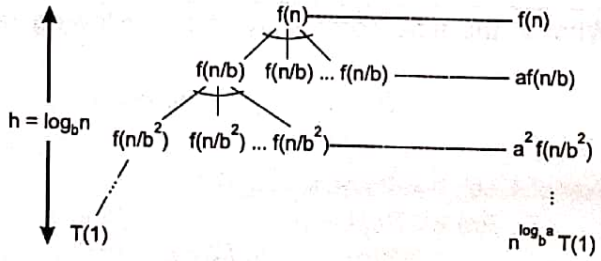
$$= n^{\log_4 3} + 4(n - n^{\log_4 3}) = 4n - 3n^{\log_4 3}$$
$$= \Theta(n)$$

Recursion-tree Method

A recursion tree models the costs (time) of a recursive execution of an algorithm. The recursion tree method is good for generating guesses for the substitution method.

The recursion-tree method can be unreliable, just like any method that uses ellipses (...). The recursion-tree method promotes intuition, however.

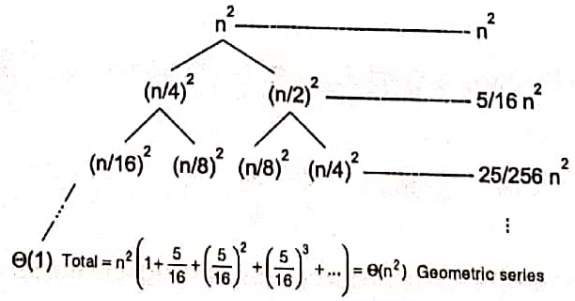
Idea of Master Theorem for using Recursion Tree



***Example* - 2.6** What is the time complexity of the following recurrence relation (use recursive tree method)?

$$T(n) = T(n/4) + T(n/2) + n^2$$

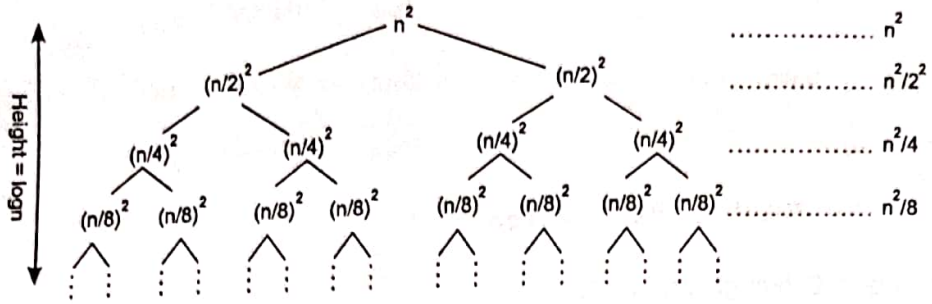
Solution:



Example - 2.7 What is the time complexity of the following recurrence relation (use recursive tree method)?

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

Solution:

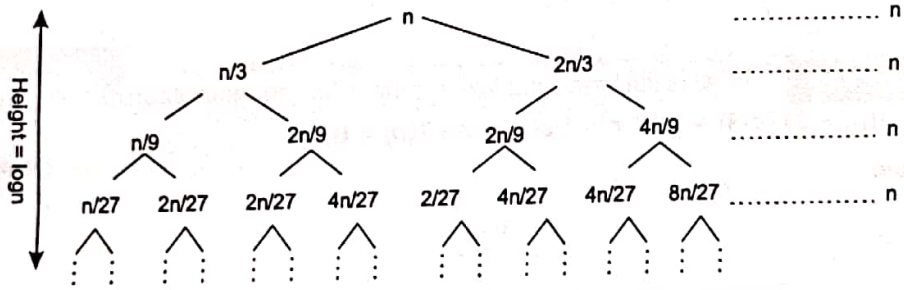


This is geometric series, thus in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.

Example - 2.8 What is the time complexity of the following recurrence relation (use recursive tree method)?

$$T(n) = T(n/3) + T(2n/3) + n$$

Solution:



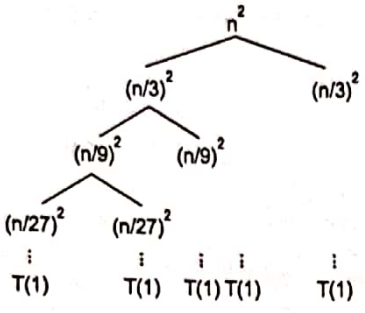
Leftmost branch peters out (when sub problems size is 1) after $\log_3 n$ levels	Rightmost branch peters out (when sub problems size is 1) after $\log_{3/2} n$ levels

Expanding out the first few levels, the recurrence tree can be drawn like this. Note that the tree here is not balanced: the longest path is the rightmost one, and its length is $\log_{3/2} n$. Hence our guess for the closed form of this recurrence is $O(n \log n)$.

Example - 2.9 What is the time complexity of the following recurrence relation (use recursive tree method)?

$$T(n) = 2T(n/3) + n^2 \text{ for all } n > 1$$

Solution:



Level	Nodes	Cost
0	$2^0 = 1$	$2^0 \times (n/3^0)^2 = (2/3^2)^0 \times n^2$
1	$2^1 = 2$	$2^1 \times (n/3^1)^2 = (2/3^2)^1 \times n^2$
2	$2^2 = 4$	$2^2 \times (n/3^2)^2 = (2/3^2)^2 \times n^2$
3	$2^3 = 8$	$2^3 \times (n/3^3)^2 = (2/3^2)^3 \times n^2$
⋮	⋮	⋮
$\log_3 n$	$2^{\log_3 n}$	$T(1) \times 2^{\log_3 n} = T(1) \times n^{\log_3 2}$

Cost for level 0 through $\log_3 n - 1$:

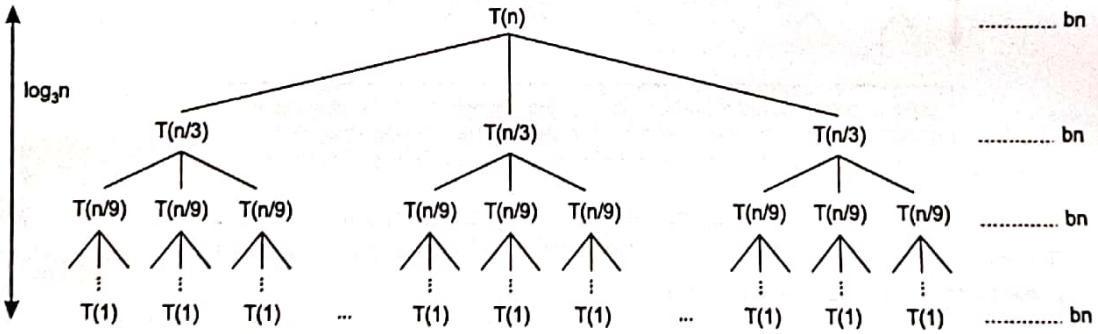
$$n^2 \sum_{i=0}^{\log_3 n - 1} \left(\frac{2}{9}\right)^i = n^2 \left[\frac{\left(\frac{2}{9}\right)^{\log_3 n}}{\frac{2}{9} - 1}\right] < n^2 \sum_{i=0}^{\infty} \left(\frac{2}{9}\right)^i = n^2 \left(\frac{1}{1 - \frac{2}{9}}\right) = \frac{9}{7} n^2 \qquad \because \sum_{i=0}^{n} x^i = \frac{x^{n+1}}{x-1}$$

Cost for bottom, level $\log_3 n$: $T(1) \times 2^{\log_3 n} = T(1) \times n^{\log_3 2} = \Theta(n^{\log_3 2})$

Total cost: $\frac{9}{7} n^2 + \Theta(n^{\log_3 2}) = O(n^2)$

Example - 2.10 What is the time complexity of the following recurrence relation (use recursive tree method)? $T(n) = 3T(n/3) + bn$ If $n > 1$ otherwise $T(n) = b$

Solution:

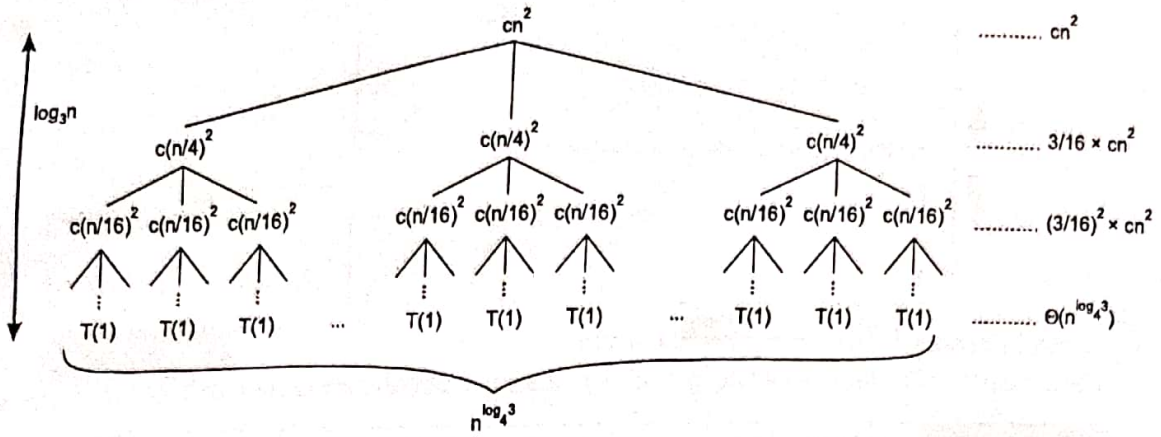


Total cost = $bn \times \log_3 n + bn = O(\log_3 n)$

Example - 2.11 What is the time complexity of the following recurrence relation (use recursive tree method)?

$$T(n) = 3T(n/4) + \Theta(n^2)$$

Solution:



$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n\log_4 3)$$

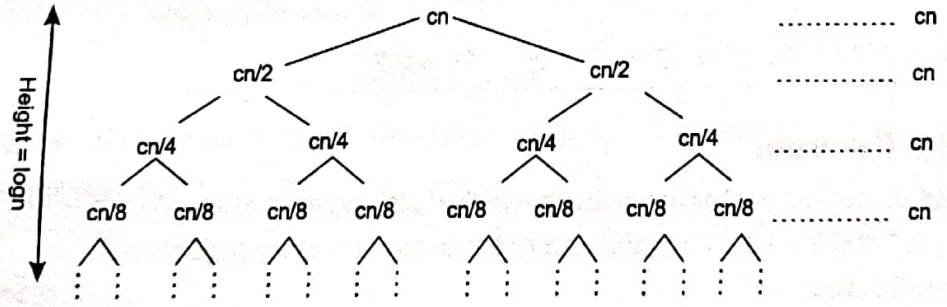
$$= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(3/16)}cn^2 + \Theta(n\log_4 3) = \frac{16}{13}cn^2 + \Theta(n\log_4 3)$$

$$= O(n^2)$$

Example-2.12 What is the time complexity of the following recurrence relation (use recursive tree method)? $T(n) = 2T(n/2) + \Theta(n)$ If $n > 1$ otherwise $T(n) = \Theta(1)$.

Solution:



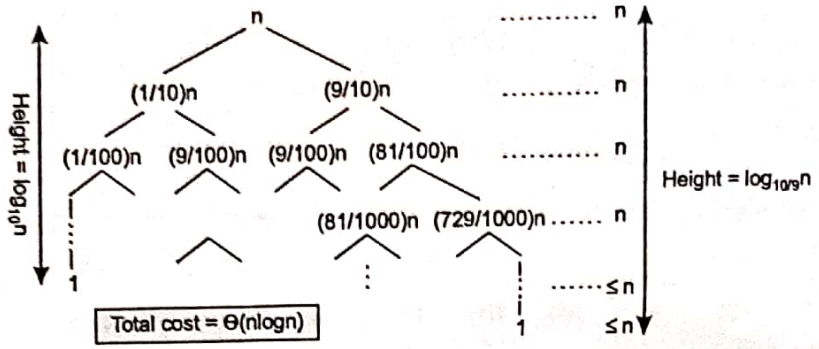
Total cost = $cn \log n + cn = \Theta(n \log n)$

The sub problems are of size $n/2^0$, $n/2^1$, $n/2^2$, the tree ends when $n/2^p = n/n = 1$, the trivial sub problem of size 1.

Thus the height of the tree is the power p to which we have to raise 2 before it becomes n, i.e., $p = \log n$. Since we start at 2^0 there are $\log n + 1$ levels. Multiplying by the work cn at each level, we get $cn \log n + cn$ for the total time.

Example-2.13 What is the time complexity of the following recurrence relation (use recursive tree method)? $T(n) = T(9n/10) + T(n/10) + n$

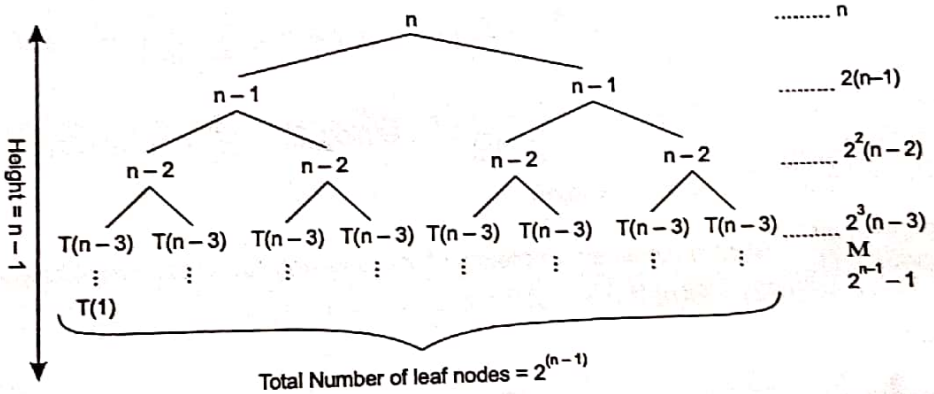
Solution:



The running time is therefore $\Theta(n \log n)$ whenever the split has constant proportionality.

Example-2.14 What is the time complexity of the following recurrence relation (use recursive tree method)? $T(n) = 2T(n-1) + n,\ n \geq 2\ T(1) = 1$.

Solution:



$2^0(n-0) + 2^1(n-1) + 2^2(n-2) + \ldots + 2^{n-1}(1) \rightarrow O(2^n)$.

2.3 Master Theorem

The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = O(n^{\log_b a} \log^k n)$ with $k \geq 0$ (most of the time $k = 0$), then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Example-2.15 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = 5T(n/2) + \Theta(n^2)$.

Solution:

$$T(n) = 5T(n/2) + \Theta(n^2)$$

$$a = 5,\ b = 2,\ f(n) = n^2$$

Compare n^2 to $n^{\log_2 5}$

So, time complexity= $O(n^{\log_2 5})$.

Example-2.16 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = 9T(n/3) + n$.

Solution:

$$T(n) = 9T(n/3) + n$$

$$a = 9,\ b = 3,\ f(n) = n$$

Here $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

$$f(n) = O(n^{\log_3 9 - \varepsilon});\quad \text{where } \varepsilon = 1$$

by using the case 1 of Master Theorem we can conclude that $T(n) = \Theta(n^2)$.

Example-2.17 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = T(2n/3) + 1$.

Solution:

$$T(n) = T(2n/3) + 1$$

$$a = 1,\ b = 3/2,\ f(n) = 1$$

Here $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

$$f(n) = \Theta(n^{\log_{3/2} 1}) = \Theta(1)$$

by using the case 2 of Master Theorem we can conclude that $T(n) = \Theta(\log n)$.

Example-2.18 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = 3T(n/4) + n\log n$.

Solution:

$$T(n) = 3T(n/4) + n\log n$$

$$a = 3,\ b = 4,\ f(n) = n\log n$$

Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$

$$f(n) = \Omega(n^{\log_4 3 + \varepsilon});\quad \text{where } \varepsilon \text{ approximately} = 0.2$$

For sufficiently large n, we have that $af(n/b) = 3(n/4)\log(n/4) \le (3/4)\, n\log n = cf(n)$ for $c = 3/4$. Consequently, by using the case 3 of Master Theorem we can conclude that $T(n) = \Theta(n\log n)$.

Example-2.19 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = 2T(n/2) + n\log n$.

Solution:

$$T(n) = 2T(n/2) + n\log n$$

Even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n\log n$, and $n^{\log_b a} = n$.

We might think that case 3 should apply $f(n) = n\log n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger.

The ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n\log n}{n} = \log n$ is asymptotically less than n^{ε} for any positive constant ε.

Hence master theorem cannot applied.

By using substitution method we get

$$T(n) = \Theta(n(\log n)^2)$$

Example-2.20 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = 7T(n/2) + \Theta(n^2)$.

Solution:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Which describes the running time of Strassen's algorithm. Here, we have $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, and thus $n^{\log_b a} = n^{\log_2 7}$. Rewritting $\log_2 7$ as $\log 7$ and recalling that $2.80 < \log 7 < 2.81$.

We see that $f(n) = O(n^{\log 7 - \varepsilon})$ for $\varepsilon = 0.8$. Again, case 1 applies, and we have the solution $T(n) = \Theta(n^{\log 7})$.

Example-2.21 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = T(\sqrt{n}) + c$.

Solution:

$$T(n) = T(\sqrt{n}) + c$$

Where Master Theorem is not possible.

Assume $n = 2^k$.

$\therefore$ $$T(2^k) = T((2^k)^{1/2}) + c$$
$$T(2^k) = T(2^{1/2}) + c$$

Assume $T(2^k) = S(k)$

$$S(k) = S(k/2) + c$$

Now we can apply Master Theorem

$a = 1$, $b = 2$ $$k^{\log_b^a} = k^{\log_2^1} = k^{\log_2^2} = 1$$
$$fck = 1$$

$\therefore$ 3rd case $\Rightarrow \Theta(1.\log k)$

$$S(k) = \Theta(\log k)$$
$$T(2k) = \Theta(\log k)$$

$$T(2^{\log_2^n}) = \Theta(\log(\log n))$$

$\Rightarrow$ $$T(n) = \Theta(\log\log(n))$$

Example-2.22 Find the time complexity of the following recurrence relation (using Master Theorem) $T(n) = T(\sqrt{n}) + \log n$.

Solution:

$$T(n) = T(\sqrt{n}) + \log n$$

Put n = 2k

$$T(2^k) = 2 \cdot T(2^{k/2}) + \log_2 2^k$$

put

$$T(2^k) = S(k)$$

$$S(k) = 2 \cdot S(k/2) + k$$

$$S(k) = k$$

$$T(2^k) = k$$

$$\Rightarrow \quad T(n) = O(\log_2 n)$$

Summary

- Three methods for solving recurrences–that is, for obtaining asymptotic "Θ" or "O" bounds on the solution.
  1. Substitution Method:
  2. Recursion Tree Method-Iteration Method:
  3. Master Method:
- Recurrences characterize the running times of algorithms. The substitution method for solving recurrences comprises two steps:
  1. Guess the form of the solution.
  2. Use mathematical induction to find the constants and show that the solution works.
- A recursion tree models the costs (time) of a recursive execution of an algorithm. The recursion tree method is good for generating guesses for the substitution method.
- The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$

Student's Assignments

Q.1 Find the time complexity of the following recurrence relation using Master Theorem

(a) $T(n) = 3T(n/2) + n^2$
(b) $T(n) = 4T(n/2) + n^2$
(c) $T(n) = T(n/2) + 2^n$
(d) $T(n) = 2^n T(n/2) + n^n$
(e) $T(n) = 16T(n/4) + n$
(f) $T(n) = 2T(n/2) + n/\log n$
(g) $T(n) = 2T(n/4) + n^{0.51}$
(h) $T(n) = 0.5T(n/2) + 1/n$
(i) $T(n) = 16T(n/4) + n!$
(j) $T(n) = \sqrt{2}\, T(n/2) + \log n$
(k) $T(n) = 3T(n/2) + n$
(l) $T(n) = 3T(n/3) + \sqrt{n}$
(m) $T(n) = 4T(n/2) + cn$
(n) $T(n) = 3T(n/4) + n\log n$
(o) $T(n) = 3T(n/3) + n/2$
(p) $T(n) = 6T(n/3) + n^2 \log n$
(q) $T(n) = 4T(n/2) + n/\log n$
(r) $T(n) = 64T(n/8) - n^2 \log n$
(s) $T(n) = 7T(n/3) + n^2$
(t) $T(n) = 4T(n/2) + \log n$

Q.2 Find the time complexity?

(a) $T(n) = 4T(n-1)$ if $n > 0$ and 1 otherwise
(b) $T(n) = T(n-1) + \log n \quad n > 1$
(c) $T(n) = T(n-1) + 1/n$ if $n > 1$ and 1 if $n = 1$
(d) $T(n) = T(n-2) + 2\log n$ if $n > 1$ and 1 if $n = 1$
(e) $T(n) = T(n-2) + n^2$ if $n > 1$ and 1 if $n = 1$

Answer Key:

1. (a) $\Theta(n^2)$
   (b) $\Theta(n^2 \log n)$
   (c) $\Theta(2^n)$
   (d) (Master theorem does not apply)
   (e) $\Theta(n^2)$
   (f) (Master theorem does not apply)
   (g) $n^{0.51}$
   (h) Master theorem does not apply
   (i) $\Theta(n!)$
   (j) $\Theta(\sqrt{n})$
   (k) $\Theta(n^{\log 3})$
   (l) $\Theta(n)$
   (m) $\Theta(n^2)$
   (n) $\Theta(n \log n)$
   (o) $\Theta(n \log n)$
   (p) $\Theta(n^2 \log n)$
   (q) $\Theta(n^2)$
   (r) (Master theorem does not apply)
   (s) $\Theta(n^2)$
   (t) $\Theta(n^2)$

2. (a) $O(4^n)$ (b) $O(n \log n)$
   (c) $O(\log n)$ (d) $O(n \log n)$
   (e) $O(n^3)$

CHAPTER 03

Divide and Conquer

3.1 Introduction

In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

- **Divide** the problem into a number of sub problems that are smaller instances of the same problem.
- **Conquer** the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
- **Combine** the solutions to the sub problems into the solution for the original problem.

When the sub problems are large enough to solve recursively, we call that the **recursive case**. Once the sub problems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the **base case**.

Applications of Divide and Conquer

1. Quick sort
2. Strassen's algorithm for matrix multiplication
3. Merge sort
4. Counting inversions
5. Binary search
6. Finding Min and Max

3.2 Quick Sort

The quick sort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quick sort is often the best practical choice for sorting because it is highly efficient on the average: $\Theta(n\log n)$ and the constant factors hidden in the nlogn notation are quite small. It also has the advantage of sorting in place and it works well even in virtual-memory environments.

Quick sort is an in place sorting algorithm (it does not take extra space more than log$n$).

In merge sorting the array is divided blindly, but in quick sort the partition is done in an extreme intelligent manner using pivot element, such that all elements to the left of pivot are less than pivot and all elements to the right of pivot are greater than pivot. This is done recursively. Hence there is no need to merge (combine) the results.

This algorithm takes constant time better than merge sort as there is no combining (partial divide and conquer application)

3.2.1 Three Step DAC Process

- **Divide:** The array $A[p...r]$ is partitioned (rearranged) into two nonempty sub arrays $A[p...q]$ and $A[q+1 \ldots r]$ such that each element of $A[p \ldots q]$ is less than or equal to each element of $A[q+1 \ldots r]$. The index q is computed as part of this partitioning procedure.
- **Conquer:** The two sub arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are sorted by recursive calls to quick sort.
- **Combine:** Since the sub arrays are sorted in place, no work is needed to combine them: the entire array $A[p \ldots r]$ is now sorted.

The following procedure implements Quick sort

QUICKSORT (A, p, r)

1. If ($p < r$)
2. q = PARTITION (A, p, r)
3. QUICKSORT (A, p, $q-1$)
4. QUICKSORT (A, $q+1$, r)

Here parameters for Initial call to QUICKSORT is given as QUICKSORT (A, 1, A.length), where A.length specifies the size of the array to be sorted

Partitioning Procedure

Partition algorithm returns the correct position of the pivot element that we have chosen. At every repetition of Partition algorithm pivot element goes to its correct place. By repeating the same algorithm and calling it recursively we get the sorted array. There are different ways that we can choose a pivot element from the array.

- Always First element
- Always Last element
- Always Middle element as pivot
- Lastly Always median as pivot

```
PARTITION (A, p, r)
X = A[r]                    /* last element is chosen as pivot
i = p - 1
      For j = p to r - 1
{
      If (A[j] ≤ x)
                            /* this condition is true whenever element smaller than pivot is found */
      {
            i = i + 1
            swap (A[i], A[j])
      }
}
```

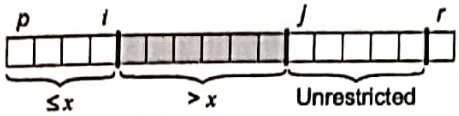
Swap ($A[i+1]$, $A[j]$) /*swap pivot and i^{th} element

Return $i + 1$ /* pivot position is returned to QUICKSORT algorithm.

Partition always selects the last element A[r] in the sub array $A[p \ldots r]$ as the pivot – the element around which to partition. As the procedure executes, the array is partitioned into four regions, some of which may be empty:

Loop invariant

- All entries in $A[p...i]$ are ≤ pivot.
- All entries in $A[i+1...j-1]$ are > pivot.
- $A[r]$ = pivot.



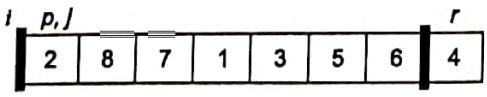
The four regions maintained by the procedure Partition on a subarray $A[p \ldots r]$. The values in $A[p \ldots i]$ are all less than or equal to x, the values in $A[i+1...j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j...r-1]$ can take on any values.

At the end of for loop the last element is pivot. Before pivot and after 'i^{th}' location all are greater than pivot from '$i+1$' onwards. Now we swap the pivot and 'i^{th}' element. Hence pivot goes to its correct position after one time execution of partition algorithm. This partition algorithm does maximum of n swaps (only 1 swap in best case i.e. swaps with itself). Every element need to be compared $(n-1)$ times in best and worst case. Therefore time complexity is $\Theta(n)$.

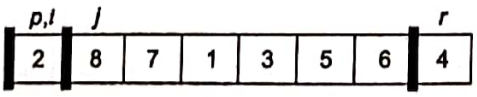
NOTE: While the loop is running, if $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot, then increment only j.

Consider the operation of partition algorithm on array of 8 elements. Figure the operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x.

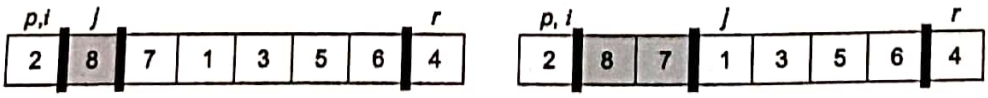
(a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions.



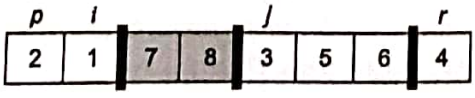
(b) The value 2 is "swapped with itself" and put in the partition of smaller values.



(c) The values 8 and 7 are added to the partition of larger values.

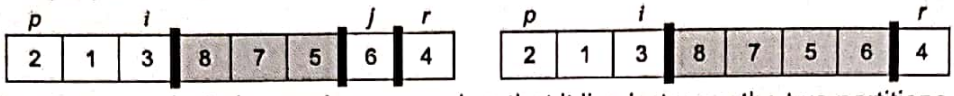


(d) The values 1 and 8 are swapped, and the smaller partition grows.

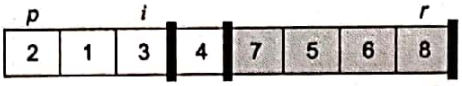


(e) The values 3 and 7 are swapped, and the smaller partition grows.

(f) The larger partition grows to include 5 and 6, and the loop terminates.



(g) In lines 7-8, the pivot element is swapped so that it lies between the two partitions.



$A[r]$ is the pivot element and is stored in p. Once the partition algorithm finishes execution the pivot element x goes to its correct place. (Here in this example 4 is the pivot).

The elements to the left of pivot x are { 2,1,3 } and less than 4 and to the right of pivot are {7, 6, 5, 8} and are greater than pivot.

3.2.2 Performance of Quick Sort

The running time of quick sort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort ($O(n^2)$).

Worst-case Partitioning

The worst-case behavior for quick sort occurs when the partitioning routine produces one sub problem with $(n-1)$ elements and other one with 0 elements. The partitioning costs $\Theta(n)$ time.

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

By using the substitution method we find that time complexity in this case of quick sort evaluates to $\Theta(n^2)$. Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Worst-case running time of quick sort is no better than that of insertion sort.

Worst case occurs when the array is already sorted

- **Insertion sort:** $O(n)$
- **Quick sort :** $\Theta(n^2)$

Worst case occurs when the array is reverse sorted order Quick sort takes $\Theta(n^2)$.

NOTE: When array is already sorted, nearly sorted or almost sorted then insertion sort is best.

Best-case Partitioning

Best case of quick sort occurs when the pivot divides the sub array exactly into two sub parts. PARTITION produces two sub problems, each of size no more than $n/2$, since one is of size floor $(n/2)$ and one of size ceiling $(n/2) - 1$. In this case, quick sort runs much faster.

$$T(n) = 2T(n/2) + \Theta(n)$$

By case 2 of the master theorem, this recurrence has the solution $\Theta(n\log n)$

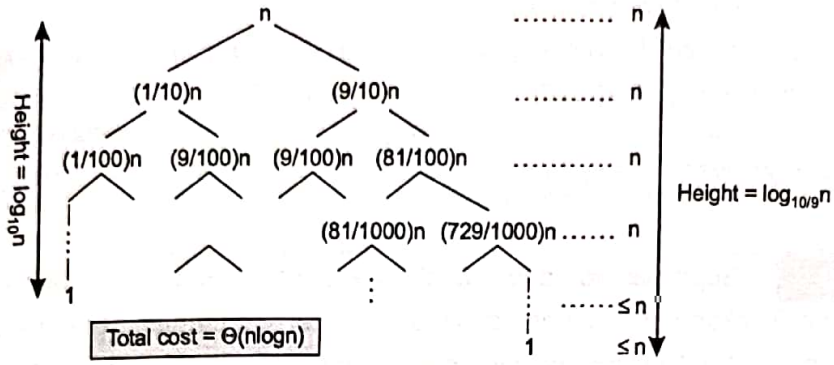
Balanced Partitioning

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which seems. But it is actually balance partition as we solve the recurrence relation

$$T(n) = T(9n/10) + T(n/10) + cn$$

Figure below shows the recursion tree for this recurrence. Notice that every level of the tree has cost cn, until the recursion reaches a boundary condition at depth $\log_{10} n = \Theta(\log n)$. The recursion terminates at depth $\log_{10/9} n = \Theta(\log n)$. The total cost of quick-sort is therefore $O(n \log n)$.

In fact, any split of constant proportionality yields a recursion tree of depth $\Theta(\log n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \log n)$ whenever the split has constant proportionality.



A recursion tree for QUICK SORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \log n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term. Some of the examples of proportionality splitting which have $\Theta(\log n)$ run time are:

- $T(n) = T(n/2) + T(n/2) + n$
- $T(n) = T(n/5) + T(4n/5) + n$
- $T(n) = T(99n/100) + T(n/100) + n$
- $T(n) = T(n/4) + T(3n/4) + n$

NOTE: The behavior of quick sort depends on the relative ordering of elements given as input, and not by the particular values in the array.

In average case the partition algorithm produces mix of good and bad splits. The running time of quick sort, when levels alternate between good and bad splits, is like running time for good splits alone: $\Theta(n \log n)$.

3.2.3 Randomized Version of Quick Sort

We use randomization to improve the performance of Quicksort against those worst-case instances.

We use the following procedure Randomized-Partition to replace Partition. It randomly picks one element in the sequence, swaps it with the first element, and then calls Partition.

- Randomized-Partition (A, p, q), which works on $A[p \ldots q]$
  r = Random (p, q) (pick an integer between p and q uniformly at random) and exchange $A[p]$ with $A[r]$
- Return Partition (A, p, q)

So essentially we randomly pick an element in the sequence and use it as a pivot to partition.

Randomized-Quicksort (A, p, q), which sorts $A[p \ldots q]$

If $p < q$

s = Randomized-Partition (A, p, q)
Randomized-Quicksort (A, p, $s - 1$)
Randomized-Quicksort (A, $s + 1$, q)

Use induction to prove its correctness. Given any input instance A, its running time $t(A)$ is now a random variable that depends on the pivots it picks randomly. Given any input sequence A of length n, Randomized-Quicksort ($A[1 \ldots n]$) has expected running time $O(n \log n)$.

This of course implies that the worst-case expected running time of Randomized-Quicksort is $O(n \log n)$. Actually, we will see from the analysis that the order of the elements in A at the beginning (almost) does not affect the running time of Randomized-Quicksort at all.

We start with some intuition of why its expected running time is $O(n \log n)$. This is by no means a proof. So we know Quicksort has worst-case running time $\Omega(n^2)$ because if the input is already sorted, every pivot we pick is the smallest/largest of the sequence, which leads to highly unbalanced subsequences and $\Omega(n^2)$ running time. But in randomized quick sort we randomly pick the pivot element and in best cases we may pick the median of the input.

Example-3.1 **Suppose we have a $O(n)$ time algorithm that finds median of an unsorted array. Now consider a Quicksort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst case time complexity of this modified Quicksort?**

Solution:
If we use median as a pivot element, then the recurrence for all cases becomes $T(n) = 2T(n/2) + O(n)$.
Time complexity by solving above recurrence relation = $O(n \log n)$.

Example-3.2 **Given an unsorted array. The array has this property that every element in array is at most k distance from its position in sorted array where k is a positive integer smaller than size of array. Which sorting algorithm can be easily modified for sorting this array and what is the obtainable time complexity?**

Solution:
Heap sort with time complexity $O(n \log K)$

Example-3.3 **In quick sort, for sorting n elements, the $(n/4)^{th}$ smallest element is selected as pivot using an $O(n)$ time algorithm. What is the worst case time complexity of the quick sort?**

Solution:
The recursion expression becomes: $T(n) = T(n/4) + T(3n/4) + cn$ after solving the above recursion, we get $\Theta(n \log n)$.

3.3 Strassen's Matrix Multiplication

Suppose we want to multiply two N × N square matrices, A and B. If $A = (a_{ij})$ and $B = (b_{ij})$ are matrices, then in the product C = A.B we will therefore have N^2 elements. Each one of these elements is naturally expressed as the sum of N products, each of an element of A with one of B. Thus we have

$$C_{i,j} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

And the number of multiplications involved in producing the product in this way is n^3. This brute force multiplication is shown below.

SQUARE-MATRIX-MULTIPLY (A, B)

1. n = a. ROWS

2. let C be a new $n \times n$ matrix
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $C_{ij} = 0$
6. for $k = 1$ to n
7. $C_{ij} = C_{ij} + a_{ik} \cdot b_{kj}$
8. return C

Because each of the triply-nested for loops runs exactly n iterations, and each execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY procedure takes n^3 time.

Suppose we want to multiply two matrices of size 2 x 2: for example A x B = C.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$C_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$C_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$C_{21} = a_{21}b_{11} + a_{22}b_{21}$$

and
$$C_{22} = a_{21}b_{12} + a_{22}b_{22}$$

A simple Divide and Conquer Algorithm

Given N x N matrices A and B for which N is factorable into R and S(N = R x S) you can similarly write them as R x R matrices whose components are each S x S matrices in the same general way without changing the definition of their matrix multiplication at all.

Explicitly, in the 4 by 4 case, if we describe A and B as

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \text{ and } B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

Then their matrix product is exactly the same as the product of the matrices

$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$, where we have

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}, A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, \text{ and } A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

and the same kind of thing for B.

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Square-Matrix-Multiply-Recursive (A, B)

1. $n = A.$ rows
2. Let C be a new $n \times n$ matrix
3. if $n == 1$
4. $c_{11} = a_{11} \times b_{11}$
5. else partition A, B and C as in equations

6. C_{11} = Square-Matrix-Multiply-Recursive (A_{11}, B_{11}) + Square-Matrix-Multiply-Recursive (A_{12}, B_{21})
7. C_{12} = Square-Matrix-Multiply-Recursive (A_{11}, B_{12}) + Square-Matrix-Multiply-Recursive (A_{12}, B_{22})
8. C_{21} = Square-Matrix-Multiply-Recursive (A_{21}, B_{11}) + Square-Matrix-Multiply-Recursive (A_{22}, B_{21})
9. C_{22} = Square-Matrix-Multiply-Recursive (A_{21}, B_{12}) + Square-Matrix-Multiply-Recursive (A_{22}, B_{22})
10. return C

The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls:

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2)$$
$$= 8T(n/2) + \Theta(n^2)$$

Time complexity using DAC is $\Theta(n^3)$ and space complexity is $2n^2$ for inputs, $\log n$ for stack and n^2 for output matrix. Comparing without divide and conquer with DAC is better, because both the methods have the same time complexity i.e. both have running time of $\Theta(n^3)$, but extra space in with DAC is $\log n$ (for stack).

In without DAC stack space is O(1).

Strassen's Improvement

If we can multiply 2 by 2 matrices using only 7 multiplications instead of the usual 8, we can copy (use) that into multiplying 4 by 4 matrices using 7 multiplications of 2 by 2 matrices each of which requires 7 multiplications of numbers, for a total of 49 multiplications.

Furthermore, by iterating this fact, we can multiply $2^k \times 2^k$ matrices with 7^k multiplications of numbers, so long as we can handle 2×2 matrices with 7 multiplications in a way that does not use commutativity. To overcome this strassens proposed new method which takes less time than $\Theta(n^3)$. His method used only 7 multiplications and 18 additions (cost of addition is very less compared to multiplication)

Unlike the DAC approach which takes 8 multiplications and 4 additions. The following is the recurrence relation which takes less than $\Theta(n^3)$.

$$T(n) = 7T(n/2) + 18(n/2)^2 \text{ if } n >= 2 \text{ otherwise } O(1)$$

This results in $O(n^{2.81})$ time

Without DAC	With DAC	Strassen's Improvement
Space complexity:	**Space complexity:**	**Space complexity:**
Input : $2n^2$	Input : $2n^2$	Input : $2n^2$
Output: n^2	Output: n^2	Output: n^2
Stack: Nill	Stack: $\log n$	Extra space required than recursive DAC procedure.
Time complexity: $\Theta(n^3)$	**Time complexity:** $\Theta(n^3)$	Time complexity: $\Theta(n^{2.807})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The submatrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

3.4 Merge Sort

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n = 2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

Key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE (A, p, q, r), where A is an array and p, q, and r are indices into the array such that $p \le q < r$. The procedure assumes that the sub arrays $A[p...q]$ and $A[q + 1...r]$ are in sorted order. It merges them to form a single sorted sub array that replaces the current sub array $A[p...r]$.

NOTE: Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged.

MERGE (A, p, q, r)

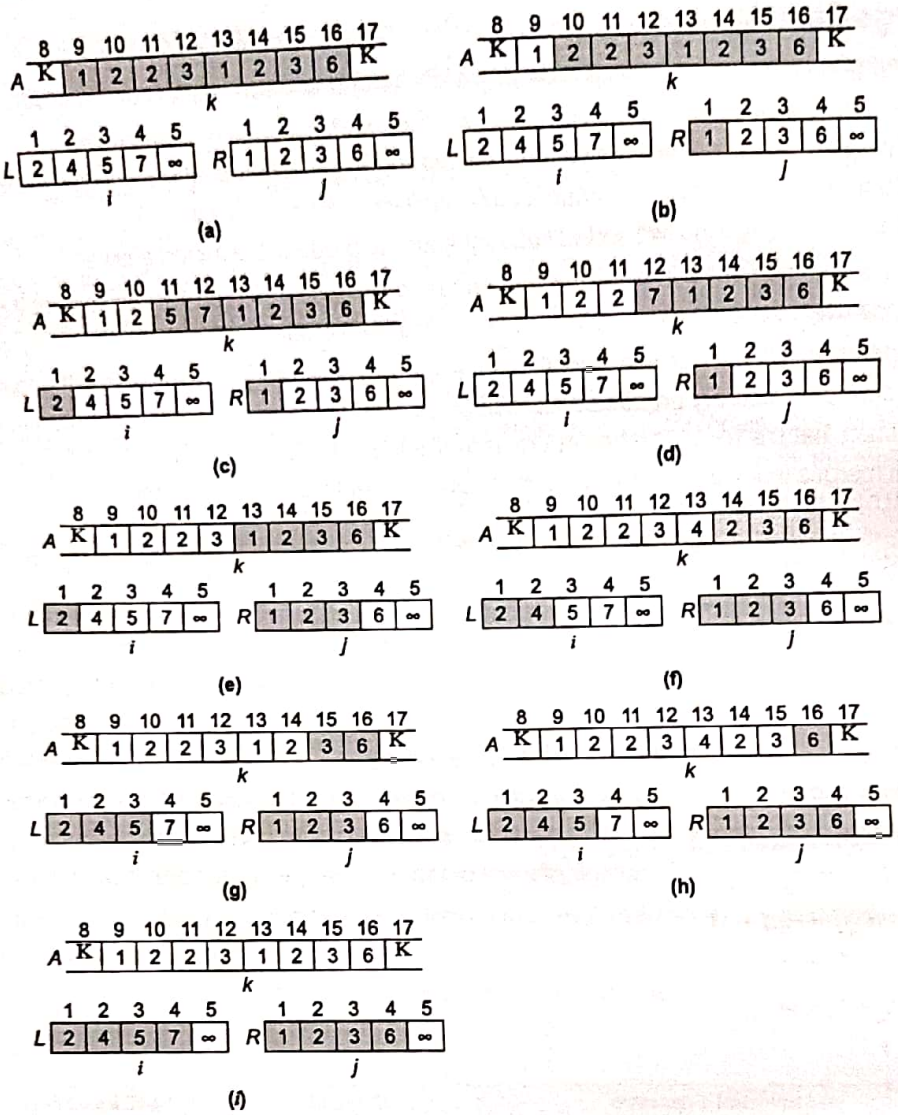
1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Let $L[1 \ldots n_1 + 1]$ and $R[1...n_2 + 1]$ be new arrays
4. *for* $i = 1$ to n_1
5. $\quad L[i] = A[p - i - j]$
6. for $j = 1$ to n_2
7. $\quad R[j] = A[q + 1]$
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. for $k = p$ to r
13. $\quad$ if $L[i] \le R[j]$
14. $\quad\quad A[k] = L[i]$
15. $\quad\quad i = i + 1$
16. $\quad$ else $A[k] = R[j]$
17. $\quad\quad j = j + 1$

Line1 computes the length n_1 of the sub array $A[p. . .q]$, and line 2 computes the length n_2 of the sub array $A[q + 1. . .r]$.

We create arrays L and R("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively.

The **for** loop of lines 4–5 copies the sub array $A[p...q]$ into $L[1...n_1]$, and the for loop of lines 6–7 copies the sub array $A[q + 1...r]$ into $R[1...n_2]$.

Lines 8–9 put the sentinels at the ends of the arrays L and R. (we choose sentinel as infinity so that we can mark it as the end of the array)



The operation of lines 10–17 in the call MERGE (A, 9, 12, 16), when the subarray A[9, 16] contains the sequence (2, 4, 5, 7, ∞), and the array R contains (1, 2, 3, 9,∞). Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in A[9...16], along with two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. (a)–(b) the arrays A, L, and R and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

The arrays and indices at termination. At this point, the subarray in A[9...16] is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

After copying and inserting sentinels , the array L contains {2, 4, 5, 7, ∞} and the array R contains {1, 2, 3, 6, ∞}. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in A[9. . .16], along with the sentinels. Heavily shaded positions in A contain values that will be copied over and heavily shaded positions in L and R contain values that have already been copied back into A.

To see that MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, we observe that each of lines 1–3 and 8–11 takes constant time. The for loops of 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time and there are n iterations of the for loop of lines 12–17, each of which takes constant time

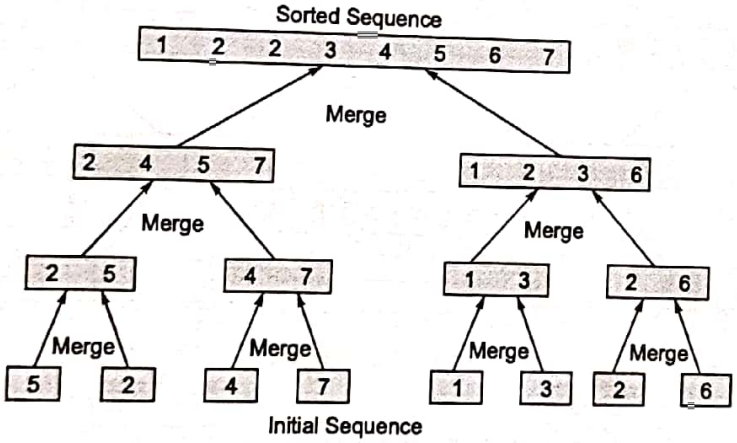
Figure (a)–(h) the arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A, p, r) sorts the elements in the subarray $A[p...r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p...q]$ into two subarrays: $A[q + 1...r]$, containing $\lfloor n = 2 \rfloor$ elements, and $A[q + 1...r]$, containing $\lfloor n = 2 \rfloor$ elements.

MERGE-SORT (A, p, r)

1. if $p < r$
2. $q = \lfloor (p + r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT $(A, q + 1; r)$
5. MERGE (A, p, q, r)

To sort the entire sequence $A = \{A[1], A[2],...A[n]\}$, we make the initial call MERGE-SORT (A,1,A. length), where once again A.length = n. Figure illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1 item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n = 2$ are merged to form the final sorted sequence of length n.



The operation of merge sort on the array A = (5, 2, 4, 7, 1, 3, 2, 6). The length of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Analysis of Merge Sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that original size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. When we have number of elements as $n > 1$ then

Divide: The dived step just computes the middle of the subarray, which takes constant time.

Thus, $D(n) = \Theta(1)$

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

The recurrence for the worst-case running time $T(n)$ of merge sort:

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

The time complexity sums up to $\Theta(n\text{Log}n)$.

Merge procedure time complexity: Maximum of (Number of comparisons, number of total moves)

NOTE

- Merge sort is not an in place sorting algorithm because in the merge algorithm it is taking $O(n)$ extra space.
- Merge sort is also not advisable for small size arrays because of unnecessary divisions and combining.

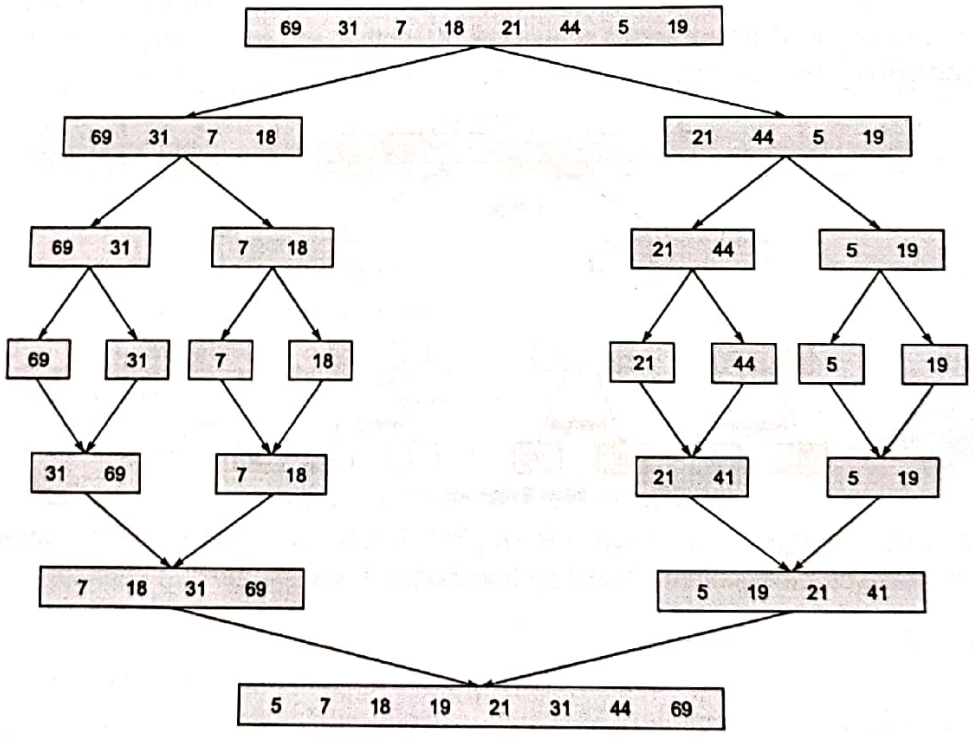
Outplace: $2T(n/2) + n$

In place: $2T(n/2) + n^2$ this has time complexity of $(\Theta(n^2))$

***Example* - 3.4** Apply merge sort for the following list of elements:

69, 31, 7, 18, 21, 44, 5, and 19

Solution:



3.5 Insertion Sort

This algorithm is best and efficient algorithm for sorting a small number of elements.

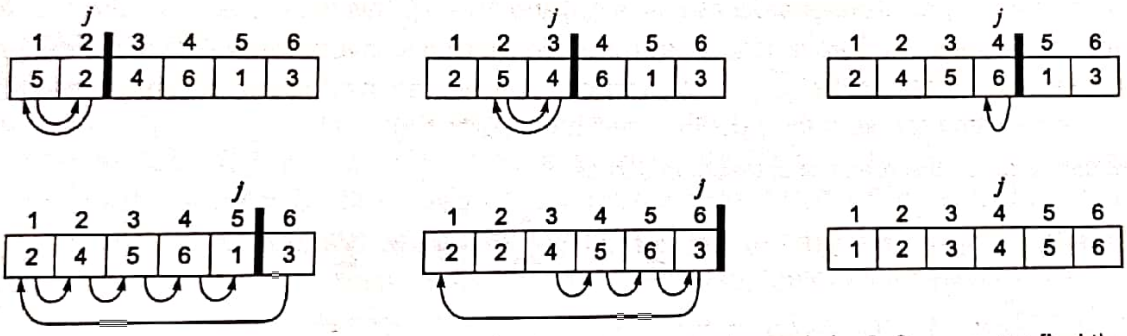
Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To nd the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

The algorithm sorts the input numbers in place: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

```
INSERTION-SORT (A)
1.  For j = 1 to A length
2.  Key = a[j]
3.      // insert a[j] into the sorted sequence A[1. . . j – 1]
4.      l = j – 1
5.      While i > 0 and A[i] > key
6.          A[i + 1] = A[i]
7.          l = i - 1
8.      A[i + 1] = key
```

The index *j*-indicates the "current card" being inserted into the hand. At the beginning of each iteration of the for loop, which is indexed by *j*, the sub array consisting of elements A[1 : : *j* – 1] constitutes the currently sorted hand, and the remaining sub array A[*j* + 1. . . *n*] corresponds to the pile of cards still on the table. In fact, elements A[1. . . *j* – 1] are the elements originally in positions 1 through *j* – 1, but now in sorted order.

Example: Following Figure shows the operation of INSERTION-SORT on the array A = (5, 2, 4, 6, 1, 3). Each part shows what happens for a particular iteration with the value of *j* indicated. *j* indexes the "current card" being inserted into the hand.



The **best case** ($O(n)$) occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we find that $A[i]$ less than or equal to the key when i has its initial value of ($j \leq 1$). In other words, when $i = j - 1$, always find the key $A[i]$ upon the first time the WHILE loop is run.

The **worst-case** ($O(n^2)$) occurs if the array is sorted in reverse order i.e., in decreasing order. In the reverse order, we always find that $A[i]$ is greater than the key in the while-loop test. So, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j \leq 1]$ and so $t_j = j$ for $j = 2, 3, \ldots, n$. Equivalently, we can say that since the while-loop exits because i reaches to 0, there is one additional test after ($j \leq 1$) tests.

The **average case** is as bad as the worst case. ($O(n^2)$)

3.6 Counting Inversions

Let A[1...*n*] be an array of *n* distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

Consider a small array of size 5 elements {2, 3, 8, 6, 1}. The five inversions of {2, 3, 8, 6, 1} are (1, 5), (2, 5), (3, 4), (3, 5), (4, 5).

The reverse-ordered list $< n, n-1, \ldots, 1 >$ has the most inversions and each of the ${}^nC_2$ pairs of elements will be an inversion.

Total number of inversions, $(n-1)+(n-2)+...+1 = n(n-1)/2$ where the 1st element has $(n-1)$ inversions, the 2nd $(n-2)$, and so on.

Insertion sort runs in $\Theta(n + f(n))$ time, where $f(n)$ denotes the number of inversion initially present in the array being sorted. To see this, consider the pseudocode for insertion sort:

INSERTION-SORT(A)	cost	times
1. for $j \leftarrow 2$ to length [A]	c_1	n
2. do key $\leftarrow$ A[j]	c_2	$n-1$
3. $i \leftarrow j-1$	c_4	$n-1$
4. while $i > 0$ and A[i] > A[$i+1$]	c_5	$\sum_{j=2}^{n} t_j$
5. do A[$i+1$] $\leftarrow$ A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
6. $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
7. A[$i+1$] $\leftarrow$ key	c_8	$n-1$

$t_j \rightarrow$ Denotes the number of times the while loop test in line 4 is executed for that value of j.

Everything except the **while** loop requires $\Theta(n)$ time. We now observe that every iteration of the **while** loop swaps an adjacent pair of out-of-order elements A[i] and A[$i+1$]. This decreases the number of inversions in A by exactly one since $(i, i+1)$ will no longer be an inversion (the other inversions are not affected). Since there is no other means of decreasing the number of inversions of A, we see that the total number of iterations of the **while** loop over the entire course of the algorithm must be equal to $f(n)$.

We can express the run time of insertion sort as:

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

$$= C_1 \sum_{j=2}^{n} (t_j - 1) + C_2 n + C_3$$

The number of times the contents of the **while** loop executes for index j is $(t_j - 1)$. The loop only executes if $0 < i < j$ and A[i] > A[j], which means (i, j) is an inversion. The loop removes only one inversion at a time, so in $(t_j - 1)$ iterations, all inversions for element A[j] have been removed. The total number of inversions is

$$f(n) = \sum_{j=2}^{n} (t_j - 1)$$

The total runtime can now be expressed as

$$T(n) = C_1 f(n) + C_2 n + C_3 > C_1 f(n) = \Omega(f(n))$$

Therefore, the running time of insertion sort is lower bounded by the number of inversions. To count the number of inversions in an array A, we modify merge sort so that it counts inversions as it sorts.

COUNT-INVERSIONS(A, p, r)

1 if $p < r$
2 then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 $l \leftarrow$ COUNT-INVERSIONS (A, p, q)
4 $r \leftarrow$ COUNT-INVERSIONS (A, q, r)
5 $m \leftarrow$ MERGE-COUNT (A, p, q, r)

```
6  return l + r + m
7  else
8  return 0
MERGE-COUNT(A, p, q, r)
1  n ← q − p + 1
2  n ← r − q
3  create arrays L[1...n1 + 1] and R[1...n2 + 1]
4  for i ← 1 to n1
5     do L[i] ← A[p + i − 1]
6  for j ← 1 to n2
7     do R[j] ← A[q + j]
8  L[n1 + 1] ← ∞
9  R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 l ← 0                  // for counting inversions
13 for k ⇐ p to r
14 do if L[i] ≤ R[j]
15       then A[k] ← L[i]
16       i ← i + 1
17 else A[k] ← R[j]
18 j ← j + 1
19 l ← l + (n1 − i)       //increment by # of elements still in L
```

At the time COUNT(*A, p, q, r, inv)* is called, every element in the $L = A[p .. q]$ is still in a lower indexed array cell compared to any element of $R = A[q + 1... r]$. Thus, whether COUNT decides to place an element of $A[q + 1 ... r]$ before an element of $A[p...q]$, it discovers an inversion. To show correctness, we must show that every inversion is counted exactly once. Since L and R are sorted, there are no inversions in L or R, only inversions between L and R. If an element r from list R is selected ahead of k items in list L, then r is less than the remaining k elements $l_1, l_2, ..., l_k$ in list L, corresponding to k inversions. Selecting r to be placed next in the sorted merged list removes all k inversions. No new inversions are created in the merging step, since the resulting array $A[p...r]$ is sorted and therefore has no inversions. No inversion can be counted twice, since removing an inversion sorts L and R into one combined list, and inversions can only exist between lists. Our new modified merge routine counts all inversions correctly in $O(n \lg n)$ time.

3.7 Binary Search

Description

Binary tree is a application of divide and conquer search algorithm. Ti inspects the middle element of the sorted list. If equal to the sought value, then the position has been found. Otherwise, if the key is less than the middle element, do a binary search on the first half, else on the second half.

Algorithm

Algorithm can be implemented as recursive or non-recursive algorithm.

ALGORITHM BS (A[0 . . . n − 1], key)

// implements non-recursive binary search

```
// i/p: Array A in ascending order, key k
// o/p: Returns position of the key matched else –1
l = 0
r = n – 1
while l ≤ r do
        m = (l + r)/2 if key == A[m]
                return m
        else
                if key < A[m]
                        r = m – 1
        return –1
                else
                        l = m + 1
```

Analysis

- **Input size:** Array size, n
- Basic operation: key comparison
- Depend on

  Best – key matched with mid element

  Worst – key not found or key sometimes in the list
- Let C(n) denotes the number of times basic operation is executed. Then C worst(n) = Worst case efficiency. Since after each comparison the algorithm divides the problem into half the size, we have C worst(n) = C worst($n/2$) + 1 for $n > 1$ C(1) = 1
- Solving the recurrence equation using master theorem, to give the number of times the search key is compared with an element in the array, we have:

  $C(n) = C(n/2) + 1$

  $a = 1$

  $b = 2$

  $f(n) = n^0$; $d = 0$ case 2 holds:

  $C(n) = \Theta(n^d \log n)$

  $= \Theta(n 0 \log n)$

  $= \Theta(\log n)$

Applications of binary search:

- Number guessing game
- Word lists/search dictionary etc

Advantages:

- Efficient on very big list
- Can be implemented iteratively/recursively

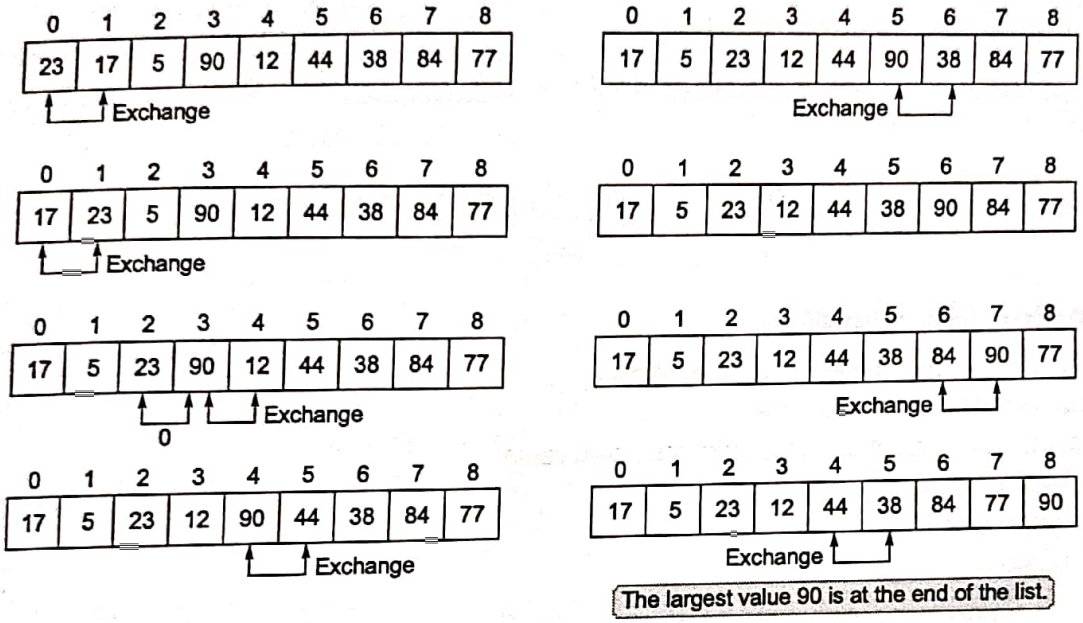
Limitations:

- Interacts poorly with the memory hierarchy (more cache miss).
- Requires given list to be sorted.
- Due to random access of list element, needs arrays instead of linked list.

3.8 Bubble Sort

The bubble sort is similar to the selection sort, but more than one value is exchanged on each pass through the array. At the end of each pass, we end up with the largest value at the upper end of the array (highest array index), instead of the smallest value at the beginning as with the selection sort.

We also have a termination condition so that once the array is sorted, we can stop instead of making all the passes that would be needed with a fully unsorted array. This can be more efficient than the selection sort, which always continues to the bitter end, even if it is sorted to begin with.



This figure illustrates the steps in one pass of the bubble-sort algorithm. Starting at the left hand side of the array (lowest index), compare the first two elements. If the first is bigger than the second, exchange (swap) them. Then compare the 2nd with the third. If the 2nd is bigger than the 3rd, swap them. Continue doing this until the last two elements in the array get compared, and swapped if needed.

At the end of this iteration, the largest element in the array has "bubbled" up to the top of the array. For the 2nd pass, we do exactly the same thing, but ignore the last element. For the 3rd pass, we'll ignore the last two elements, etc.

After completing all the passes, the array will be sorted.

One advantage to the bubble-sort is the ability to check and see if any swaps are made during any specific pass. If we make a pass through the array, and don't have to make any swaps, that means that the array is sorted, and we can quit. Thus, if the array is sorted to begin with, one pass of the bubble-sort will be made, no swaps will happen, and we will know to quit.

This is much more efficient than the selection sort, where all of it's passes will be made, no matter the original state of the array. So in the best case (array already sorted), bubble sort is much more efficient than selection sort. In the worst case (array anti-sorted), both algorithms are approximately N^2 and cost about the same.

```
Void bubbleSort (in tar[ ]))
{
        for (int i = (ar.length = 1); I > = 0; i == i)
        {
                for (int j – 1; j <= i; j++)
                {
                        if (ar [j – 1] > ar [j])
                        {
                                int tempt = ar [j – 1];
                                ar [j – 1] = ar [j];
                                ar [j] = temp;
                        }
                }
        }
}
```

O(1) O(i) $\sum_{i=0}^{i=n} O(i)$

$$\sum_{i=0}^{i=n} O(i) = 1 + 2 + 3 + \ldots + (n-1) = O(n^2)$$

3.9 Finding Min and Max

Algorithm straightforward (*a*, *n*, max, min)
Input: array [a] with *n* elements
Output: max: maximum value, min: minimum value
Max = min = a[1]
for *i* = 2 to *n* do
begin
if (a[*i*]> max) then max = a[*i*]
else if (a[*i*] < min) then min = a[*i*]
end

Best case: Elements in increasing order, No. of comparisons = **(*n* – 1)**
Worst case: Elements in decreasing order, No. of comparisons = **2(*n* – 1)**
Average case: *a*(*i*) is greater than max half the time, No. of comparisons= **3*n*/2 – 2**

A Divide and Conquer Approach

In this approach we divide the given problem (input) into smaller problem and find the solution for the smaller sub problem. We do this recursively by splitting the list into two sub lists of equal size. (Assume that the initial list size is a power of two.) Find the maxima and minima of the subsists. Two more comparisons are then sufficient to find the maximum and minimum of the list.

Algorithm MaxMin (*i*, *j*, max, min)
input: array of N elements, *i* lower bound, *j* upper bound
output: max: largest value
min: smallest value.
if (*i* = *j*) then max = min = *a*[*i*]
else
if (*i* = *j* – 1) then
if (*a*[*i*] < *a*[*j*]) then
max = *a*[*j*]

```
                min = a[i]
        else
                max = a[i]
                min = a[j]
        else
                mid = (i + j)/2
                maxmin (i, mid, max, min)
                maxmin (mid + 1, j, max 1, min 1)
    if (max < max1) then max = max 1
    if (min > min1) then min = min 1
end
```

The following recurrence relation specifies the total number of comparisons required for finding the maximum and minimum element in the array.

$$C_n = \begin{cases} C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 & \text{for } n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

$$\begin{aligned} C_n &= 2C_{n/2} + 2 \\ &= 2(2C_{n/2} + 2) + 2 \\ &= 4C_{n/4} + 4 + 2 \\ &\vdots \\ &= 2^{k-1}C_2 + \sum_{i=1}^{k-1} 2^i = 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned}$$

When n is a power of 2, $n = 2^k$

In above recurrence relation, 2 specify the number of comparisons in comparing the solutions of sub problems. These will recursively gets added in the recurrence relation.

We assume that in a 1 element list the sole element is both the maximum and the minimum element.

Summary

- The quick sort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers.
- The running time of quick sort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.
- When array is already sorted, nearly sorted or almost sorted then insertion sort is best.
- The behavior of quick sort depends on the relative ordering of elements given as input, and not by the particular values in the array.
- Generally Strassen's Method is not preferred for practical applications for following reasons:

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.

- Merge sort is not an in place sorting algorithm because in the merge algorithm it is taking $O(n)$ extra space.
- Merge sort is also not advisable for small size arrays because of unnecessary divisions and combining.

Outplace: $2T(n/2) + n$
In place: $2T(n/2) + n^2$ this has time complexity of $(\Theta(n^2))$

Student's Assignments

Q.1 Merge sort uses
(a) Divide and conquer strategy
(b) Back tracking Approach
(c) Heuristic search
(d) Greedy approach

Q.2 Suppose we want to arrange the n numbers stored in an array such that all negative value occur before all positive ones, maximum number of exchanges required by using best algo is
(a) $n-1$ (b) n^2
(c) n (d) $n(n+1)/2$

Q.3 The average time required to perform a successful sequential search for an element in an array $A(1:n)$ is given by
(a) $\log_2 n$ (b) n^2
(c) $(n+1)/2$ (d) None of these

Q.4 How many comparisons are needed to sort an array of length 5 if selection sort is used and array is already in the opposite order?
(a) 15 (b) 20
(c) 10 (d) 1

Q.5 Which of the following is not good for linked list?
(a) Sorting (b) queues
(c) Binary reach (d) none of these

Q.6 Which sort will operate in quadratic time relative to the number of elements in the array (on the average)?
(a) quick sort (b) bubble sort
(c) heap sort (d) merge sort

Q.7 Sorting is useful for
(a) report generation
(b) responding to queries easily
(c) making searching easier and efficient
(d) All of these

Q.8 A sorting techniques that guarantees that records with the same primary key occurs in the same order in the sorted list as in the original unsorted list is said to be
(a) linear (b) external
(c) stable (d) consistant

Q.9 If the binary search algorithm determines that the search argument is in the upper half of the array, which of the following statements will set the appropriate variable to the appropriate value?
(a) Start sub = middle sub-1
(b) Start sub = middle sub+1;
(c) Stop sub = middle sub-1
(d) Stop sub = middle sub+1;

Q.10 Find the average number of comparisons in a binary search onm a sorted array of 10 consecutive integers starting from 1.
(a) 2.6 (b) 2.7
(c) 2.8 (d) 2.9

Q.11 The array given a s input to a quick sort algorithm is already sorted. What will be the running time of quick sort if the pivot is taken to be (i) the first element (ii) median of three (first, middle and last) elements?

(a) (i) : $O(n)$, (ii) $(n \log n)$
(b) (ii) : $O(n^2)$, (ii) : $O(n^2)$
(c) (iii) : $O(n^2)$, (ii) $O(n)$
(d) (i) : $O(n^2)$, (ii) : $O(n \log n)$

Q.12 Given 2 sorted list of sizes 'm' and 'n' respectively. Number of comparisons need in the worst case by the merge sort algorithm will be
(a) $m * n$ (b) max (m, n)
(c) min (m, n) (d) $m + n - 1$

Q.13 If the address of A [1, 1} and A [1, 2] are 1000 and 1010 respectively and each element occupies 2 bytes, then the array has been stored in
(a) Row major (b) Column major
(c) Compiler dependent (d) None of these

Q.14 A machine take 200 second to sort 200 names, using bubble sort. In 800 seconds it can approximately sort
(a) 400 names (b) 800 names
(c) 750 names (d) none of these

Q.15 Which of the following pairs both sorting algorithm are stable
(a) Quick sort and insertion sort
(b) insertion sort and bubble sort
(c) Quick sort and heap sort
(d) Quick sort and bubble sort

Q.16 Consider the following array with 7 elements for insertion sort ?

25, 15, 30, 9, 99, 20, 26

In how many passes, the given sequence will be sorted?
(a) 4 pass (b) 5 pass
(c) 6 p7ass (d) More than 6 pas

Q.17 If one uses straight merge sort algorithm to sort the following elements in ascending order:

20, 47, 15, 8, 9, 4, 40, 30, 12, 17

then the order of these elements after second pass of the algorithm is:
(a) 8, 9, 15, 20, 47, 4, 12, 30, 40
(b) 8, 15, 20, 47, 4, 9, 30, 40, 12, 17,
(c) 15, 20, 47, 4, 8, 9, 12, 30, 40, 17
(d) 4, 8, 9, 15, 20, 47, 12, 17, 30, 40

Answer Key:

1. (a)	**2.** (a)	**3.** (c)	**4.** (c)	**5.** (c)
6. (b)	**7.** (d)	**8.** (c)	**9.** (b)	**10.** (d)
11. (d)	**12.** (d)	**13.** (b)	**14.** (a)	**15.** (b)
16. (c)	**17.** (b)			

CHAPTER 04

Greedy Techniques

4.1 Introduction

Algorithm based on Greedy approach are typically used to solve an **optimization problem.** An Optimization problem is one in which, a set of input values is given which are required to be either maximize or minimize with respect to some constraints. Generally an optimization problem has *n*-inputs, we are required to obtain a subset of C that satisfies the given constraints or conditions. A subset $S \subseteq C$, that satisfies the given constraints, is called a **feasible** solution, that maximizes or minimizes a given objective function. The feasible solution is called an **optimal solution.**

A greedy algorithm proceeds step-by-step, by considering one input at a time. At each stage, a decision has to be made regarding whether a particular input (say x) chosen gives an optimal solution or not. The choice of selecting input x is being guided by the selection function (say **select**). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. If not then this input x is not added to the partial solution. The input once tried and is rejected is never considered again. In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say x, from input domain C.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set $S \leftarrow S \cup \{x\}$. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or C is exhausted whichever is earlier.

NOTE: From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function (either maximize or minimize, as the case may be), is called **optimal solution.**

Characteristics of Greedy Algorithm

- Used to solve optimization problem
- Most general, straightforward method to solve a problem.

- Easy to implement, and if exist, are efficient.
- Always makes the choice that looks best at the moment. That is, it makes a locally optimal choice expection of this choice can lead to an overall global optimal solution.
- Once any choice of input from C is rejected then it never considered again. Do not always yield an optimal solution; but for many problems they do.

4.2 Basic Examples of Greedy Techniques

Suppose you have been given Indian currency notes of all denominations, e.g. {1,2,5,10,20,50,100,500,1000}. The **problem is to find the minimum number of currency notes to make the required amount A, required for payment**. Further, it is assumed that currency notes of each denomination are available in sufficient numbers, so that one can choose as many notes of the same denomination as are required for amount A.

Example - 4.1 **Given notes of 1, 2,5,10,20,50,100,500 and 1000 find the minimum number of notes required to make the amount ₹ 354.**

Solution:

Intuitively, to begin with, pick-up a note of denomination D, satisfying the conditions.

(*i*) $D \leq 354$

(*ii*) if D1 is another denomination of a note such that $D1 \leq 354$, then $D1 \leq D$.

In other words, the picked-up note's denomination D is the largest among all the denominations satisfying condition (*i*) above.

We apply the above mentioned intuitive solution as follows: To deliver ₹ 354 with minimum number of currency notes, the notes of different denominations are chosen and rejected as shown below:

Chosen-Note-Denomination	Total-Value-So Far	Denomination Selected?
100	0 + 100 ≤ 354	Yes
100	100 + 100 ≤ 354	Yes
100	300 + 100 ≤ 354	Yes
50	300 + 50 ≤ 354	Yes
50	350 + 50 > 354	No
20	350 + 20 ≤ 354	No
20	350 + 20 > 354	No
10	350 + 10 ≤ 354	No
10	350 + 10 > 354	No
5	350 + 5 ≤ 354	No
5	350 + 5 > 354	No
2	350 + 2 < 354	Yes
2	352 + 2 = 354	Yes

The above sequence of steps is based on Greedy technique, which constitutes an algorithm to solve the problem. In short, in the above mentioned solution, we have used the strategy of choosing, at any stage, the maximum denomination note, subject to the condition that the sum of the denominations of the chosen notes should not exceed the required amount A = 354.

The above strategy is the essence of greedy technique.

Next, we consider an example in which for a given amount A and a set of available denominations, the greedy algorithm does not provide a solution, even when a solution by some other method exists.

Example-4.2 **Given notes of {20, 30, 50} find the minimum number of notes required to make the amount ₹ 90.**

Solution:

Apply greedy technique:

(*i*) First, pick up a note of denomination 50, because 50 ≤ 90. The amount obtained by adding denominations of all notes picked up so far is 50.

(*ii*) Now, we can not pick up a note of denomination 50 again. However, if we pick up another note of denomination 50, then the amount of the picked-up notes becomes 100, which is greater than 90. Therefore, we can not pick up any note of denomination 50 or above.

(*iii*) Therefore, we pick up a note of next denomination, viz., of 30. The amount made up by the sum of the denominations 50 and 30 is 80, which is less then 90. Therefore, we have to accept a note of denomination 30.

(*iv*) Again, we can not pick up another note of denomination 30, because it picked, then the sum of denominations of picked up notes, becomes 80 + 30 = 110, which is greater more than 90. Therefore, we do not pick up any note of denomination 30 or above.

(*v*) Next, we attempt to pick up a note of next denomination, viz., 20. But, in this case also the sum of the denomination of the picked up notes becomes 80 + 20 = 100, which is again greater than 90. Therefore, we do not pick up only note of denomination 20 or above.

(*vi*) Next, we attempt to pick up a note of still next lesser denomination. However, there are no more lesser denominations available.

Hence greedy algorithm fails to deliver a solution to the problem.

However, by some other technique, we have the following solution to the problem: First pick up a note of denomination 50 then two notes each of denomination 20.

Thus, we get 90 and it can be easily seen that at least 3 notes are required to make an amount of 90. Another alternative solution is to pick up 3 notes each of denomination 30.

Example-4.3 **Given notes of {10, 40, 60} find the minimum number of notes required to make the amount ₹ 80.**

Solution:

Using the greedy technique, to make an amount of 80, first, we use a note of denomination 60. For the remaining amount of 20, we can choose note of only denomination 10. And , finally, for the remaining amount, we choose another note of denomination 10. Thus, greedy technique suggests the following solution using 3 notes:

$$80 = 60 + 10 + 10.$$

However, the following solution uses only two notes: 80 = 40 + 40 Thus, the solutions suggested by Greedy technique may not be optimal.

4.3 Greedy Technique Formalization

To solve optimization problem using greedy technique, there is a need of the following data structures and functions:

1. A candidate set from which a solution is created. It may be set of nodes, edges in a graph etc. call this set as: **P** : Set of given values or set of candidates.
2. A solution set S (where $S \subseteq P$, in which we build up a solution. This structure contains those candidate values, which are considered and chosen by the greedy technique to reach a solution. Call this set as: **S** : Set of selected candidates (or input) which is used to give optimal solution.
3. A function 'solution' to test whether a given set of candidates give a solution (not necessarily optimal).
4. A selection function 'select' which chooses the best candidate form *P* to be added to the solution set S.
5. A function 'feasible' to test if a set S can be extended to a solution (not necessarily optimal)
6. An objective function 'ObjF' which assigns a value to a solution, or a partial solution.

A general form for greedy technique can be illustrated as:

```
Algorithm Greedy (P, n)
/* Input: A input domain (or Candidate set ) P of size n, from which solution is to be Obtained. */
// function select (P: candidate_set) return an element (or candidate).
// function solution (S: candidate_set) return Boolean
// function feasible (S: candidate_set) return Boolean
/* Output: A solution set S, where S ⊆ P, which maximize or minimize the selection criteria with respect to
given constraints */
{
    S ← ϕ      // Initially a solution set S is empty.
While (not solution(S) and P ≠ ϕ)
{
    x ← select (P) /* A "best" element x is selected from P which maximize or minimize the selection
                      criteria. */
    P ← P – {x}      /* once x is selected, it is removed from P
    if (feasible (S ∪ {x}) then   /* x is now checked for feasibility S ← S ∪ {x}
}
    If (solution (S))
    return S;
else
    return "No Solution"
}   // end of while
```

Now in the following sections we apply greedy method to solve some optimization problem such as Knapsack (fractional) problem, Minimum Spanning tree and Single source shortest path problem etc.

4.4 Knapsack (Fractional) Problem

The fractional Knapsack problem is defined as:

- Given a list of n objects say $\{I_1, I_2, \ldots I_n\}$ and a Knapsack (or bag).
- Capacity of Knapsack is M.
- Each object I_i has a weight w_i and a profit of p_i.
- If a fraction x_i (where $x_i \in \{0, \ldots, 1\}$) of an object I_i is placed into a knapsack then a profit of $p_i x_i$ is earned.

The **problem** (or Objective) is to fill a knapsack (up to its maximum capacity M) which maximizes the total profit earned.

Mathematically:

$$\text{Maximize (the profit)} = \sum_{i=1}^{n} p_i x_i \text{ ; subjected to the constraints}$$

$$= \sum_{i=1}^{n} w_i x_i \le M \text{ and } x_i \in \{0, \ldots, 1\}, 1 \le i \le n\}$$

Note that the value of x_i will be any value between 0 and 1 (inclusive). If any object I_i is completely placed into a knapsack then its value is 1 (i.e., $x_i = 1$), if we do not pick (or select) that object to fill into a knapsack then its value is 0 (i.e., $x_i = 0$), Otherwise if we take a fraction of any object then its value will be any value between 0 and 1. To understand this problem, consider the following instance of a knapsack problem:

Number of objects; n = 3 Capacity of Knapsack;

M = 20, $(p_1, p_2, p_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$.

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i / w_i that fits into the knapsack.

Let us apply all above 3 approaches on the given knapsack instance:

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$\left(1, \frac{2}{15}, 0\right)$	18+2+0 = 20	28.2
2	$\left(0, \frac{2}{3}, 1\right)$	0+10+10 = 20	31.0
3	$\left(0, 1, \frac{1}{2}\right)$	0+15+5 = 20	31.5

Approach 1: (Selection of object in decreasing order of profit):

In this approach, we select those object first which has maximum profit, then next maximum profit and so on. Thus we select 1st object (since its profit is 25, which is maximum among all profits) first to fill into a Knapsack, now after filling this object ($w_1 = 18$) into Knapsack remaining capacity is now 2(i.e., 20 – 18 = 2). Next we select the 2nd object, but its weight $w_2 = 15$, so we take a fraction of this object (i.e., $x_2 = 2/15$). Now Knapsack is full (i.e., $\sum_{i=1}^{3} p_i x_i = 28$) so 3rd object is not selected.

Hence we get total profit $\sum_{i=1}^{3} p_i x_i = 28$ units and the solution set $(x_1, x_2, x_3) = \left(1, \frac{2}{15}, 0\right)$.

Approach 2: (Selection of object in increasing order of weights)

In this approach, we select those object first which has minimum weight, then next minimum weight and so on.

Thus we select objects in the sequence 2nd then 3rd then 1st. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i = 31.0$ units and the solution set $(x_1, x_2, x_3) = \left(0, \frac{2}{3}, 1\right)$.

Approach 3: (Selection of object in decreasing order of the ratio p_i / W_i)

In this approach, we select those object first which has maximum value of p_i / W_i, that is we select those object first which has maximum profit per unit weight. Since $(p_1 / w_1, p_2 / w_2, p_3 / w_3,) = (13, 1.6, 1.5)$. Thus we select 2nd object first, then 3rd object then 1st object. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i = 31.5$ units and the solution set $(x_1, x_2, x_3) = \left(0, 1, \frac{1}{2}\right)$. Thus from above all 3 approaches, it may be noticed that

- Greedy approaches do not always yield an optimal solution. In such cases the greedy method is frequently the basis of a heuristic approach.
- Approach3 (Selection of object in decreasing order of the ratio p_i / w_i) gives a optimal solution for Knapsack problem.

A pseudo-code for solving knapsack problem using greedy approach is:

```
Greedy Fractional-Knapsack (P[1...n], W[1...n], X[1...n], M)
/* P[1...n] and W[1...n] contains the profit and weight of the n-objects ordered such that
X[1...n] is a solution set and M is the capacity of KnapSack*/
    {
1.      For i ← 1 to n do
2.          X[i] ← 0
3.          profit ← 0   // Total profit of item filled in Knapsack
4.          weight ← 0   // Total weight of items packed in KnapSack
5.          i ← 1
6.  While (Weight < M) // M is the Knapsack Capacity
    {
7.      if (weight + W[i] ≤ M)
8.      X[i] = 1
9.          weight = weight + W[i]
10.     else
11.         X[i] = (M – weight)/w[i]
12.         weight = M
13. Profit = profit + p[i] * X[i]
14. i++;
    }   // end of while
    }   // end of Algorithm
```

Running Time of Knapsack (Fractional) Problem

Sorting of n items (or objects) in decreasing order of the ratio P_i/W_i takes $O(n\log n)$ time. Since this is the lower bound for any comparison based sorting algorithm. Line 6 of **Greedy Fractional-Knapsack** takes $O(n)$ time. Therefore, the total time including sort is $O(n\log n)$.

Example - 4.4 Find an optimal solution for the knapsack instance $n = 7$ and $M = 15$, $(p_1, p_2,\ldots,p_7) = (10, 5, 15, 7, 6, 18, 3)$, $(w_1, w_2,\ldots, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Solution:

Greedy algorithm gives a optimal solution for knapsack problem if you select the object in decreasing order of the ratio p_i/W_i. That is we select those object first which has maximum value of the ratio p_i/W_i; for all $i = 1,\ldots, 7$. This ratio is also called profit per unit weight.

Since $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2},\ldots, \frac{p_7}{w_7}\right) = (5, 1.67, 3, 1, 6, 4.5, 3)$. Thus we select 5th object first, then 1st object, then 3rd (or 7th) object, and so on.

Approach	$(x_1,x_2,x_3,x_4,x_5,x_6,x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
Selection of object in decreasing order of the ratio p_i/wi	$\left(1,\frac{2}{3},1,0,1,1,1\right)$	$1+2+4+5+1+2 = 15$	$6+10+18+15+3+3.33 = 55.33$

4.5 Representations of Graphs

Adjacency-list representation

- The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V.
- Adj[u] contains all the vertices v such that there is an edge (u, v) belongs to E. That is, Adj[u] consists of all the vertices adjacent to u in G.
- If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$.
- The amount of memory it requires is $\Theta(V + E)$.
- It has disadvantage of traversing entire list for finding adjacent vertices.
- Hence this representation of graph is used when the graph has less number of vertices.

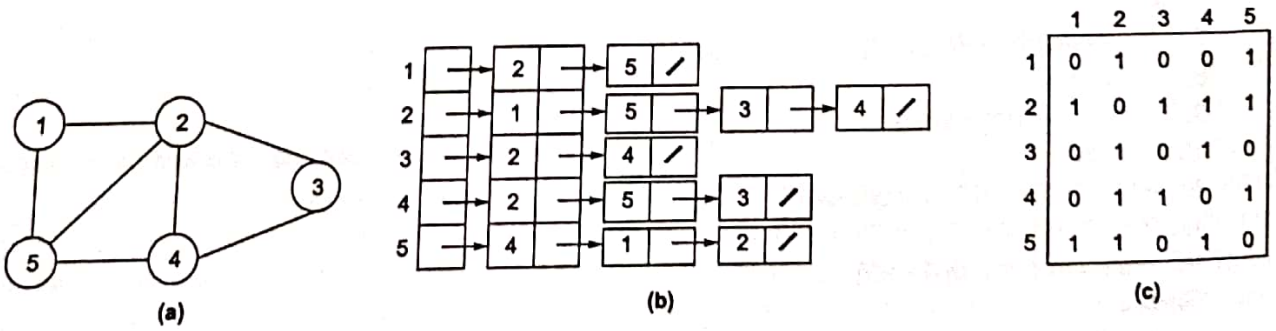


***Figure:** Two representation of an undirected graph (a) An undirected graph G with 5 vertices and 7 edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.*

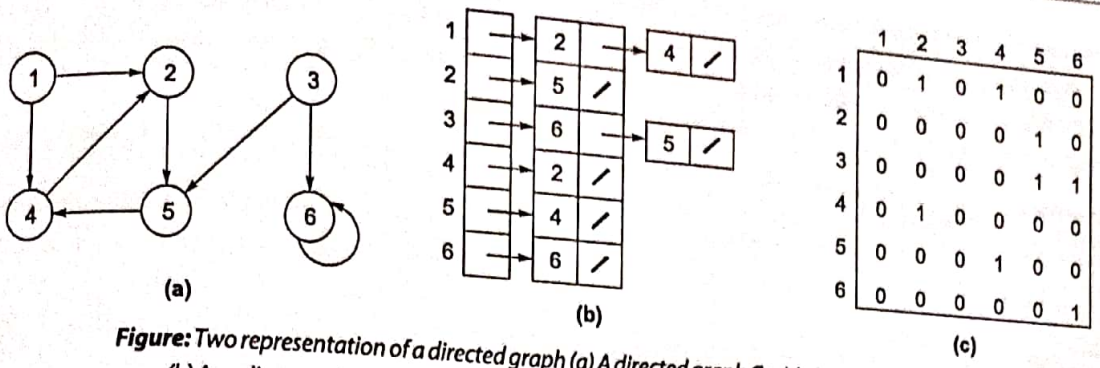


Figure: *Two representation of a directed graph (a) A directed graph G with 6 vertices and 8 edges (b) An adjacency-list representation of G (c) The adjacency-matrix representation of G.*

Adjacency-matrix Representation

- For the adjacency-matrix representations of a graph $G = (V, E)$, we assume that the vertices are numbered 1, 2, 3... $|V|$ in some arbitrary manner.
- The adjacency matrix representations of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \text{ belongs to } E \\ 0 & \text{otherwise} \end{cases}$$

- The adjacency matrix representation requires $\Theta(V^2)$ memory, independent of number of edges in the graph.
- Adjacency-matrix representation is the transpose of itself. For weighted graph we replace 1 or 0 by the edge weights within the matrix.

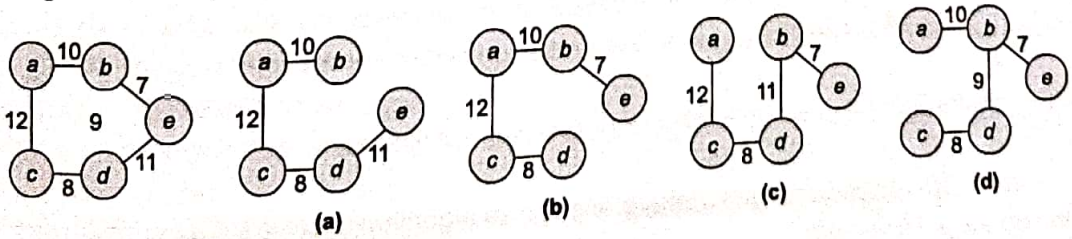
4.6 Minimum Cost Spanning Tree (MCST) Problem

Spanning tree: (Spanning tree): Let G = (V, E) be an undirected connected graph. A **subgraph** T = (V, E') of G is a spanning tree of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.

Minimum cost Spanning tree: Suppose G is a **weighted connected graph**. A **weighted graph** is one in which every edge of G is assigned some positive weight. A graph G is having several spanning tree. A **minimum cost spanning tree** (MCST) of a weighted connected graph G is that spanning tree whose sum of weight of all the edges is minimum, among all the possible spanning tree of G.

In general, a *complete graph* (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if $n = 4$ then total number of spanning tree is 16.

For *example*, consider the following **weighted connected graph** G (as shown in Figure). Out of all possible spanning trees, four spanning trees M_1, M_2, M_3 and M_4 of G are shown in Figure a to Figure (d).



A sum of the weights of the edges in M_1, M_2, M_3 and, M_4 is: 41, 37, 38 and 34 (some other spanning trees M_5, M_6,... are also possible). We are interested to find that spanning tree, out of all possible spanning trees M_1, M_2, M_3, M_4, M_5, ...; whose sum of weights of all its edges are minimum. For a given graph G, M_3 is the MCST, since weight of all its edges is minimum among all possible spanning trees of G.

NOTE

Application of spanning tree:

- Spanning trees are widely used in designing an efficient network. For example, suppose we are asked to design a network in which a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. This problem can be converted into a graph problem in which nodes are telephones (or computers), undirected edges are potential links. The goal is to pick enough of these edges that the nodes are connected. Each link (edge) also has a maintenance cost, reflected in that edge's weight. Now question is "what is the cheapest possible network? An answer to this question is MCST, which connects everyone at a minimum possible cost.
- Another application of MCST is in the **designing of efficient routing algorithm.**

  Suppose we want to find a airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, when we travel more, the more it will cost. So MCST can be applied to optimize airline routes by finding the least costly paths with no cycle.

To find a MCST of a given graph G, one of the following algorithms is used:

1. Prims algorithm
2. Kruskal algorithm

These two algorithms use Greedy approach.

In General for constructing a MCST:

- We will build a set S of edges that is always a subset of some MCST.
- Initially, S has no edges (i.e. empty set).
- At each step, an edge (u, v) is determined such that $S \cup \{(u, v)\}$ is also a subset of a MCST. This edge (u, v) is called a **safe edge.**
- At each step, we can add only ***safe edges*** to set S.
- **Termination:** when all *safe* edges are added to S, we stop. Now S contains edges of spanning tree that is also of minimum cost.

Thus a general MCST algorithm is:

```
GENERIC_MCST(G, w)
{   S ← ϕ
    While A is not a spanning tree
    {   find an edge (u, v) that is safe for S
        S ← A ∪ {(u, v)}
    }
    return S
}
```

A main **difference** between **kruskal's** and **Prim's** algorithm to solve MCST problem is that the order in which the edges are selected.

Kruskal's Algorithm	Prim's Algorithm
➢ Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST.	➢ Prim's algorithm always selects a vertex (say, v) to find MCST.
➢ In Kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps).	➢ In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps).
➢ At intermediate step of algorithm, there can be more than one connected components possible.	➢ At intermediate step of algorithm, there can be only one connected components are possible.

For solving MCST problem using Greedy algorithm, we uses the following data structure and functions, as mentioned earlier.

(*i*) **P: The set of candidates (or given values):** Here P = E, the set of edges of G(V, E).

(*ii*) **S: Set of selected candidates (or input) which is used to give optimal solution.** Here the subset of edges, E′ (i. e. $E' \subseteq E$) is a solution, if the graph T(V, E′) is a spanning tree of G(V, E).

(*iii*) In case of MCST problem, the function **Solution** checks whether a solution is reached or not. The function basically evalution:

(a) All the edges in S form a tree.

(b) The set of vertices of the edges in S equal to V.

(c) The sum of the weights of the edges in S is minimum possible of the edges which satisfy (a) and (b) above. The selection function (say **select**) which chooses the best candidate from P is to be added to the solution set S.

(*iv*) The **select** function chooses the best candidate from P. In case of **Kruskal's algorithm**, it selects an edge, which has least length (from the remaining candidates). But in case of Prim's algorithm, it select a vertex, which is added to the already selected vertices, to minimize the cost of the spanning tree.

(*v*) A function ***feasible*** checks the feasibility of the newly selected candidate (i.e., edge (u, v)). It checks whether a newly selected edge (u, v) form a cycle with the earlier selected edges. If answer is "yes" then the edge (u, v) is rejected, otherwise it is added to the solution set S.

(*vi*) Here the objective function **ObjF** gives the **sum of the edge lengths** in a **Solution.**

4.6.1 Kruskal's Algorithm

Let G(V, E) be a connected, weighted graph. Kruskal's algorithm is find used to a minimum-cost spanning tree (MCST) of a given graph G. It uses *a* **greedy approach** to find MCST, because at each step it adds an edge of least possible weight to the set A. In this algorithm,

- First examine the edges of G in order of increasing weight.
- Then select an edge $(u, v) \in E$ of minimum weight and checks whether its end points belongs to same component or different connected components.
- If u and v belongs to different connected components then we add it to set A, otherwise it is rejected because it can create a cycle.
- The algorithm terminates, when only one connected components remains (i.e. all the vertices of G have been reached).

Following pseudo-code is used to constructing a MCST, using Kruskal's algorithm:

```
KRUSKAL MCST (G, w)
    /* Input: A undirected connected weighted graph G = (V, E).
    /* Output: A minimum cost spanning tree T(V, E') of G
    {
1.      Sort the edges of E in order of increasing weight
2.      A ← ϕ
3.  for (each vertex v ∈ V[G])
4.      do MAKE_SET(v)
5.  for (each edge (u, v) ∈ E, taken in increasing order of weight
        {
6.          if (FIND_SET (u) ≠ FIND_SET (v))
7.          A ← A ∪ {(u, v)}
8.          MERGE(u, v)
        }
9.  return A
    }
```

Kruskal's algorithm works as follows:

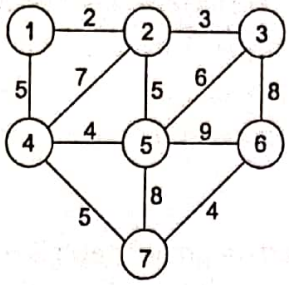
- First, sort the edges of E in order of increasing weight
- We build a set *A* of edges that contains the edges of the MCST. Initially A is empty.
- In line 3-4, the function **MAKE_SET(*v*)**, create a new set {v} for all vertices of G. For a graph with *n* vertices, it create *n* components of disjoint set such as {1}, {2},. . . and so on.
- In line 5-8: An edge $(u, v) \in E$, of minimum weight is added to the set A, if and only if it joins two nodes which belongs to **different** components (to check this use a **FIND_SET()** function, which returns a same integer value, if *u* and *v* belongs to same components (In this case adding (*u*, *v*) to A creates a cycle), otherwise it returns a different integer value)
- If an edge is added to A then the two components containing its end points are merged into a single component.
- The algorithm terminates, when there is just a single component.

Analysis of Kruskal's Algorithm

Let $|V|$ be the number of vertices and $|E|$ be the number of edges

1. Sorting of edges requires $O(|E| \log |E|)$ time.
2. Since, in any graph, minimum number of edges is (V – 1) and maximum number of edges (when graph is complete) is V(V – 1)/2. Hence $V - 1 \le E \le V(V-1)/2$.
   Thus $O(E \log E) = O(E \log (V(V-1)/2)) = O(E \log V)$
3. Initializing n-disjoint set (in line 3-4) using **MAKE_SET** will requires O(V) time.
4. There are at most $2|E|$ **FIND_SET** operations (since there are E edges and each edge has 2 vertices) and (V – 1) **MERGE** operations. Thus we requires $O((2E + (V-1) \log n)$ time.
5. At worst, O(V) time for the remaining operations.
6. For a connected graph, we know that $E \ge (V - 1)$. So the total time for Kruskal's algorithm is $O((2E + (V-1) \log V) = O(|E| \log |V|)$

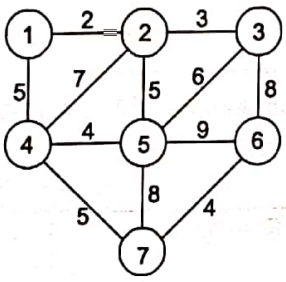
Example - 4.5 Apply Kruskal's algorithm on the following graph to find minimum-cost-spanning tree (MCST).



Solution:

First, we sorts the edges of G = (V, E) in order of increasing weights as:

Edges	(1, 2)	(2, 3)	(4, 5)	(6, 7)	(1, 4)	(2, 5)	(4, 7)	(3, 5)	(2, 4)	(3, 6)	(5, 7)	(5, 6)
weights	2	3	4	4	5	5	5	6	7	8	8	9



The kruskal's Algorithm proceeds as follows:

Step	Edge considered	Connected components	Spanning forests (a)
Initialization		{1} {2} {3} {4} {5} {6} {7} (using line 3-4)	(1) (2) (3) (4) (5) (6) (7)
1.	(1, 2)	{1, 2}, {3}, {4}, {5}, {6}, {7}	(1)–(2) (3) (4) (5) (6) (7)
2.	(2, 3)	{1,2,3}, {4}, {5}, {6}, {7}	(1)–(2)–(3) (4) (5) (6) (7)
3.	(4, 5)	{1,2,3}, {4,5}, {6}, {7}	(1)–(2)–(3) (4)–(5) (6)–(7)
4.	(6, 7)	{1,2,3}, {4,5}, {6,7}	(1)–(2)–(3), (4)–(5) (6)–(7)
5.	(1, 4)	{1,2,3,4,5}, {6,7}	(1)–(2)–(3) | (4)–(5) (7)–(6)
6.	(2, 5)	Edge (2, 5) is rejected, because its end point belongs to same connected component, so create a cycle.	
7.	(4, 7)	{1,2,3,4,5,6,7}	(1)–(2)–(3) | (4)–(5) | (7)–(6)

Total cost of spanning tree, T = 2 + 3 + 5 + 4 + 5 + 4 = 23

Prim's Algorithm

PRIM's algorithm has the property that the edges in the set T which contains the edges of the minimum spanning tree, when algorithm proceed step-by step always form a single tree, i.e. at each step there is only one connected component.

- Begin with one starting vertex (say *v*) of a given graph G(V, E).
- Then, in each iteration, choose a minimum weight edge *(u, v)* connecting a vertex *v* in the set A to the vertices in the set (V – A). That is, we always find an edge *(u, v)* of minimum weight such that $v \in A$ and $u \in V - A$. Then we modify the set A by adding u i.e. $A \leftarrow A \cup \{u\}$
- The process is repeated until $\neq V$, i.e. until all the vertices are not in the set A. Following pseudocode is used to constructing a MCST, using PRIM's algorithm:

```
PRIMS_MCST(G, w)
    /* Input: An undirected connected weighted graph G = (V, E).
    /* Output: A minimum cost spanning tree T(V, E') of G
    {
1.      T ← ϕ        // T contains the edges of the MST
2.      A ← {Any arbitrary member of V}
3.      while (A ≠ V)
        {
4.          find an edge (u, v) of minimum weight such that u ∈ V – A and v ∈ A
5.          A ← A ∪ {(u, v)}
6.          B ← B ∪ {u}
        }
7.  return T
    }
```

Working of the algorithm:

1. Initially the set A of nodes contains a single arbitrary node (i.e. starting vertex) and the set T of edges are empty.
2. At each step PRIM's algorithm search for the shortest possible edge *(u, v)* such that $u \in V - A$ and $v \in A$.
3. In this way the edges in T form at any instance a minimal spanning tree for the nodes in A. Repeat this process until $A \neq V$.

Analysis of PRIM's Algorithm (By using min heap)

For constructing heap: O(V)

By loop executes |V| times and extracting minimum element from min heap it takes log V time.

So while loop takes VlogV.

Total 'E' decrease key operations. Hence takes E log V

$(V \log V + E \log V) = (V + E) \log V \simeq E \log V$

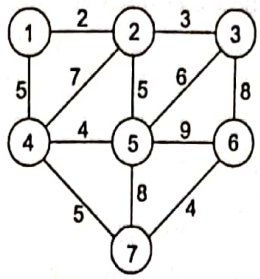
Using fibonacci heap

The time complexity of prim's algorithm is O(E + V log V)

Using binomial heap

The time complexity of prim's algorithm is O(V + E)

Example - 4.6 Apply PRIM's algorithm on the following graph to find minimum-cost-spanning tree (MCST).



Solution:

In PRIM's, First we select an arbitrary member of V as a starting vertex (say 1), then the algorithm proceeds as follows:

Step	Edge Considered (4, v)	Connected Components (set a)	Spanning Forests (Set T)
Initialization		{1}	(1)
1.	(1, 2)	{1, 2}	(1)-(2)
2.	(2, 3)	{1, 2, 3}	(1)-(2)-(3)
3.	(1, 4)	{1, 2, 3, 4}	(1)-(2)-(3) | (4)
4.	(4, 5)	{1, 2, 3, 4, 5}	(1)-(2)-(3) | (4)-(5)
5.	(4, 7)	{1, 2, 3, 4, 5, 7}	(1)-(2)-(3) | (4)-(5) | (7)
6.	(6, 7)	{1, 2, 3, 4, 5, 6, 7}	(1)-(2)-(3) | (4)-(5) | (7)-(6)

Total cost of the minimum spanning tree = 2 + 3 + 5 + 4 + 5 + 4 = 23

4.7 Single Source Shortest Path Problem (SSSPP)

Given: A directed graph G(V, E) with weight edge $w(u, v)$.

- We define the weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ as:

$$W(p) = \sum_{j=1}^{k} w(v_{j-1}, v_j) = \text{sum of edge weights on path } p.$$

- We can define the shortest-path weight from u to v as:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \to v\}; & \text{if there exist a path } u \to v \\ \infty; & \text{otherwise} \end{cases}$$

Single-Source-Shortest Path Problem (SSSPP) Problem

Given a directed graph G(V, E) with weight edge $w(u, v)$. We have to find a shortest path from source vertex $s \in V$ to every other vertex $v \in \{V - s\}$.

SSSP Problem can also be used to solve some other related problems:

- **Single-destination shortest path problem (SDSPP):** Find the transpose graph (i.e. reverse the edge directions) and use **single-source-shortest** path.
- **Single-pair shortest path (i.e. a specific destination, say *v*):** If we solve the SSSPP with source vertex s, we also solved this problem. No algorithm for this problem is known that is asymptotically faster than the SSSP in worst case.
- **All pair shortest path problem (APSPP):** Find a shortest path between every pair of vertices *u* and *v*. One technique is to use SSSP for each vertex, but there are some more efficient algorithms. Such as Floydd-Warshall's algorithm.

To find a SSSP for directed graphs G(V, E), we have two different algorithms:

1. Bellman-Ford algorithm
2. Dijkstra's algorithm

- Bellman-ford algorithm, allow **negative weight edges** in the input graph. This algorithm either finds a shortest path from source vertex $s \in V$ to every other vertex $v \in V$ or detect a negative weight cycles in G, hence **no solution.** If there is no negative weight cycles that is reachable from source vertex s, then we can find a shortest path form source vertex $s \in V$ to every other vertex $v \in V$. If there exist a negative weight cycles in the input graph, then the algorithm can detect it, and hence return no solution.
- **Dijkstra's algorithm** allows only positive **weight edges** in the input graph and finds a shortest path from source vertex $s \in V$ to every other vertex $v \in V$.

To understand the basic concept of negative-weight cycle, consider the following 2 cases:

Case1: ***Shortest-path cannot contain a cycle; it is just a simple path (i.e. no repeated vertex):***

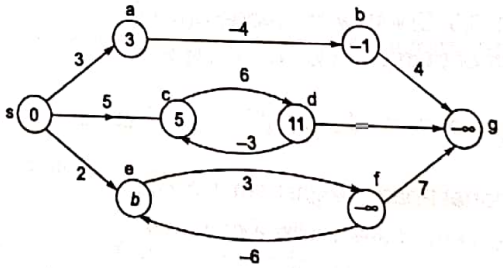
If some path from s to v contains a negative cost cycle, then there does not exist a shortest path. Otherwise there exists a shortest path (i.e. a simple path) from $u \rightarrow v$.

Case 2: Graph containing a negative-weight cycle:

- There is no problem, if it is not reachable from the source vertex s.
- If it is reachable from source vertex s, then just keep going around it and producing a path weight of $-\infty$. If there is a negative-weight cycle on some path from *s* to *v*, we define $\delta(s, v) = -\infty$ for all *v* on the cycle.

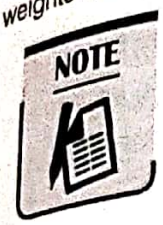
For example, consider a graph with negative weight cycle:

If there is a negative weight cycle on some path from s to *v*, we define $\delta(s, v) = -\infty$.



There are infinitely many paths from *s* to *c*: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$ and so on. Because the cycle $\langle c, d, c\rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from *s* to *c* is $\langle s, c\rangle$, with weight $\delta(s, c) = 5$. Similarly, the shortest path from *s* to *d* is $\langle s, c, d\rangle$ with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from *s* to *e*: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, and so on. Since the cycle $\langle e, f, e\rangle$ has

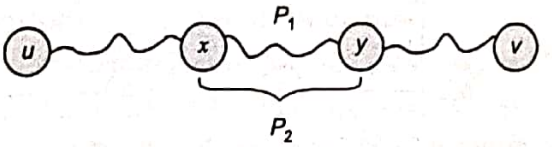
weight $3 + (-6) = -3 < 0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle (e, f, e) arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) = -\infty$. Because g is reachable from f, we can also find paths with arbitrarily large negative weights from s to g, and $\delta(s, g) = -\infty$.



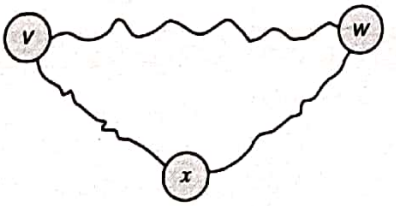
Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in the road-map example. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Shortest Path: Properties

1. **Optimal substructure property:** Any sub-path of a shortest path is also a shortest path. Let P_1 is any sub-path (say $x \to y$) of a shortest $s \to v$, and P_2 is $axyx \to y$ path; then the cost of $P_1 \le$ cost of P_2; otherwise $s \to v$ is not a shortest path.



2. **Triangle inequality:** Let $d(v, w)$ be the length of the shortest path from v to w, then $d(v, w) \le d(v, x) + d(x, w)$



3. **Shortest path does not contains a cycle:**
   - Negative weight cycles are not allowed when it is reachable from source vertex s, since in this case there is no shortest path.
   - If Positive weight cycles are there then by removing the cycle, we can get a shorter path.

Generic Algorithm for solving single-source-shortest path (SSSP) problem:

Given a directed weighted graph G(V, E), algorithm always maintain the following two fields for each vertex $v \in V$.

1. $d[v] \cong \delta(s, v)$ = the length of the shortest path from starting vertex s to v, (initially $d[v] = \delta$), and the value of $d[v]$ reduces as the algorithm progress. Thus we call $d[v]$, a **shortest path estimate.**
2. Pred[v], which is a predecessor of v on a shortest path from s. The value of Pred[v] is either a another vertex or NIL

Initialization:

All the shortest-paths algorithms start with initializing $d[v]$ and Pred[v] by using the following procedure INITIALIZE SINGLE SOURCE.

```
INITIALIZE_SINGLE_SOURCE (V, s)
1. for each v ∈ V
2.     do d[v] ← ∞
3.        pred[v] ← NIL
4. d[s] ← 0
```

After this initialization procedure, $d[v] = 0$ for start vertex s, and $d[v] = \infty$ for $v \in V - \{s\}$ and $pred[v] = NIL$ for all $v \in V$.

Relaxing an Edge (u, v)

The SSSP algorithms are based on the technique known as **edge relaxation**. The process of relaxing an edge (u, v) consists of testing whether we can improve (or reduce) the shortest path to v found so far (i.e. $d[v]$) by going through u and taking (u, v) and, if so, update $d[v]$ and $pred[v]$. This is accomplished by the following procedure:

RELAX (u, v, w)
1. if $(d[v] > d[u] + w(u, v))$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $pred[v] \leftarrow u$

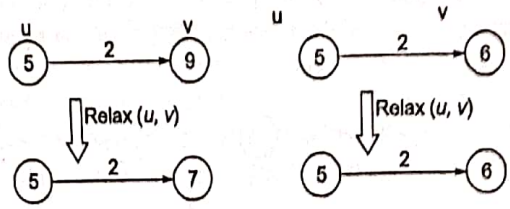


Figure (a): $d[v] > d[u] + w(u, v)$ i.e. since, $9 > 5 + 2$, hence $d[v] = 7$
Figure (b): No change in $d[v]$ $\quad d[v] < d[u] + w(u, v)$

NOTE

- For all the SSSP algorithm, we always start by calling INITIALIZE_SINGLE_SOURCE (V, s) and then relax edges.
- In **Dijkstra's algorithm** and the other shortest-path algorithm for directed acyclic graph, each edge is relaxed exactly once. In a **Bellman-Ford** algorithm, each edge is relaxed several times

4.7.1 Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with **non-negative edge** weights i.e. we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. That is, all vertices $v \in S$, we have $d[v] = \delta(s, v)$ the algorithm repeatedly selects the vertex $u \in V\text{-}S$ with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in $V - s$ keyed by their d values. Graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)
1. INITIALIZE-SINGLE-SOURCE (G, s)
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. while $Q \neq \emptyset$
5. do $u \leftarrow$ EXTRACT-MIN (Q)
6. $S \leftarrow S \cup (u)$
7. for each vertex $v \in Adj[u]$
8. do RELAX (u, v, w)

Because Dijkstra's algorithm always choose the "lightest" or "closest" vertex in V-S to insert into set S, we say that it uses a greedy strategy.

Dijkstra's algorithm bears some similarly to both breadth-first search and Prim's algorithm for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth-first distances.

Dijkstra's algorithm is like prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

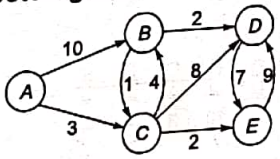
Analysis of Dijkstra's algorithm

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as function of $|E|$ and $|V|$ using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q.

In this case, the running time is $O(|V|^2 + |E|) = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently, storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires $O((|E| + |V|)$ time (which is dominated by $O(|E| \log |V|)$ assuming every vertex is connected, and the Fibonacci heap improves this to $O(|E| + |V| \log |V|)$.

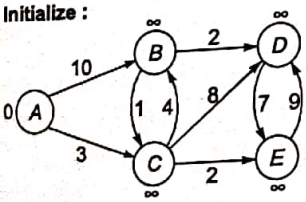
Example - 4.7 **Apply Dijkstra's algorithm to find shortest path from source vertex A to each of the other vertices of the following directed graph.**



Solution:

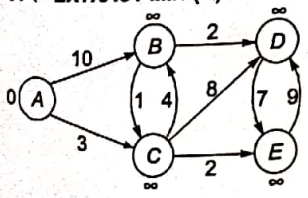
Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights from the source have already been determined. The algorithm repeatedly selects the vertex $u \in$ V-S with the minimum shortest-path estimate, inserts u into S and relaxes all edges leaving u. We maintain a min-priority queue Q that contains all the vertices in V – s keyed by their d values.

Initialize :



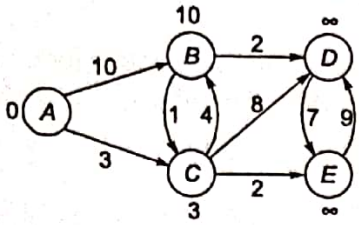
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

'A' ← EXTRACT-MIN (Q) :



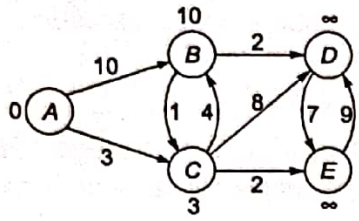
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

Relax all edges Leaving 'A' S: {A}



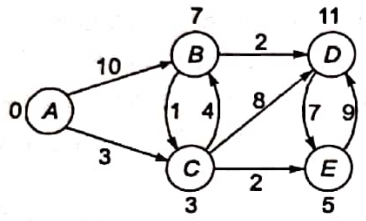
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

'C' ← EXTRACT-MIN (Q) :



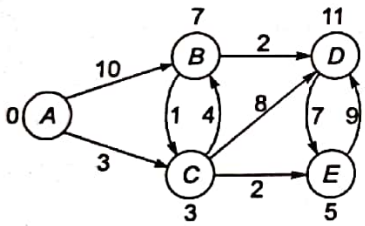
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

Relax all edges Leaving 'A' S: {A, C}



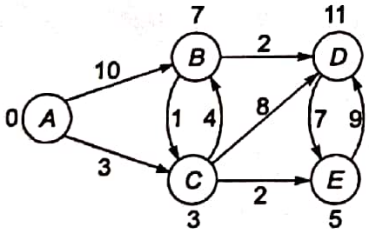
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

'E' ← EXTRACT-MIN (Q) :



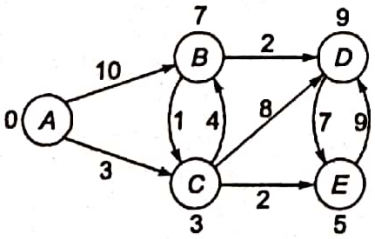
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

Relax all edges Leaving 'A' S: {A, C, E}



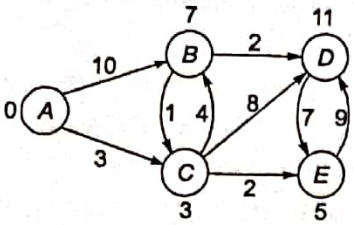
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

'B' ← EXTRACT-MIN (Q) :



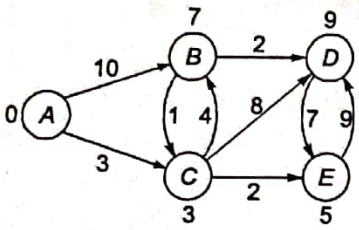
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

Relax all edges Leaving 'A' S: {A, C, E, B}



Q:	A	B	C	D	E
	[0]	∞	∞	∞	∞
		10	[3]	∞	∞
		7		11	[5]
		7		11	

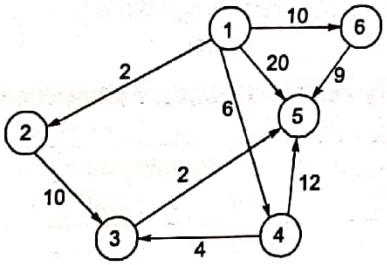
'D' ← EXTRACT-MIN (Q) :



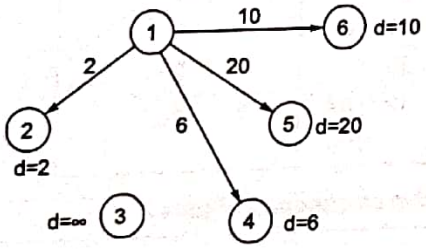
Q:	A	B	C	D	E
	[0]	∞	∞	∞	∞
		10	[3]	∞	∞
		7		11	[5]
		[7]		11	
				9	

S: {A, C, E, B, D}

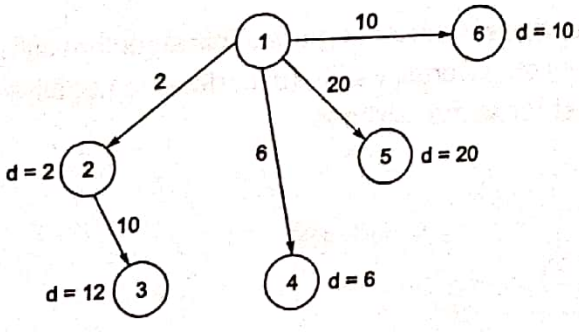
Example - 4.8 Apply dijkstra's algorithm on the following digraph (1 is starting vertex) source



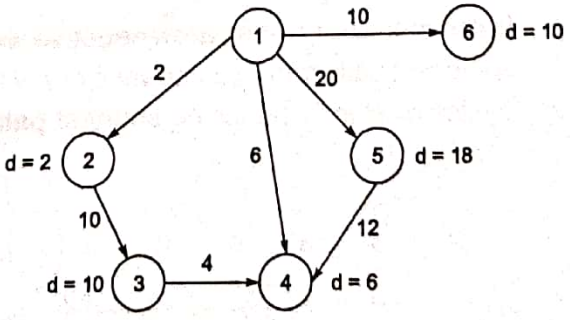
Solution:



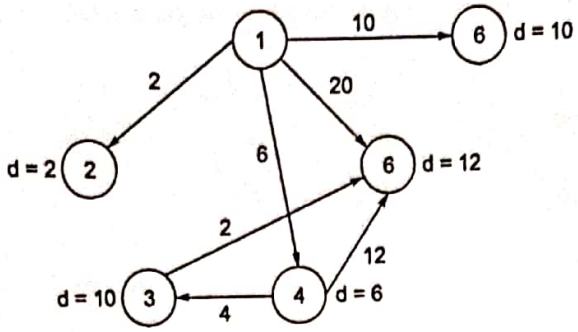
Initially S = {1}



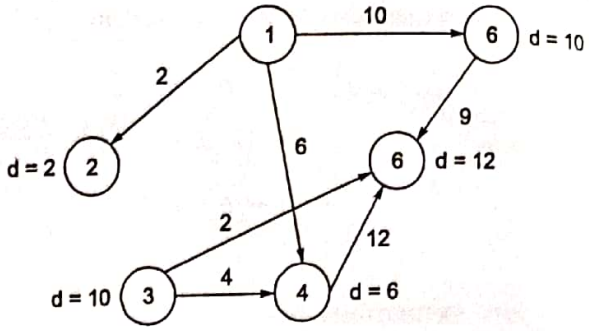
(a) S = {1, 2}



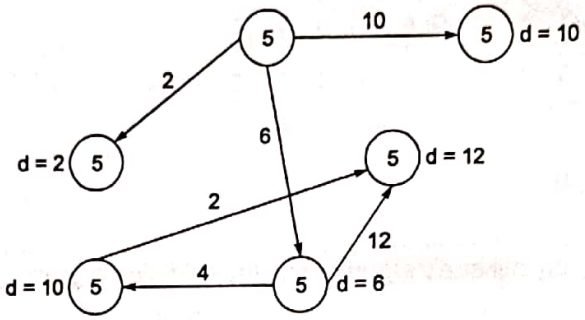
(b) S = {1, 2, 4}



(c) S = {1, 2, 4, 3}



(d) S = {1, 2, 4, 3, 6}



(e)

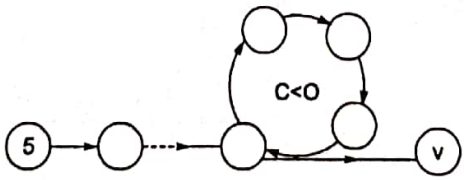
Figure: Stages of Dijkstra's Algorithm

Iterations	S	Q (Priority Queue) d[1] d[2] d[3] d[4] d[5] d[6]	EXTRACT_MIN(Q)
Initial	{ }	0 ∞ ∞ ∞ ∞ ∞	1
1.	{1}	[0] 2 ∞ 6 20 10	2
2.	{1, 2}	[2] 12 6 20 10	4
3.	{1, 2, 4}	10 [6] 18 10	3
4.	{1, 2, 4, 3}	[10] 18 10	6
5.	{1, 2, 4, 3, 6}	12 [10]	5
	{1, 2, 4, 3, 6, 5}	[12]	

Table: Computation of Dijkstra's algorithm on digraph of Figure 2

4.7.2 Bellman-Ford Algorithm

- Bellman-ford algorithm, allow **negative weight edges** in the input graph. This algorithm either finds a shortest path from a source vertex $s \in V$ to every other vertex $v \in V - \{s\}$ or detect a negative weight cycles exist in G, hence **no shortest path exist for some vertices.**



C: Cycle cost

- Thus Given a weighted, directed graph with G = (V, E) with weight function w: E → R, the Bellman-ford algorithm returns a **Boolean value** (either **TRUE** or **FALSE**) indicating whether or not there is a

negative weight cycle is reachable from the source vertex. The algorithm returns a Boolean TRUE if the given graph G contains no negative weight cycle that are reachable from source vertex *s*, otherwise it returns Boolean FALSE.

- The algorithm uses a technique of **relaxation** and progressively decreases an estimate $d[v]$ on the weight of a shortest path from the source vertex s to each vertex $v \in V$ until it achieves the actual shortest path. We also maintain a predecessor value pred[v] for all $v \in V$.

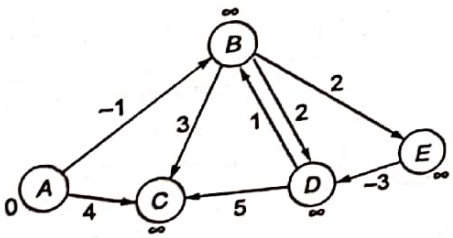
BELLMAN_FORD (G, w, s)

1. INITIALIZE SIGLE SOURCE(G, s)
2. for $i \leftarrow 1$ to $|V| - 1$
3. do for each edge $(u, v) \in E[G]$
4. do RELAX (u, v, w)
5. for each edge $(u, v) \in E[G]$
6. do if $(d[v] > d[u] + w(u, v)$
7. then return FALSE // we detect a negative weight cycle exist
8. return TRUE

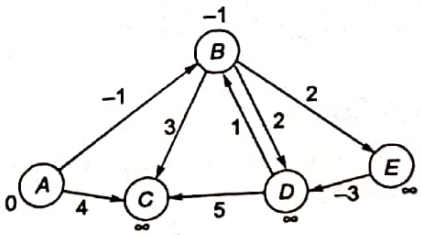
Analysis of Bellman-Ford algorithm

1. Line 1 for initializing
   $d[v]$'s, Pred[v]'s and setting $v[s] = 0$ takes O(V) time.
2. For loop at line 2 executed ($|V| - 1$) times which Relaxes all the E edges, so line 2-4 requires ($|V| - 1$).
   O(E) = O(VE).
3. For loop at line 5 checks negative weight cycle for all the E edges, which requires O(E) time. Thus the run time of Bellman ford algorithm is O(V + E + VE) = O(VE).

Order of edge: (B, E), (D, B), (B, D), (A, C), (D, C), (B, C), (E, D)

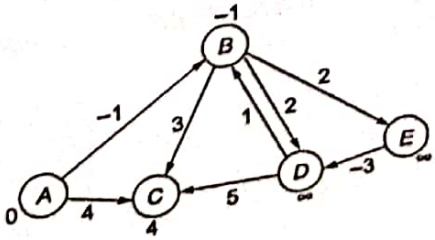


A	B	C	D	E
0	∞	∞	∞	∞

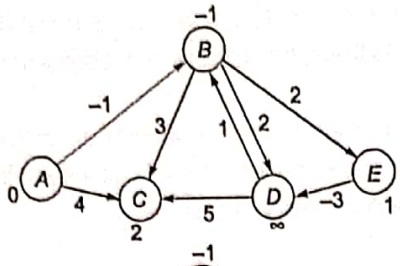


A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞

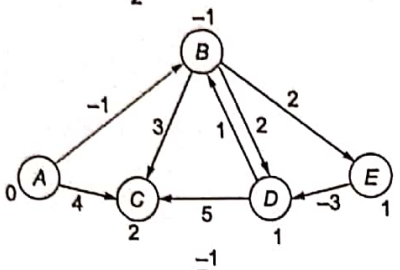
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞



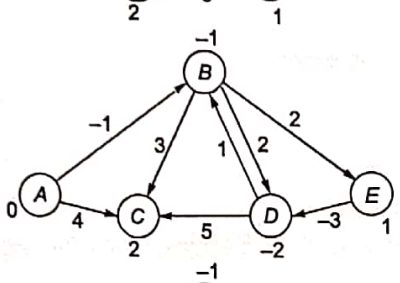
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞



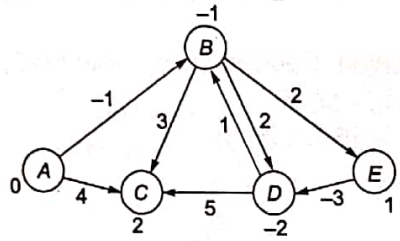
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	1	1
0	−1	2	1	1
0	−1	2	−2	1



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

4.8 Huffman Coding

- Consider a scenario where have a 100,000 character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

Character	a	b	c	d	e	f
Frequency in thousands	45	13	12	16	9	5

- A binary code encodes each character as a binary string or codeword. We need to find a binary code that encodes the file using as few bits as possible, i.e., compresses it as much as possible.
- In a **fixed-length code** each codeword is given the same length. In a **variable-length code** codeword may have different lengths.
- Here are examples of fixed and variable length codes for our problem. A fixed length code must have at least 3 bits per codeword because there are 6 characters.

Character	a	b	c	d	e	f
Frequency in thousands	45	13	12	16	9	5
fixed-length code	000	001	010	011	100	101
variable-length code	0	101	100	111	1101	1100

- The fixed length code requires 300,000 bits to store the codeword, where as variable length code requires 224,000 bits.

  $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4)1000 = 224,000$ bits

Encoding

- Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message.
- Just replace the characters by the code words.

Example: $\Sigma = \{a, b, c, d\}$

If the code is $C_1 = \{a = 00, b = 01, c = 10, d = 11\}$ then **bad** is encoded as 010011

If the code is $C_2 = \{a = 0, b = 110, c = 10, d = 111\}$ then **bad** is encoded as 1101111

Decoding

$C_1 = \{a = 00, b = 01, c = 10, d = 11\}$

$C_2 = \{a = 0, b = 110, c = 10, d = 111\}$

$C_3 = \{a = 1, b = 110, c = 10, d = 111\}$

- Given an encoded message, decoding is the process of reversing the code into the original message.
- For example by using C_1, **010011** is uniquely decodable to bad. By using C_2 **1101111** is uniquely decodable to bad. But, relative to C_3, **1101111** is not uniquely decipherable since it could have encoded either **bad** or **acad**.
- Hence the unique decipherability property is required in order for a code to be efficient.

Prefix-Codes

- Fixed-length codes are always uniquely decipherable, but they do not always give the best compression so we prefer to use variable length codes.
- **Prefix Code:** A code is called a prefix (free) code if no codeword is a prefix of another one.
- *Example:* $\{a = 0, b = 110, c = 10, d = 111\}$ is a prefix code.

NOTE: Every message encoded by a prefix free code is uniquely decipherable. Since no code- word is a prefix of any other we can always find the first codeword in a message, peel it off, and continue decoding.

Optimum Source Coding Problem

- **The problem:** Given an alphabet $P = \{p_1, \ldots, p_n\}$ with frequency distribution $f(p_i)$ find a binary prefix code C for P that minimizes the number of bits needed to encode a message of $\sum_{a=1}^{n} f(a)$ characters, where $c(p_i)$ is the codeword for encoding p_i, and $L(c(p_i))$ is the length of the codeword $c(p_i)$.

$$B(C) = \sum_{p=1}^{n} f(p_i) L(c(p_i))$$

- Huffman developed greedy algorithm for solving this problem and produce a minimum cost (optimum) prefix code. The code that it produces is called a Huffman code.

  Step 1: Pick two letters *a, b* from alphabet P with the smallest frequencies and create a sub tree that has these two characters as leaves, label the root of this sub tree as c.

  Step 2: Set frequency $f(c) = f(a) + f(b)$. Remove *a, b* and add *c* creating new alphabet $P' = P \cup \{c\} - \{a, b\}$. Note that $|P'| = |P| - 1$.
- Repeat this procedure, called merge, with new alphabet P′ until an alphabet with only one symbol is left.
- The resulting tree is the Huffman code.

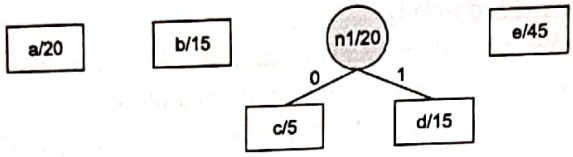
Procedure of finding prefix codes using Binary Trees

1. Correspondence between leaves and characters.
2. Label of leaf is frequency of character.
3. Left edge is labelled 0; right edge is labelled 1
4. Path from root to leaf is codeword associated with character.

Example-4.9 Let A = {*a*/20, *b*/15, *c*/5, *d*/15, *e*/45} be the alphabet and its frequency distribution. What is the optimum (minimum-cost) prefix code for this distribution?

Solution:

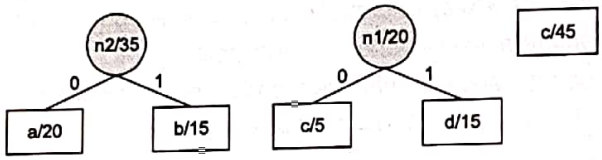
In the first step Huffman coding merges *c* and *d*.



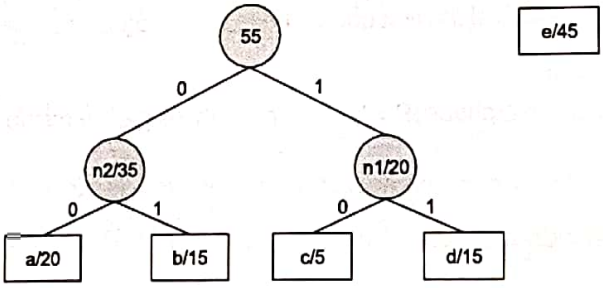
Alphabet is now A_1 = {*a*/20, *b*/15, n_1/20, *e*/45}

Alphabet is now A_1 = {*a*/20, *b*/15, n_1/20, *e*/45}

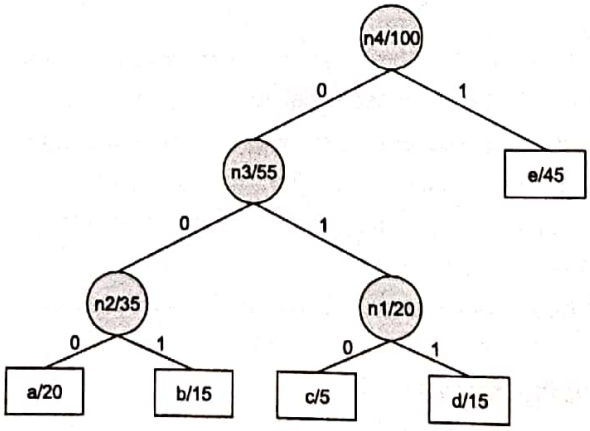
Algorithm merges *a* and *b* (could also have merged n_1 and *b*)



Alphabet is A_2 = {*n*2/35, *n*1/35, *n*1/20, *e*/45}



Algorithm merges *n*1 and *n*2



New alphabet is $A_3 = \{n3/55, e/45\}$
Current alphabet is $A_3 = \{n3/55, e/45\}$
Algorithm merges e and $n3$ and finishes.
Huffman code is $a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 1$.
This is the optimum (minimum-cost) prefix code for this distribution.

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a): a \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with labels (frequencies) as keys.

```
Huffman (A)
{
    n = |A|;
    Q = A;              /* the future leaves
    for i = 1 to n – 1
    {
        z = new node;
        Left [z] = Extract-Min (Q);
        Right[z] = Extract-Min (Q);
        f[z] = f[left [z]] + f[right [z]]; Insert (Q, z);
    }
    return Extract-Min (Q)    /* root of the tree
}
```

Running time is $O(n \log n)$, as each priority queue operation takes time $O(\log n)$.

NOTE: The tree for any optimal prefix code must be "full", meaning that every internal node has exactly two children. Huffman's algorithm produces an optimum prefix code tree.

Summary

- Greedy algorithms are typically used to solve an **optimization problem.**
- An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or minimized w. r. t. some constraints or conditions.
- Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it **solution set**, S where S**c** C) that satisfies the given constraints or conditions. Any subset S **c C**, which satisfies the given constraints, is called a **feasible** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called a **optimal solution**.
- Greedy algorithm always makes the choice that looks best at the moment. That is, it makes a **locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.**
- Greedy algorithm does not always yield an optimal solution; but for many problems they do.
- The (fractional) Knapsack problem is to fill a knapsack or bag (up to its maximum capacity M) with the given n items, which maximizes the total profit earned.

- Let G = (V, E) be an undirected connected graph. A **subgraph** T = (V, E') of G is a **spanning tree** of G if and only if T is a tree (i.e. no cycle exist in T) and contains **all the vertices** of G.
- A complete graph (each vertex in G is connected to every other vertices) with n vertices has total n^{n-2} spanning tree. For example, if $n = 5$ then total number of spanning tree is 125.
- A **minimum cost spanning tree** (MCST) of a weighted connected graph G is that spanning tree whose sum of length (or weight) of all its edges is minimum, among all the possible spanning tree of G.
- There are two algorithm to find a MCST of a given directed graph G, namely Kruskal's algorithm and Prim's algorithm.
- The basic difference between Kruskal's and Prim's algorithm is that in kruskal's algorithm it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). Thus At intermediate step of algorithm, there are may be more than one connected components of trees are possible. But in case of Prim's algorithm it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). Thus at intermediate step of algorithm, there will be only one connected components are possible.
- Kruskal's algorithm runs in $O(|E| \log |V|)$ time and Prim's algorithm runs in time (n^2), where n is the number of nodes in the graph.
- **Single-source-shortest path problem (SSSPP) problem** is to find a shortest path from source vertex $s \in V$ to every other vertex $v \in V$-s in a given graph G = (V, E)
- To find a SSSP for directed graphs G(V, E), we have two different algorithms: **Bellman-Ford algorithm** and **Dijkstra's algorithm**
- **Bellman-ford algorithm,** allow **negative weight edges** also in the input graph where as **Dijkstra's algorithm** allows only **positive weight edges** in the input graph.
- **Bellman-ford algorithm** runs in time $O(|V||E|)$ **whereas Dijkstra's algorithm runs in time** $O(n^2)$.

Student's Assignments

Q.1 The essence of greedy algorithm is the _____ policy.

(a) optimal (b) maximization
(c) selection (d) none of these

Q.2 The running time of KRUSKAL's algorithm, where $|E|$ is the number of edges and $|V|$ is the number of nodes in a graph using min-heap tree

(a) $O(|E|^2)$ (b) $O(V^2)$
(c) $O(|E| \log |V|)$ (d) $O(|V| \log |V|)$

Q.3 The running time of PRIM's algorithm, where $|E|$ is the number of edges and $|V|$ is the number of nodes in a graph using adjacency-list and binary heap tree

(a) $O(|E|)$ (b) $O(|E| \log |E|)$
(c) $O(|E| \log |V|)$ (d) $O(|V| \log |V|)$

Q.4 The optimal solution to the Knapsack instance $n = 3$, M = 20, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(W_1, W_2, W_3) = (18, 15, 10)$ is:

(a) 28.2 (b) 31.0
(c) 31.5 (d) 41.5

Q.5 The solution set $X = \{X_1, X_2, X_3\}$ for the problem given in Q.4 is

(a) $\left(\frac{1}{15}, \frac{2}{15}, 0\right)$ (b) $\left(0, \frac{2}{3}, 1\right)$

(c) $\left(0, 1, \frac{1}{2}\right)$ (d) None of these

Q.6 Total number of spanning tree in a complete graph with 5 nodes are

(a) 5^2 (b) 5^3

(c) 10 (d) 100

Q.7 Let, (i, j, c) denotes edge i to j with c cost. Consider the following edges and cost in order of increasing length:

(b, e, 3), (a, c, 4), (e, f, 4), (b, c, 5), (f, g, 5), (a, b, 6), (c, d, 6), (e, f, 6), (b, d, 7), (d, e, 7), (d, f, 7), (c, f, 7)

Which of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

(a) (b, e), (e, f), (a, c), (b, c), (f, g), (c, d)

(b) (b, e), (e, f), (a, c), (f, g), (b, c), (c, d)

(c) (b, e), (a, c), (e, f), (b, c), (f, g), (c, d)

(d) (b, e), (e, f), (b, c), (a, c), (f, g), (c, d)

Q.8 Consider a question given in Q.7. Applying Kruskal's algorithm to find total cost of a Minimum spanning tree.

Q.9 State whether the following statements are false.

(a) If e is a minimum edge weight in a connected weighted graph, it must be among the edges of at least one minimum spanning tree of the graph.

(b) If e is a minimum edge weight in a connected weighted graph , it must be among the edges of each one minimum spanning tree of the graph.

(c) If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.

(d) If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

(e) If edge weights of a connected weighted graph are not all distinct, the minimum cost of each one minimum spanning tree is same.

Q.10 Find the optimal solution to the knapsack instance $n = 5$, M = 10, $(P_1, P_2, ..., P_5) = (12, 32, 40, 30, 50)$ $(W_1, W_2, ..., W_5) = (4, 8, 2, 6, 1)$.

Q.11 Let S = {a, b, c, d, e, f, g} be a collection of objects with Profit-Weight values as follows: a:(12, 4), b:(10, 6), c:(8, 5), d:(11, 7), e:(14, 3), f:(7, 1) and g:(9, 6). What is the optimal solution to the fractional Knapsack problem for S, assuming we have a Knapsack that can hold objects with total weight 18? What is the complexity of this method.

Q.12 This of the following algorithm allows negative edge weight in a graph to find shortest path?

(a) Dijkstra's algorithm

(b) Bellman-Ford algorithm

(c) Kruskal algo

(d) Prim's Algo

Q.13 The running time of Bellman-Ford algorithm is

(a) $O(|E|^2)$ (b) $O(|V|^2)$

(c) $O(|E| \log |V|)$ (d) $O(|E|\,|V|)$

Q.14 Consider a weighted undirected graph with positive edge weights and let su be an edge in the graph. It is known that the shortest path from source vertex s to u has weight 60 and the shortest path from source vertex s to v has weight 75. Which statement is always true?

(a) weight $(u, v) = 15$ (b) weight $(u, v) \geq 15$

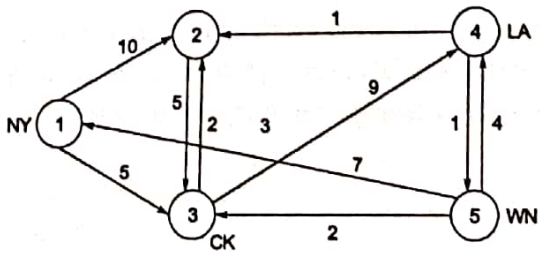
(c) weight $(u, v) \geq 15$ (d) weight $(u, v) > 15$

Q.15 Which data structure is used to maintained the distance of each node in a Dijkstras's algorithm.

(a) Stack (b) Queue

(c) Priority Queue (d) Tree

Q.16 Find the minimum distance of each station from New York (NY) using Dijkstra's algorithm. Show all the steps.



Q.17 Assume MADEEASY has 80,000 students. All students are joined in A, B, C and D batches. MADEEASY want to compress the student record for batch codes A, B, C and D. Suppose A is appeared 40,000 times, B is appeared 15,000 times, C is appeared 20,000 times and D is appeared 5000 times. Using minimum average bits per letter find the prefix code for A, B, C and D respectively?

(a) 0, 10, 111, 001
(b) 1, 000, 01, 001
(c) 1, 001, 000, 00
(d) 0, 111, 10, 001

Q.18 Consider a file has the following frequencies of 6 characters

	g	*a*	*t*	*e*	*c*	*s*
Freqency	5	9	12	13	16	45

Find the number of bits required to encode a file?

(a) 124
(b) 224
(c) 324
(d) 424

Answers Key:

1. (a)	**2.** (c)	**3.** (c)	**4.** (c)	**5.** (c)
6. (b)	**7.** (c)	**8.** (27)	**9.** (a)	**10.** (158)
11. (49.4)	**12.** (b)	**13.** (d)	**14.** (c)	**15.** (c)
16. (15)	**17.** (c)	**18.** (b)		

CHAPTER 05

Graph Based Algorithms

5.1 Introduction

When we characterize the running time of a graph algorithm on a given graph $G = (V, E)$, we usually measure the size of the input in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the graph. That is, we describe the size of the input with two parameters, not just one.

5.2 Graph Searching

Searching a graph: Systematically follow the edges of a graph to visit the vertices of the graph. Used to discover the structure of a graph.

Standard graph-searching algorithms:

- Breadth-first Search (BFS)
- Depth-first Search (DFS)

5.2.1 Depth First Search

Main Idea: DFS is a systematic method of visiting the vertices of a graph. Its general step requires that if we are currently visiting vertex 'u', then we next visit a vertex adjacent to 'u' which has not yet been visited. If no such vertex exists then we return to the vertex visited just before u and the search is repeated until every vertex in that component of the graph has been visited.

DFS uses a strategy that searches "deeper" in the graph whenever possible, unlike BFS which discovers all vertices at distance 'k' from the source before discovering any vertices at distance $k + 1$.

The predecessor subgraph produced by DFS may be composed of several trees, because the search may be repeated from several sources. This predecessor subgraph forms a depth-first forest 'E' composed of several depth-first trees and the edges in 'E' are called tree edges. On the other hand the predecessor subgraph of BFS forms a tree.

DFS Algorithm

Input: Graph $G(V, E)$

Output: Predecessor subgraph (depth-first forest) of G.

```
DFS(G)
    // initialization
    for each vertex u ∈ V[G] do
            color[u] ← WHITE
            π[u] ← Null
    time ← 0
    for each vertex u ∈ V[G] do
            if color [u] = WHITE
                    then DFS-Visit (u)

DFS-Visit(u)
    color [u] ← GREY
// white vertex u has just been discovered
    d[u] ← time
    time ← time +1
    for each v Adj[u] do          // explore edge (u, v)
    if color [v] = WHITE
            then [v]u ← u
                    DFS-Visit(v)
color[u] ← BLACK                  // u is finished
f[u] ← time
time ← time +1
```

In this algorithm we assign two timestamps to each vertex: discovery and finishing time. Three colours are used for colouring the vertices.

- **White:** Vertex is not discovered yet.
- **Grey:** Discovered but not finished.
- **Black:** If no white descendant of v exists, then v becomes black.

Algorithm initializes each vertex to "white" to indicate that they are not discovered yet. It also sets each vertex's parent to null. The procedure begins by selecting one vertex u from the graph, setting its color to "grey" to indicate that the vertex is now discovered (but not finished) and assigning to it discovery time 0.

For each vertex v that belongs to the set Adj[u], and is still marked as "white", DFS-Visit is called recursively, assigning to each vertex the appropriate discovery time d[v] (the time variable is incremented at each step)

If no white descendant of v exists, then v becomes black and is assigned the appropriate finishing time, and the algorithm returns to the exploration of v's ancestor (v).

If all of u's descendants are black, u becomes black and if there are no other white vertices in the graph, the algorithm reaches a finishing state, otherwise a new "source" vertex is selected, from the remaining white vertices, and the procedure continues as before.

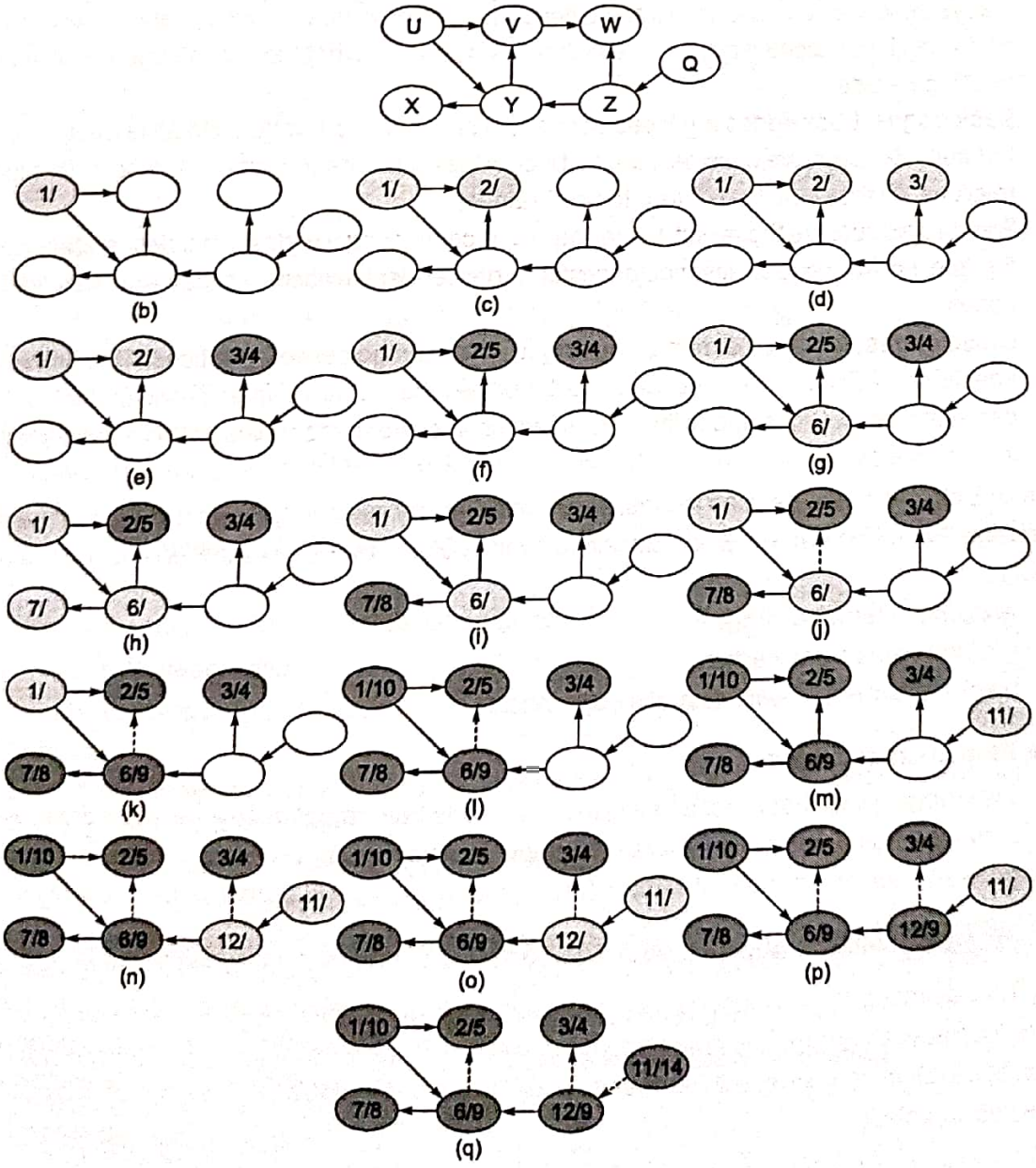
Important Properties

- The structure of the resulting depth-first trees, maps directly the structure of the recursive calls of DFS-Visit, as $u = (v)$ if and only if DFS-Visit was called during a search of u's adjacency list.
- As a result, the predecessor subgraph constructed with DFS forms a forest of trees.
- **Parenthesis structure:** If the discovery time of a vertex v is represented with a left parenthesis "(v", and finishing time is represented with a right parenthesis "v)", then the history of discoveries and finishes forms an expression in which the parentheses are properly nested.

Time Complexity

The initialization part of DFS has time complexity $\Theta(n)$, as every vertex must be visited once so as to mark it as "white". The main (recursive) part of the algorithm has time complexity $\Theta(m)$, as every edge must be crossed (twice) during the examination of the adjacent vertices of every vertex. In total, the algorithm's time complexity is $\Theta(m + n)$.

Example of Depth first search:



- In any Depth-First search of a (directed or undirected) graph G = (V, E), for any two vertices u and v, exactly one of the following three conditions holds:
  (a) The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint.
  (b) The interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the depth-first tree
  (c) The interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the depth-first tree
- Vertex v is a proper descendant of vertex u in the depth first forest for a graph G if and only if $d[u] < d[v] < f[v] < f[u]$

Edge Classification

During a depth-first search, a vertex can be classified as one of the following types:

1. **Tree edges:** Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v). A tree edge always describes a relation between a node and one of its direct descendants. This indicates that $d[u] < d[v]$, (u's discovery time is less than v's discovery time), so a tree edge points from a "low" to a "high" node.
2. **Back edges:** Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges. Back edges describe descendant-to-ancestor relations, as they lead from "high" to "low" nodes.
3. **Forward edges:** are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree. Forward edges describe ancestor-to-descendant relations, as they lead from "low" to "high" nodes.
4. **Cross edges:** Are all other remaining edges. They can go between vertices in the same depth-first tree as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. Cross edges link nodes with no ancestor-descendant relation and point from "high" to "low" nodes.

The DFS algorithm can be used to classify graph edges by assigning colors to them in a manner similar to vertex coloring. Each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored:

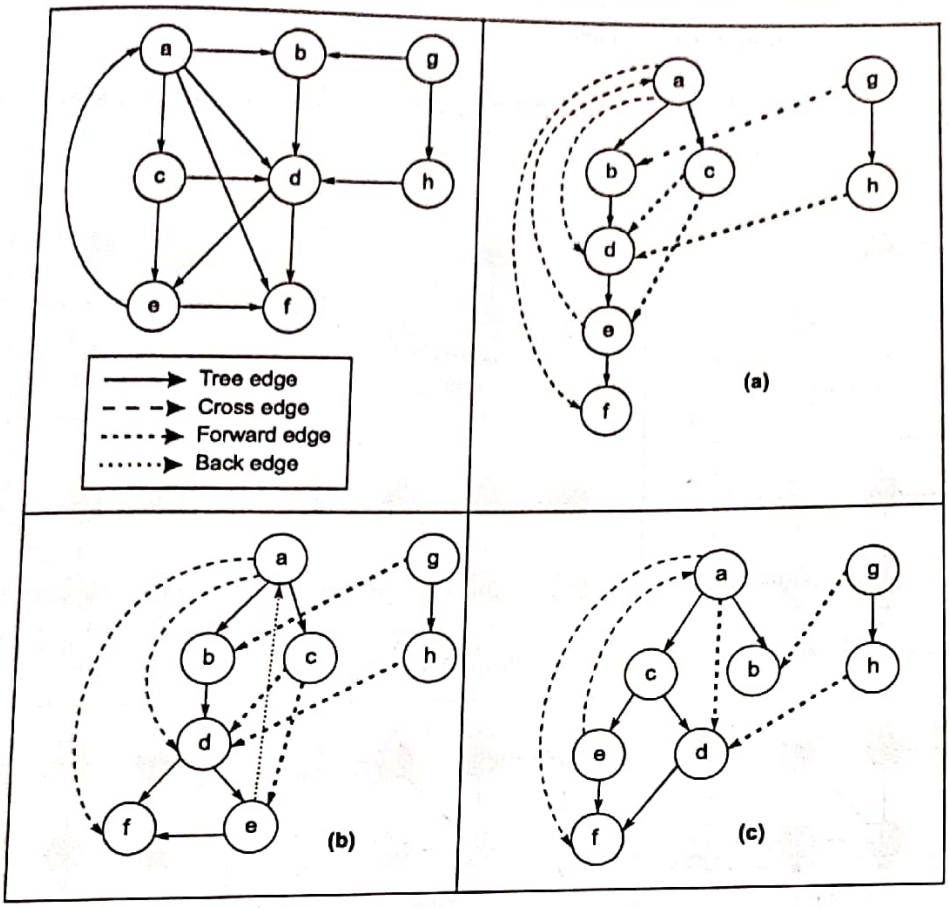
- white indicates a tree edge
- gray indicates a back edge
- black indicates a forward or cross edge

Back Edge Vs and Cross Edges

For every edge (u, v), that is not a tree edge, that leads from "high" to "low" we must determine if v is an ancestor of u: starting from u me must traverse the depth-first tree to visit u's ancestors. If v is found then (u, v) is a back edge, else-if we reach the root without having found v – (u, v) is a cross edge.

NOTE: A directed graph G is acyclic (DAG) if and only if a depth-first search of G yields no back edges.

DAG is a directed graph where no path starts and ends at the same vertex (i.e. it has no cycles). It is also called an oriented acyclic graph. The note above is true because suppose that there is a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.



5.2.2 Breadth-first Search

- Input: Graph G = (V, E), either directed or undirected, and **source vertex** $s \in V$.
- Output:
  (a) $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  (b) $p[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
  u is v's predecessor.
  (c) Builds breadth-first tree with root s that contains all reachable vertices.

Definitions:

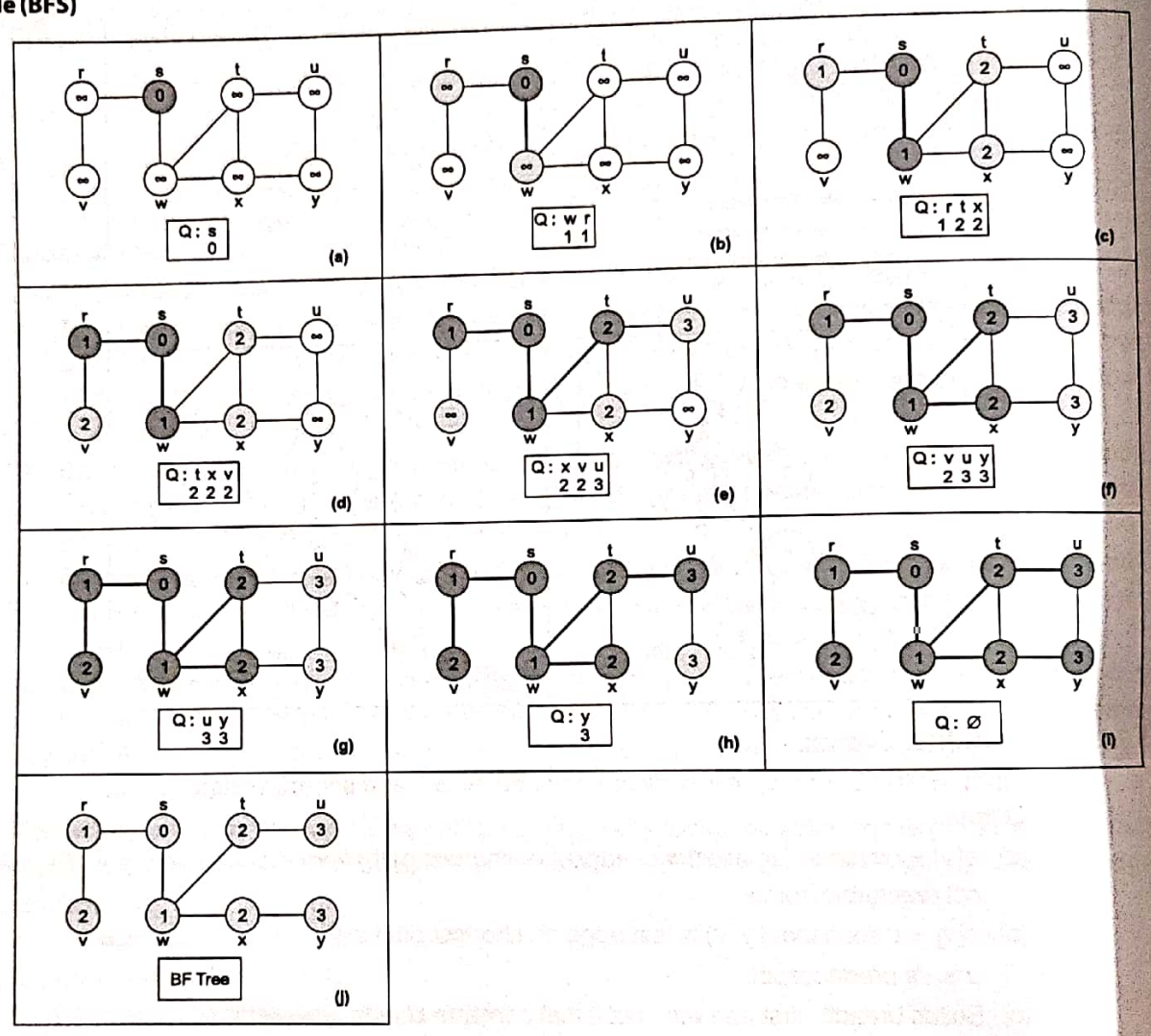
- Path between vertices u and v: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u = v_1$ and $v = v_k$, and $(v_i, v_{i+1}) \in E$, for all $1 \le i \le k-1$.
- Length of the path: Number of edges in the path.
- Path is simple if no vertex is repeated.
- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  (a) A vertex is "discovered" the first time it is encountered during the search.
  (b) A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  (a) White – Undiscovered.

(b) Gray – Discovered but not finished.

(c) Black – Finished.

- Colors are required only to reason about the algorithm. Can be implemented without colors.

Example (BFS)



BSF (G, s)

1. for each vertex u in V[G] – {s}
2. do color[u] ← white
3. $d[u] \leftarrow \alpha$
4. $\pi[u] \leftarrow$ nil
5. color[s] ← gray
6. $d[s] \leftarrow 0$
7. $\pi[s] \leftarrow$ nil
8. $Q \neq \Phi$
9. enqueue (Q, s)
10. while Q = Φ

```
11.     do u ← dequeue (Q)
12.         for each v in adj[u]
13.             do in color[v] = white
14.                 then color[v] ← gray
15.                     d[v] ← d[u] +1
16.                     π[v] ← u
17.                     enqueue (Q, v)
18          color[u] ← black
```

Q : A queue of discovered vertices
color[v] : Color of v
d[v] : Distance from s to v
π[u] : Predecessor of v
White : Undiscovered
Gray : Discovered
Black: Finished

Analysis of BFS

- Initialization takes O(V).
- Traversal Loop
  (a) After initialization, each vertex is enqueued and dequeued at most once, and each operation takes O(1). So, total time for queuing is O(V).
  (b) The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is Θ(E).
- Summing up over all vertices ⇒ Total running time of BFS is O(V + E), linear in the size of the adjacency list representation of graph.

Breadth-first Tree

- For a graph G = (V, E) with sources, the predecessor subgraph of G is $G_p = (V_p, E_p)$ where
  (a) $V_p = \{v \in V : p[v] \neq \text{NIL}\} \cup \{s\}$
  (b) $E_p = \{(p[v], v) \in E : v \in V_p - \{s\}\}$
- The predecessor subgraph G_p is a breadth-first tree if:
  (a) V_p consists of the vertices reachable from s and
  (b) for all $v \in V_p$, there is a unique simple path from s to v in G_p that is also a shortest path from s to v in G.
- The edges in E_p are called tree edges. $|E_p| = |V_p| - 1$.

5.3 Directed Acyclic Graphs (DAG)

Definition

DAG is a directed graph where no path starts and ends at the same vertex (i.e. it has no cycles). It is also called an oriented acyclic graph.

Source: A source is a node that has no incoming edges.

Sink: A sink is a node that has no outgoing edges.

Properties

A DAG has at least one source and one sink. If it has more that one source, we can create a DAG with a single source, by adding a new vertex, which only has outgoing edges to the sources, thus becoming the new single (super-) source. The same method can be applied in order to create a single (super-) sink, which only has incoming edges from the initials sinks.

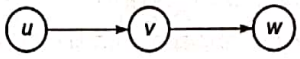
5.4 Topological Sorting

Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. Only one task can be performed at a time and each task must be completed before the next task begins. Such a relationship can be represented as a directed graph and topological sorting can be used to schedule tasks under precedence constraints. We are going to determine if an order exists such that this set of tasks can be completed under the given constraints. Such an order is called a *topological sort* of graph *G*, where *G* is the graph containing all tasks, the vertices are tasks and the edges are constraints. This kind of ordering exists if and only if the graph does not contain any cycle (that is, no self-contradict constraints). These precedence constraints form a directed acyclic graph, and any topological sort (also known as a *linear extension)* defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

5.4.1 Definition

The topological sort of a DAG G(V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering. Mathematically, this ordering is described by a function t that maps each vertex to a number

$t: V \rightarrow \{1, 2, \ldots, n\} : t(v) = i$ such that $t(u) < t(v) < t(w)$ for the vertices u, v, w of the following diagram:



5.4.2 Algorithm

The following algorithms perform topological sorting of a DAG:

Topological sorting using DFS

1. Perform DFS to compute finishing times f[*v*] for each vertex v
2. As each vertex is finished, insert it to the front of a linked list
3. Return the linked list of vertices

Topological sorting using bucket sorting

This algorithm computes a topological sorting of a DAG beginning from a source and using bucket sorting to arrange the nodes of *G* by their in-degree (# incoming edges).

The bucket structure is formed by an array of lists of nodes. Each node also maintains links to the vertices (nodes) to which its outgoing edges lead. The lists are sorted so that all vertices with *n* incoming edges lay on the list at the n^{th} position of the table, as shown in figure:

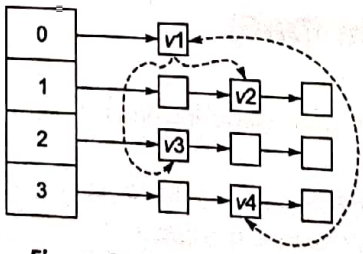


Figure: *Bucket Structure Example*

The main idea is that at each step we exclude a node which does not depend on any other. That maps to removing an entry *v*1 from the 0^{th} index of the bucket structure (which contains the sources of G) and so reducing by one the in-degree of the entry's adjacent nodes and reallocating them in the bucket. This means that node *v*2 is moved to the list at the 0^{th} index, and thus becoming a source. *v*3 is moved to the list at the 1^{st} index, and *v*4

is moved to the list at the 2nd index. After we subtract the source from the graph, new sources may appear, because the edges of the source are excluded from the graph, or there is more than one source in the graph. Thus the algorithm subtracts a source at every step and inserts it to a list, until no sources (and consequently no nodes) exist. The resulting list represents the topological sorting of the graph.

5.4.3 Time Complexity

DFS takes $\Theta(m+n)$ time and the insertion of each of the vertices to the linked list takes 0(1) time. Thus topological sorting using DFS takes time $\Theta(m+n)$.

Points to be Remembered

Three important facts about topological sorting are:

1. Only directed acyclic graphs can have linear extensions, since any directed cycle is an inherent contradiction to a linear order of tasks. This means that it is impossible to determine a proper schedule for a set of tasks if all of the tasks depend on some "previous" task of the same set in a cyclic manner.

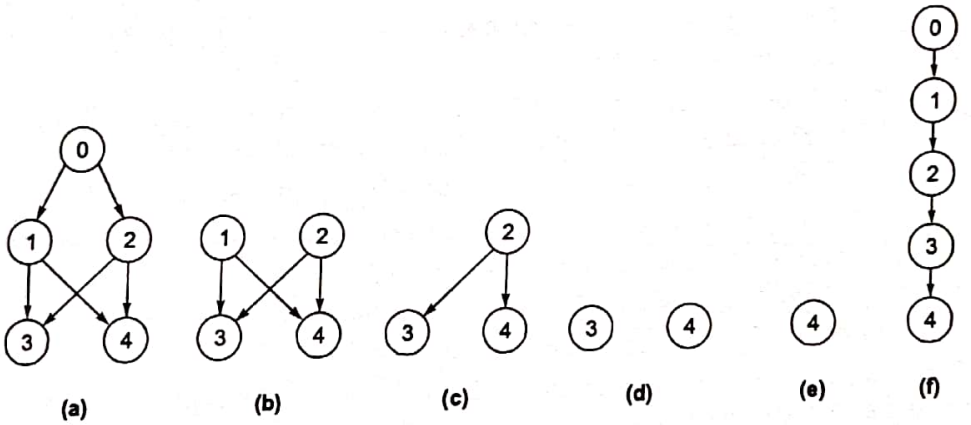


Figure: *Execution sequence of the topological sorting algorithm*

   (a) Select source 0.
   (b) Source 0 excluded. Resulting sources 1 and 2. Select source 1.
   (c) Source 1 excluded. No new sources. Select source 2.
   (d) Source 2 excluded. Resulting sources 3 and 4. Select source 3.
   (e) Source 3 excluded. No new sources. Select source 4.
   (f) Topological Sorting of the graph.
2. Every DAG can be topologically sorted, so there must always be at least one schedule for any reasonable precedence constraints among jobs.
3. DAGs typically allow many such schedules, especially when there are few constraints. Consider n jobs without any constraints. Any of the $n!$ permutations of the jobs constitutes a valid linear extension. That is if there are no dependencies among tasks so that we can perform them in any order, then any selection is "legal".

Example: Figure shows the possible topological sorting results for three different DAGs.

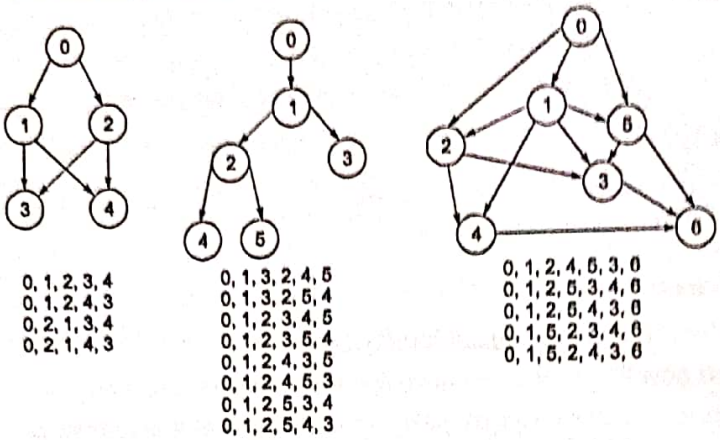


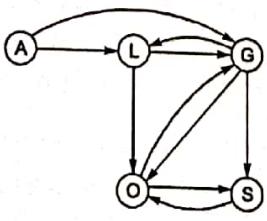
Figure: *Possible topological sorting results for three DAGS*

Summary

- DFS uses a strategy that searches "deeper" in the graph whenever possible, unlike BFS which discovers all vertices at distance k from the source before discovering any vertices at distance $k + 1$.
- In any Depth-First search of a (directed or undirected) graph G = (V, E), for any two vertices u and v, exactly one of the following three conditions holds:
  - (a) The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint.
  - (b) The interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the depth-first tree
  - (c) The interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the depth-first tree
- Vertex v is a proper descendant of vertex u in the depth first forest for a graph G if and only if $d[u] < d[v] < f[v] < f[u]$
- DAG is a directed graph where no path starts and ends at the same vertex (i.e. it has no cycles). It is also called an oriented acyclic graph.
- A DAG has at least one source and one sink.
- The topological sort of a DAG G(V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.

Student's Assignments

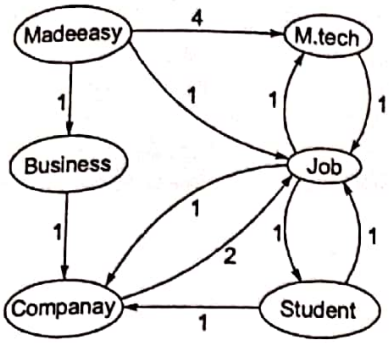
Q.1 Consider the following graph.



Find which of the following is not a BFS traversal of above graph with 'A' as starting vertex

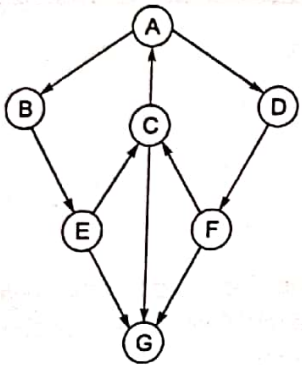
(a) ALGOS (b) AGLSO
(c) AGLOS (d) ALGSO

Q.2 Find the number of paths from "Madeeasy" to "Student" with the cost "6" using the following graph. [vertices and edges may be repeated in the paths].



(a) 2 (b) 4
(c) 6 (d) None of these

Q.3 Consider the following graph G.



Perform a DFS on G starting at vertex A and selection of adjacent vertex in DFS decided by the lexicographical order. In graph G, B and D are adjacent to A. First it selects B, because B comes first in lexicographical order. Find number of cross edges when the given DFS is performed on G?

Q.4 Which of the following is false?

(*i*) An edge (u, v) is a forward edge if and only if $d[u] \le d[v] \le f[v] \le f[u]$

(*ii*) An edge (u, v) is a cross edge if and only if $d[u] < f[u] < d[v] < f[v]$

(*iii*) An edge (u, v) is a back edge if end only if; $d[v] < d[u] < f[u] < f[v]$

(a) (*i*), (*iii*) (b) (*ii*), (*iii*)
(c) (*iii*) only (d) All of these

Q.5 The problem of finding the set of vertices reachable from a given vertex in a graph can be solved in time:

(a) $O(|V|)^2$ (b) $C(|V| + |E|)$
(c) $O(|V||E|)$ (d) intractable

Q.6 Assume, the BFS algorithm is written such that it colors the vertices of a graph in the following way. Initially, all vertices are white. As a vertex is "visited", it is colored gray and enquired. A vertex is dequeued when its adjacent vertices are "visited" and then it is colored block

(a) to test for the presence of a cycle in a graph
(b) to topologically sort a graph
(c) to obtain the transpose of a graph
(d) to obtain strongly connected components of a graph

Q.7 Consider a simple connected graph G with n vertices and n edges ($n > 2$). Then, which of the following statements are true?

(a) G has no cycle.
(b) G has atleast one cycle
(c) The graph obtained by removing any edge from G is not connected.
(d) The graph obtained any removing any two edges from G is not connected.

Q.8 Queue can be used to implement

(a) recursion (b) Quick sort
(c) DFS (d) None of these

Answer Key:

1. (d) **2.** (a) **3.** (2) **4.** (c) **5.** (b)
6. (a) **7.** (b, d) **8.** (d)

CHAPTER 06

Dynamic Programing

6.1 Introduction

We have seen algorithms that involve recursion. In some situations, these algorithms solve fairly difficult problems efficiently, but in other cases they are inefficient because they recalculate certain function values many times. The idea of dynamic programming is to avoid making redundant method calls. Instead, one should store the answers to all necessary method calls in memory and simply look these up as necessary. Using the dynamic programming approach we are going to discuss the following Algorithms:

1. Fibonacci series
2. All pair shortest paths problem
3. Matrix chain multiplication
4. LCS problem
5. 0/1 Knapsack
6. Multi stage graph

6.2 Fibonacci Numbers

The Fibonacci sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... The next number is found by adding up the two numbers before it. Similarly, the 3 is found by adding the two numbers before it (1 + 2). It's the Fibonacci sequence, described by the recursive formula:

$$F_0 := 0; \quad F_1 := 1;$$

$$F_n = F_{n-1} + F_{n-2}, \text{ for all } n > 2.$$

Trivial algorithm for computing F_n

```
int fib (int n)
{   if (n == 0) return 0
    else if (n == 1) return 1;
    else
    return fib (n – 1) + fib (n – 2)
}
```

Figure shows how the recursion unravels.

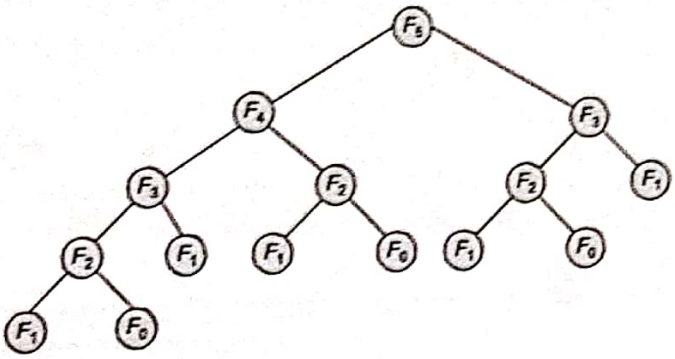


Figure: *fib (5) recursive tree*

Runtime Analysis

Let $T(n)$ be time complexity of above algorithm, then

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\geq 2T(n-2) + c \\ &\geq \ldots \\ &\geq 2^k T(n-2 \cdot k) + c(2^{k-1} + 2^{k-2} + \ldots + 2 + 1) \\ &= \Omega(c2^{n/2}), \end{aligned}$$

where c is the time needed to add n-bit numbers.

Hence $T(n) \geq \Omega(n2^{n/2})$

This is exponential function of time taken by fib(n). Hence use used improve the time complexity.

Problem with Recursive Algorithm

Computes $F(n-2)$ twice, $F(n-3)$ three times, etc., each time from scratch. Hence we need to avoid redundant work.

Improved Fibonacci Algorithm

Never recompute a subproblem $F(k)$, $k \leq n$, if has been computed before. This technique of remembering previously computed values is called **memorization.**

Recursive Formulation of Algorithm:

```
Memory = { }
d_fib(n)
{
    if (n in Memory) return Memory[n]
    else if (n==0) return 0
        else if (n== 1) return 1
        else f = d_fib (n – 1) + d_fib (n – 2)
        Memory [n] = f
        return f
}
```

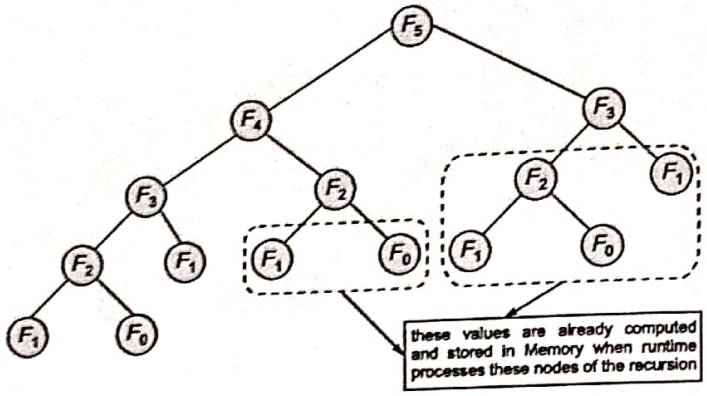


Figure: *Unraveling the Recursion of the Clever Fibonacci Algorithm*

Let $T(n)$ be time complexity of above algorithm, then

$$T(n) = T(n-1) + c = O(cn) = O(n)$$

NOTE: There is an O(log n) time algorithm using divide and conquer technic.

Using Dynamic Programming (DP)

* DP ≈ recursion + memorization (i.e. reuse)
* DP ≈ controlled brute force"

DP results in an efficient algorithm, if the following conditions hold:

- The optimal solution can be produced by combining optimal solutions of subproblems;
- The optimal solution of each subproblem can be produced by combining optimal solutions of sub-subproblems, etc;
- The total number of subproblems arising recursively is polynomial.

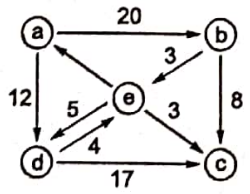
Implementation Trick:

- Remember **(memorize)** previously solved "subproblems"; e.g., in Fibonacci, we memorized the solutions to the subproblems $F_0, F_1, \ldots, F_{n-1}$, while unraveling the recursion.
- If we encounter a subproblem that has already been solved, reuse solution.

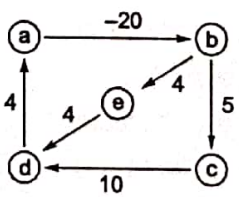
Runtime ≈ # of subproblems · time/subproblem

6.3 All-Pairs Shortest Paths Problem

Given a weighted digraph G = (V, E) with a weight function w: E → R, where R is the set of real numbers, determine the length of the shortest path (i.e., distance) between all pairs of vertices in G. Here we assume that there are no cycle with zero or negative cost.



Without negative cost cycle



With negative cost cycle

Case-I: All positive edged weight.

Which run Dijkstra's algorithm '|V|' times at each vertex. By using binary heap the time complexity of dijktra's algorithm is O(V + E) log V.

For dense graph $E = V^2$

$\therefore$ Time complexity of finding all pair shortest path using dijkstra's algorithm is:

V * Dijktra's time complexity

V*(V + E) log V

$\Rightarrow$ $O(V^3 (\log V))$

Case-II: Assuming negative edge weights. When the graph have negative weights or negative cycles Dijktra's fail to given correct solution. Therefore we use Bellman Ford algorithm while deadling with negative edge weighted graph.

Approach is same as above Run Bellman ford algorithm $|V|$ times at each vertex:

$|V|$ Bellman Ford time complexity

$\Rightarrow$ $|V| \cdot (VE)$

$\Rightarrow$ $O(V^4)$ in worst case.

Case-III: Graph is unweighted graph.

When ever the given graph G = (V, E) is unweighted we use breadth first Traversal to find shortest path. To obtain all pair shortest paths we need to run BFS at each vertex.

The BFT (graph traversal) takes O(V + E) time.

$\therefore$ $V \cdot (V + E) \Rightarrow O(V^3)$ for dense graphs.

In all the above cases instead if we use floyd Warshall's algorithm, it will not take more than $O(V^3)$ time.

6.3.1 Floyd Warshall's Algorithm

We assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

$$W_{ij} = \begin{cases} 0 & \text{If } i = j \\ w(i, j) & \text{If } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{If } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

Output Format: An $n \times n$ distance $D = [d_{ij}]$ where d_{ij} is the distance from vertex i to j.

Step 1: The Floyd-Warshall Decomposition

Definition: The vertices $v_2, v_3, \ldots, v_{l-1}$ are called the intermediate vertices of the path $p = \langle v_1, v_2, \ldots, v_l \rangle$.

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that all intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.
  $d_{ij}^{(0)}$ is set to be w_{ij}, i.e., these are direct cost with no intermediate vertex. Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.
- **Claim:** $d_{ij}^{(n)}$ is the distance from i to j. So our aim is to compute $D^{(n)}$.
- **Subproblems:** Compute $D^{(k)}$ for $k = 0, 1, \ldots, n$.

NOTE: $d_{ij}^{(0)}$ means no intermediate node between i and j. D^0 means no intermedaite node between any two pair of vertices.

Step 2: Structure of shortest paths

Observation 1: A shortest path does not contain the same vertex twice.

Proof: A path containing the same vertex twice contains a cycle. Removing cycle gives a shorter path.

Observation 2: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$, there are two possibilities:

1. k is not a vertex on the path. The shortest such path has length $d_{ij}^{(k-1)}$.
2. k is a vertex on the path. The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Consider a shortest path from i to j containing the vertex k. It consists of a subpath from i to k and a subpath from k to j. Each subpath can only contain intermediate vertices in $\{1, \ldots, k-1\}$ and must be as short as possible, namely they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

Hence the path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Combining the two cases we get

$$d_{kj}^{(k-1)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$$

Step 3: The Bottom-up Computation

- Bottom: $D^{(0)} = [w_{ij}]$ the weight matrix.
- Compute $D^{(k)}$ from $D^{(k-1)}$ using k as intermediate node.

$$d_{kj}^{(k-1)} = \min\left\{d_{ij}^{(k-1)}, d_{ij}^{(k-1)} + d_{kj}^{(k-1)}\right\} \quad \text{for } k = 1, \ldots, n.$$

The Floyd-Warshall Algorithm: Version 1

```
Floyd-Warshall (w, n)
{   for i=1 to n do                          // initialize
    for j = 1 to n do
    { D^0 [i, j] = w[i, j];
        pred [i, j] = nil;
    }
for k = 1 to n do                            // dynamic programming
for i = 1 to n do
for j = 1 to n do
if (d^(k-1) [i, k] + d^(k-1) [k, j] < d^(k-1) [i, j])
    { d^(k) [i, j] = d^(k-1) [i, k] + d^(k-1) [k, j];
        pred [i, j] = k;
    }
else d^(k) [i, j] = d^(k-1) [i, j];
return d^(n) [1...n, 1...n];
}
```

Run Time Analysis of Floyd-Warshall Algorithm

- The algorithm's running time is clearly $\Theta(n^3)$.
- The predecessor pointer pred $[i, j]$ can be used to extract the final path.
- **Problem:** The algorithm uses $\Theta(n^3)$ space. It is possible to reduce this down to $\Theta(n^2)$ space by keeping only one matrix instead of n.

The Floyd-Warshall Algorithm: Version 2

```
Floyd-Warshall (w, n)
{   for i = 1 to n do
    for j = 1 to n do
    { d[i, j] = w[i, j];
```

```
            pred [i, j] = nil;
        }
    for k = 1 to n do
    for i = 1 to n do
    for j = 1 to n do
    if (d[i, k] + d[k, j] < d[i, j])
        { d[i, j] = d[i, k] + d[k, j];
            pred [i, j] = k;
        }
    return d[1...n, 1...n];
    }
```

Extracting the Shortest Paths

The predecessor pointers pred $[i, j]$ can be used to extract the final path. The idea is as follows.

Whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set pre $[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then pred $[i, j]$.

To find the shortest path from i to j, we consult pred $[i, j]$ = nill.

If it is nil, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to pred $[i, j]$ and the shortest path from pred $[i, j]$ to j.

The Algorithm for Extracting the Shortest Paths

```
Path (i, j)
{
    if (pred [i, j] == nil)                    // single edge
        output (i, j);
    else                                       // compute the two parts of the path
    {
        Path (i, pred [i, j]);
        Path (pred [i, j], j);
    }
}
```

Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2 . . 3	Path (2, 3)	pred [2, 3] = 4	
2 . . 4 . . 3	Path (2, 4)	pred [2, 4] = 5	
2 . . 5 . . 4 . . 3	Path (2, 5)	pred [2, 5] = nil	Output (2, 5)
25 . . 4 . . 3	Path (5, 4)	pred [5, 4] = nil	Output (5, 4)
254 . . 3	Path (4, 3)	pred [4, 3] = 6	
254 . . 6 . . 3	Path (4, 6)	pred [4, 6] = nil	Output (4, 6)
2546 . . 3	Path (6, 3)	pred [6, 3] = nil	Output (6, 3)
225463			

6.4 Matrix Chain Multiplication

Matrix: A n × m matrix A = [$a[i, j]$] is a two-dimensional array

$$A = \begin{bmatrix} a[1,1] & a[1,2] & \cdots & a[1,m-1] & a[1,m] \\ a[2,1] & a[2,2] & \cdots & a[2,m-2] & a[2,m] \\ \vdots & \vdots & & \vdots & \vdots \\ a[n,1] & a[n,2] & \cdots & a[n,m-1] & a[n,m] \end{bmatrix}$$, which has n rows and m columns.

Example: The following is a 4 × 5 matrix:

$$\begin{bmatrix} 12 & 8 & 9 & 7 & 6 \\ 7 & 6 & 89 & 56 & 2 \\ 5 & 5 & 6 & 9 & 10 \\ 8 & 6 & 0 & -8 & -1 \end{bmatrix}$$

Recalling Matrix Multiplication

The product C = AB of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by

$$c[i, j] = \sum_{k=1}^{q} a[i, k]\, b[k, j]$$

for $1 \le i \le p$ and $1 \le j \le r$.

Example: If

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix}$$

then $$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}$$

Remarks on Matrix Multiplication

- If AB is defined, BA may not be defined.
- It is possible that AB ≠ BA.
- Multiplication is recursively defined by

  $A_1 A_2 A_3 \ldots A_{S-1} A_S = A_1 (A_2 (A_3 \ldots (A_{S-1} A_S)))$
- Matrix multiplication is **associative**, e.g.,

  $A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3)$,

  So parenthesization does not change result.

Direct Matrix Multiplication AB

Given a $p \times q$ matrix A and a $q \times r$ matrix B, the direct way of multiplying C = AB is to compute each

$$c[i, j] = \sum_{k=1}^{q} a[i, k]\, b[k, j]$$

for $1 \le i \le p$ and $1 \le j \le r$.

Complexity of Direct Matrix Multiplication

Note that C has pr entries and each entry takes $\Theta(q)$ time to compute so the total procedure takes $\Theta(pqr)$ time.

Given a $p \times q$ matrix A, a $q \times r$ matrix B and a $r \times s$ matrix C, then ABC can be computed in two ways (AB)C and A(BC):

The number of multiplications needed are:

$$\text{mult [(AB)C]} = pqr + prs,$$
$$\text{mult [A(BC)]} = qrs + pqs.$$

When $p = 5$, $q = 4$, $r = 6$ and $s = 2$, then

$$\text{mult [(AB)C]} = 180$$
$$\text{mult [A(BC)]} = 88$$

As we see there is a big difference in the number of multiplication required.

Implication: The multiplication "sequence" (parenthesization) is important. So as to minimize the number of multiplication.

NOTE: Parenthesization of matrices does not change the result of multiplication but may alter the number of scalar multiplications required to obtain the resultant matrix.

The Chain Matrix Multiplication Problem

Given dimensions $p_0, p_1, \ldots, p_n$, corresponding to matrix sequence $A_1, A_2, \ldots, A_n$ where A_i has dimension $p_{i-1} \times p_i$, determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1 A_2 \ldots A_n$. That is, determine how to parenthesize the multiplications.

$$\begin{aligned} A_1 A_2 A_3 A_4 &= (A_1 A_2)(A_3 A_4) \\ &= A_1(A_2 (A_3 A_4) = A_1 ((A_2 A_3) A_4) \\ &= ((A_1 A_2) A_3)(A_4) = (A_1(A_2 A_3))(A_4) \end{aligned}$$

Exhaustive search: $\Omega(4^n/n^{3/2})$.

6.4.1 Developing a Dynamic Programming Algorithm

Step 1: Determine the structure of an optimal solution (in this case, a parenthesization).

Decompose the problem into subproblems: For each pair, $1 \le i \le j \le n$, determine the multiplication sequence for $A_{i..j} = A_i A_{i+1} \ldots A_j$ that minimizes the number of multiplications.

Clearly, $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

Original Problem: Determine sequence of multiplication for $A_{1..n}$.

High-Level Parenthesization for $A_{i..j}$: For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k. That is,

Example:

$$A_{3..6} = (A_3 (A_4 A_5))(A_6) = A_{3..5} A_{6..6}$$

Here $k = 5$.

Thus the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

- How do we decide where to split the chain (what is k)? (Search all possible values of k)
- How do we parenthesize the subchains $A_{i..k}$ and $A_{k+1..j}$? (Problem has optimal substructure property that $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively)

Optimal Substructure Property: If final "optimal" solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at final step then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in final optimal solution must also be optimal for the subproblems "standing alone":

If parenthesization of $A_{i..k}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, leading to a contradiction.

Similarly if parenthesization of $A_{k+..j}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, also leading to a contradiction.

Step-2: Recursively define the value of an optimal solution.

As with the 0-1 Knapsack problem, we will store the solutions to the subproblems in an array.

For $1 \le i \le j \le n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive definition.

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$

Proof: Any optimal sequence of multiplication for $A_{i..j}$ is equivalent to some choice of splitting

$$A_{i..j} = A_{i..k}\, A_{k+1..j}$$

for some k, where the sequences of multiplications for $A_{i..k}$ and $A_{k+1...j}$ also are optimal.

Hence $$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}\, p_k p_j$$

We know that, for some k

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}\, p_k p_j$$

We don't know what is, though

But, there are only $j - i$ possible values of k so we can check them all and find the one which returns a smallest cost.

Therefore $$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$

Step-3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$

$m[i, j]$ only defined for $i \le j$.

The important point is that when we use the equation

$$m[i, j] = \min_{i \le k < j}([i, k] + m[k+1, j] + p_{i-1}\, p_k p_j)$$

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k+1, j]$. For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$. Hence, the algorithm should fill the table in increasing order of the length of the matrix-chain.

That is, we calculate in the order

$m[1, 2], m[2, 3], m[3, 4], \ldots, m[n-3, n-2], m[n-2, n-1], m[n-1, n]$

$m[1, 3], m[2, 4], m[3, 5], \ldots, m[n-3, n-1], m[n-2, n]$

$m[1, 4], m[2, 5], m[3, 6], \ldots, m[n-3, n]$

$\vdots$

$m[1, n-1], m[2, n],$

$m[1, n]$

6.4.2 Dynamic Programming Design

When designing a dynamic programming algorithm there are two parts:

1. Finding an appropriate optimal substructure property and corresponding recurrence relation on table items.

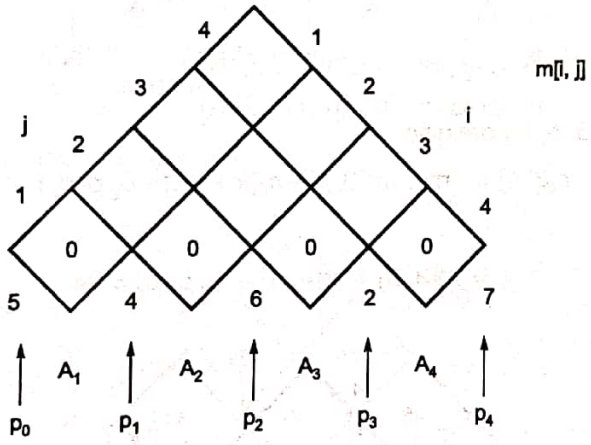
   Example: $$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

2. Filling in the table properly: This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated.
   In our example this means that by the time $m[i, j]$ is calculated all of the values $m[i, k]$ and $m[k+1,j]$ were already calculated.

Example for the Bottom-Up Computation

Example: Given a chain of four matrices A_1, A_2, A_3, and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$, minimum number of multiplication required to multiply A_1, A_2, A_3, and A_4.

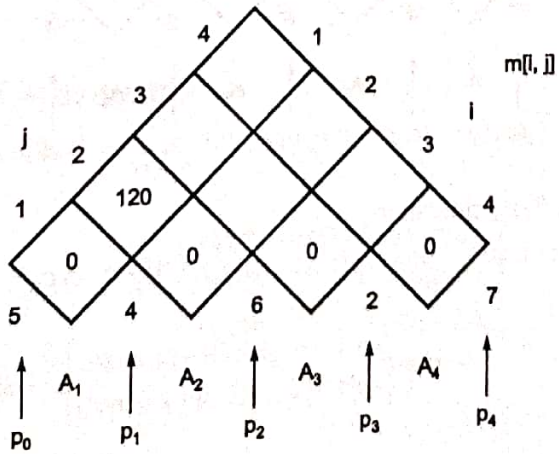
S0: Initialization



Step-1: Computing $m[1, 2]$

$$m[1, 2] = \min_{1 \le k < 2} (m[1, k] + m[k+1, 2] + p_0 p_k p_2)$$

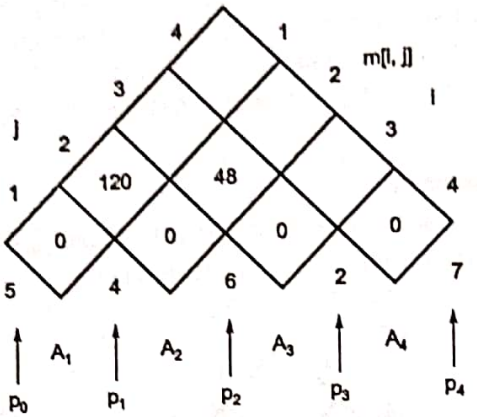
$$= m[1, 1] + m[2, 2] + p_0 p_1 p_2 = 120$$



Step-2: Computing $m[2, 3]$ By definition

$$m[2,3] = \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1 p_k p_3)$$

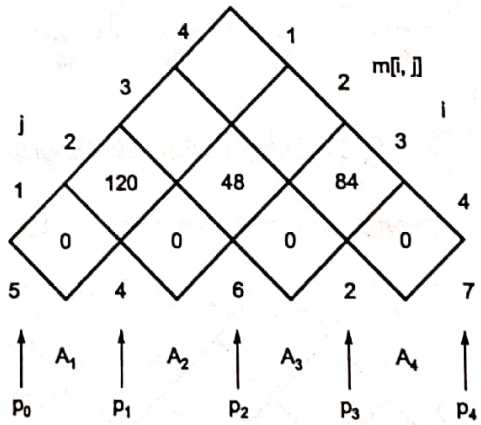
$$= m[2,2] + m[3,3] + p_1 p_2 p_3 = 48$$



Step-3: Computing $m[3, 4]$ By definition

$$m[3,4] = \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4)$$

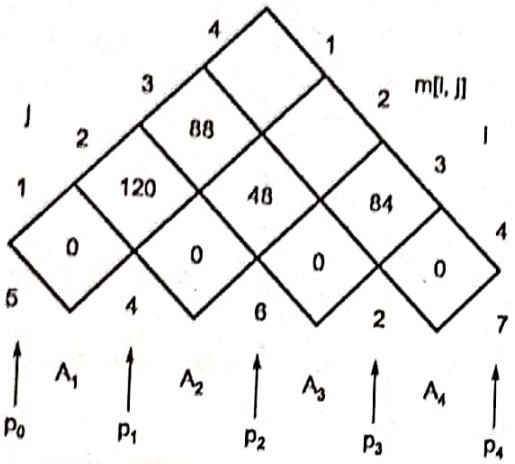
$$= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84$$



Step-4: Computing $m[1, 3]$ By definition

$$m[1,3] = \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3)$$

$$= \min \begin{Bmatrix} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{Bmatrix} = 88$$

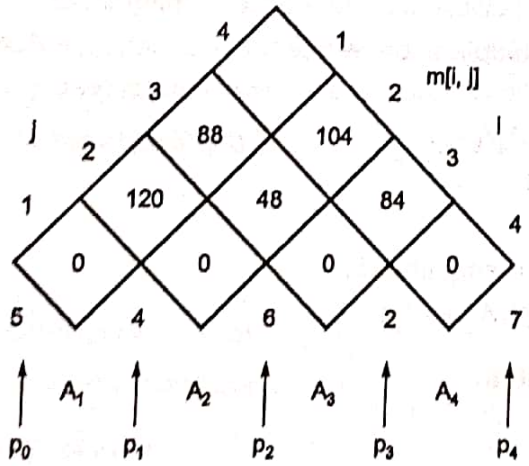


Step-5: Computing $m[2, 4]$ By definition

$$m[2,4] = \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4)$$

$$= \min \begin{Bmatrix} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{Bmatrix}$$

$$= 104$$

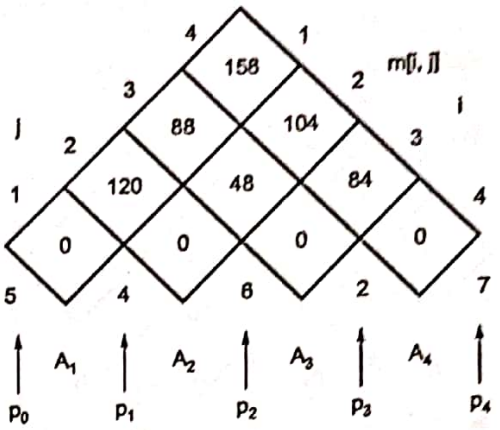


Step-6: Computing $m[1, 4]$ By definition

$$m[2,4] = \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1 p_k p_4)$$

$$= \min \begin{Bmatrix} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{Bmatrix}$$

$$= 158$$



Step-4: Construct an optimal solution from computed information – extract the actual sequence.

Idea: Maintain an array $s[1...n, 1...n]$, where $s[i, j]$ denotes k for the optimal splitting in computing $A_{i..j} = A_{i..k} A_{k+1..j}$. The array $s[1..n,1..n]$ can be used recursively to recover the multiplication sequence.

How to Recover the Multiplication Sequence?

$s[1, n]$ $(A_1 \cdots A_{s[1, n]})(A_{s[1, n]+1} \cdots A_n)$

$s[1, s[1, n]]$ $(A_1 \cdots A_{s[1, s[1, n]]})(A_{s[1, s[1, n]]+1} \cdots A_{s[1, n]})$

$s[s[1, n] + 1, n]$ $(A_{s[1, n]+1} \cdots A_{s[s[1, n]+1, n]}) \times (A_{s[s[1, n]+1, n]+1} \cdots A_n)$

⋮ ⋮

Do this recursively until the multiplication sequence is determined.

Example of Finding the Multiplication Sequence: Consider $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.

$s[1, 6] = 3$ $(A_1 A_2 A_3)(A_4 A_5 A_6)$

$s[1, 3] = 1$ $(A_1 (A_2 A_3))$

$s[[4, 6] = 5$ $((A_4 A_5) A_6)$

Hence the final multiplication sequence is

$(A_1(A_2 A_3))((A_4 A_5) A_6)$

The Dynamic Programming Algorithm

```
Matrix-Chain( )
{
    for (i = 1 to n) m[i, i] = 0;
    for (i = 2 to n)
    {
        for (i = 1 to n – l + 1)
        {
            j = i + l – 1;
            m[i, j] = ∞;
            for (k = i to j – 1)
            {
                q = m[i, k] + m[k + 1, j] + p[i – 1] × p[k] × p[j];
                if (q < m[i, j])
```

```
                {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s; (optimum in m[1, n])
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.

Constructing an Optimal Solution: Compute $A_{1 \ldots n}$.

The actual multiplication code uses the $s[i, j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices A[1..n] and that $s[i, j]$ is global to this recursive procedure. The procedure returns a matrix.

```
Mult(A, s, i, j)
{
    if (i < j)
    {
        X = Mult (A, s, i, s [i, j]);          X is now A_i . . . A_k, where k is s[i, j]
        Y = Mult (A, s, s[i, j] + 1, j);       Y is now A_{k+1} . . . A_j
        return X x Y; multiply matrices X and Y
    }
    else return A[i];
}
```

To compute $A_1 A_2 \ldots A_n$, call mult (A, s, 1, n).

Constructing an Optimal Solution: Compute $A_{1 \ldots n}$

Example of Constructing an Optimal Solution

Compute $A_{1 \ldots 6}$.

Assume that the array $s[1 \ldots 6, 1 \ldots 6]$ has been computed. The multiplication sequence is recovered as follows.

Mult (A, s, 1, 6), $s[1, 6] = 3$, $(A_1 A_2 A_3)(A_4 A_5 A_6)$

Mult (A, s, 1, 3), $s[1, 3] = 1$, $((A_1)(A_2 A_3))(A_4 A_5 A_6)$

Mult (A, s, 4, 6), $s[4, 6] = 5$, $((A_1)(A_2 A_3))((A_4 A_5)(A_6))$

Mult (A, s, 2, 3), $s[2, 3] = 2$, $((A_1)((A_2)(A_3)))((A_4 A_5)(A_6))$

Mult (A, s, 4, 5), $s[4, 5] = 4$, $((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6))$

Hence the product is computed as follows: $(A_1 (A_2 A_3))((A_4 A_5) A_6)$.

6.5 Longest Common Subsequence (LCS) Problem

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that $x_{i_j} = z_j$ for all $j = 1, 2, \ldots, k$. $\langle A_2, C_3, B_4, B_7, C_8, W_{11} \rangle$ is a subsequence of ⟨T, A, C, B, B, W, B, C, W, T, W⟩.

Given two sequences X and Y, a sequence Z is called the common subsequence of X and Y if Z is a subsequence of both X and Y. ⟨A, C, B, B, C, W⟩ is a common subsequence of ⟨T, A, C, B, B, W, B, C, W, T, W⟩ and ⟨A, A, B, C, B, W, B, C, A, A, W, T⟩.

The **longest-common-subsequence problem** is to find a common subsequence Z = LCS(X, Y) with the maximum length from both sequences X and Y.

Denote by

$$X_m = \langle x_1, x_2, \ldots, x_m \rangle$$
$$Y_n = \langle y_1, y_2, \ldots, y_n \rangle$$

Our task then is to find an LCS (X_m, Y_n) of X_m and Y_n.

Key observation (Characterize the structure of an optimal solution)

If $x_m = y_n$, then the LCS includes x_m and y_n. So, it consists of an LCS of $\langle x_1, x_2, \ldots, x_{m-1} \rangle$ and $\langle y_1, y_2, \ldots, y_{n-1} \rangle$, x_m, i.e., LCS $[X_m, Y_n]$ = LCS $[X_m - 1, Y_{n-1}]$, x_m.

If $x_m \neq y_n$, the LCS cannot include both x_m and y_n. So, either is an LCS of

1. $X_{m-1} = \langle x_1, x_2, \ldots, x_{m-1} \rangle$ and $Y_n = \langle y_1, y_2, \ldots, y_n \rangle$, or
2. $X_m = \langle x_1, x_2, \ldots, x_m \rangle$ i.e., LCS (X_m, Y_n) is either LCS (X_{m-1}, Y_n) or LCS (X_m, Y_{n-1}).

The optimal solution recurrence

Let $c[i, j]$ be the length of an LCS of $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$. Using the key observations above

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Compute using increasing i, j until $c[m, n]$ is obtained. Row by row or column by column orders are ok. As usual, keeping track of which option provided the optimum at each step allows us to work backwards from the answer to find the actual subsequence with length $c[m, n]$.

An array b can be used to record which case yielded the optimum at every step.

LCS Length (X,Y)

```
1.  for i ← 1 to m do c[i, 0] ← 0
2.  for j ← 1 to n do c[0, j] ← 0
3.  for i ← 1 to m
4.  for j ← 1 to n do
5.      if(x_i == y_j)
6.          then c[i, j] ← c[i − 1, j − 1] + 1; b[i, j] ← " ↖ "
7.          else if c[i − 1, j] ≥ c[i, j − 1]
8.              then c[i, j] ← c[i − 1, j]; b[i, j] ← " ↑ "
9.              else c[i, j] ← c[i, j − 1]; b[i, j] ← " ← "
```

To find the LCS, it follows the path in b back from $b[m, n]$, where ↖ implies that $x_i = y_j$, ↑ implies the two strings are X_{i-1} and Y_j, while ← implies the two strings are X_i and Y_{j-1}.

Example: X = (A, B, C, B, D, A, B), Y = (B, D, C, A, B, A).

i		j	0	1	2	3	4	5	6
			y_1	B	D	C	A	B	A
0	x_1		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	①	← 1	← 1	↑ 1	↖ 2	↑ 2
3	C		0	↑ 1	↑ 1	②	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	③	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↖ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	④
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Use of the LCS Algorithm

Given a sequence of n elements, find a longest increasing (or decreasing) subsequence of the sequence.

Let $X = \langle x_1, x_2, \ldots, x_n \rangle$, find the longest length palindrome of X, where a palindrome is a subsequence which is identical no matter whether you read the sequence from its left side or from its right side.

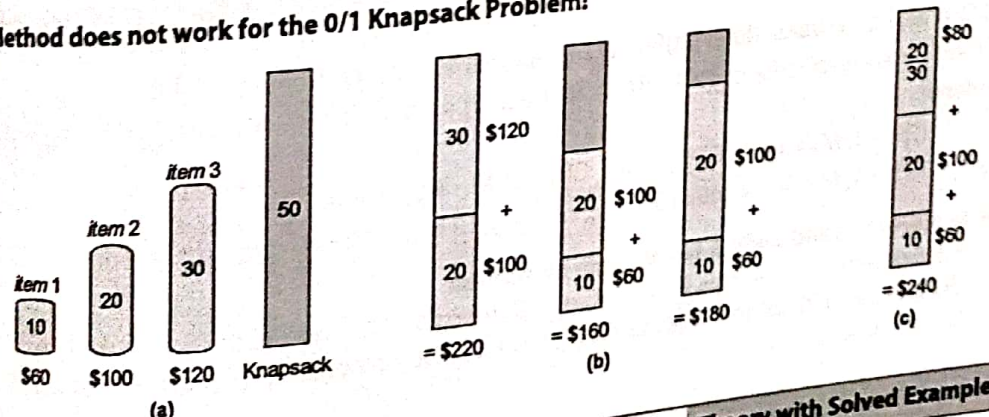
6.6 The 0/1 Knapsack Problem

If we limit the x_i to only 1 or 0 (take it or leave it), this results in the 0/1 Knapsack problem.

Optimization Problem: Find $x_1, x_2, \ldots, x_n$ such that:

$$\begin{cases} \text{Maximum} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{Subject to :} & \sum_{i=1}^{n} w_i \cdot x_i \le m \\ & x_i \in \{0, 1\}, 1 \le i \le n \end{cases}$$

Greedy Method does not work for the 0/1 Knapsack Problem!



The greedy strategy does not work for the 0-1 Knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional Knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The Knapsack Problem

There are two versions of the problem:

1. "Fractional" Knapsack problem.
2. "0/1" Knapsack problem.

1 **Items are divisible:** You can take any fraction of an item. Solved with a greedy algorithm.

2 **Item are indivisible:** You either take an item or not. Solved with dynamic programming.

0/1 Knapsack problem: the brute-force approach

Let's first solve this problem with a straightforward algorithm:

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the maximum value and with total weight less or equal to m.
- Running time will be $O(2^n)$.

Can we do better?

- Improvement over brute force method can be done by an algorithm based on **dynamic programming**.
- Two key ingredients of optimization problems that lead to a dynamic programming solution:
  - (a) **Optimal substructure:** An optimal solution to the problem contains within it optimal solutions to subproblems.
  - (b) **Overlapping subproblems:** Same subproblem will be visited again and again (i.e., subproblems share subsubproblems).

Optimal Substructure of 0/1 Knapsack problem

- Let KNAP (1, n, M) denote the 0/1 Knapsack problem, choosing objects from [1...n] under the capacity constraint of M.
- If $(x_1, x_2, \ldots, x_n)$ is an optimal solution for the problem KNAP (1, n, M), then:
  1. If $x_n = 0$ (we do not pick the n^{th} object), then $(x_1, x_2, \ldots, x_{n-1})$ must be an optimal solution for the problem KNAP (1, $n-1$, M).
  2. If $x_n = 1$ (we pick the n^{th} object), then $(x_1, x_2, \ldots, x_{n-1})$ must be an optimal solution for the problem KNAP (1, $n-1$, M $- w_n$).

Solution in terms of subproblems

Based on the optimal substructure, we can write down the solution for the 0/1 Knapsack problem as follows:

Let C[n, M] be the value (total profits) of the optimal solution for KNAP (1, n, M).

C[n, M] = max (profits for case 1, profits for case 2) = max (C[$n-1$, M], C[$n-1$, M $- w_n$] + p_n).

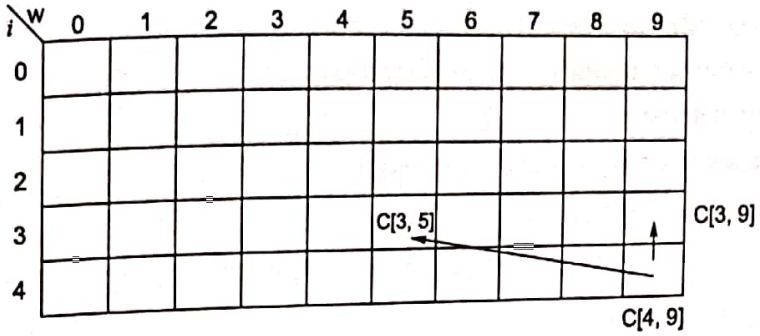
Similarly

$$C[n-1, M] = \max(C[n-2, M], C[n-2, M - w_{n-1}] + p_{n-1}).$$
$$C[n-1, M - w_n] = \max(C[n-2, M - w_n], C[n-2, M - w_n - w_{n-1}] + p_{n-1}).$$

Use a table to store C[·,·] and build it in a bottom up fashion

- For example, if $n = 4$, M = 9; $w_4 = 4$, $p_4 = 2$, then C[4, 9] = max (C[3, 9], C[3, 9 – 4] + 2).
- We can use a 2D table to contain C[·,·]; If we want to compute C[4, 9], C[3, 9] and C[3, 9 – 4] have to be ready.

- Look at the value $C[n, M] = \max(C[n-1, M], C[n-1, M-w_n] + p_n)$, to compute $C[n, M]$, we only need the values in the row $C[n-1,]$.
- So the table $C[\cdot,\cdot]$ can be built in a bottom up fashion: (1) Compute the first row C[0, 0], C[0, 1], C[0, 2]... etc; (2) Row by row, fill the table.



- The term "programming" used to refer to a tabular method, and it predates computer programming.

Construct the table: A recursive solution

- Let $C[i, \varpi]$ be a cell in the table $C[\cdot, \cdot]$; it represents the value (total profits) of the optimal solution for the problem KNAP $(1, i, \varpi)$, which is the subproblem of selecting items in $[1 \ldots i]$ subject to the capacity constraint of ϖ.
- Then $C[i, \varpi] = \max(C[i-1, \varpi], C[i-1, \varpi - w_i] + p_i)$.

Boundary conditions

We need to consider the boundary conditions:

- When $i = 0$; no object to choose, so $C[i, \varpi] = 0$;
- When $\varpi = 0$; no capacity available, $C[i, \varpi] = 0$;
- When $w_i > \varpi$; the current object i exceeds the capacity, definitely we can not pick it. So $C[i, \varpi] = C[i-1, \varpi]$ for this case.

Complete recursive formulation

Thus overall the recursive solution is:

$$C[i, \varpi] = \begin{cases} 0 & \text{if } i = 0 \text{ or } \varpi = 0 \\ C[i-1, \varpi] & \text{if } wi > \varpi \\ \max(C[i-1, \varpi], C[i-1, \varpi - w_i] + p_i) & \text{if } i > 0 \text{ and } \varpi \geq w_i \end{cases}$$

The solution (optimal total profits) for the original 0/1 problem KNAP (1, n, M) is in $C[n, M]$.

Algorithm

```
DP-01KNAPSACK (p[ ], w[ ], n, M)          // n: number of items; M: capacity
    for ϖ := 0 to M      C[0,ϖ] := 0;
    for i := 0 to n      C[i, 0] := 0;
    for i := 1 to n
        for ϖ := 1 to M                   // cannot pick item i
            if (w[i] > ϖ)
                C[i, ϖ] := C[i – 1, ϖ];
            else
                if (p[i] + C[i – 1, ϖ – w[i]]) > C[i – 1, ϖ])
                    C[i, ϖ] := p[i] + C[i – 1, ϖ – w[i]];
```

else

$C[i, \varpi] := C[i-1, \varpi]$;

return C[n, M];

The time complexity of 0/1 Knapsack can be expressed as

$\Theta(M) + \Theta(n) + \Theta(nM) \Rightarrow O(nM)$

An example: Let's run our algorithm on the following data:

$n = 4$ (number of items)

M = 5 (Knapsack capacity = maximum weight)

(w_i, p_i) : (2, 3), (3, 4), (4, 5), (5, 6)

Execution

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

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					
4	0					

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

How to find the actual items in the Knapsack?

- All of the information we need is in the table.
- C[n, M] is the maximal value of items that can be placed in the Knapsack.
- Let $i = n$ and k = M

  if $C[i, k] \neq C[i-1, k]$ then mark the i-th item as in the Knapsack

  $i = i - 1$, $k = k - w_i$. else

  $i = i - 1$

Finding the items

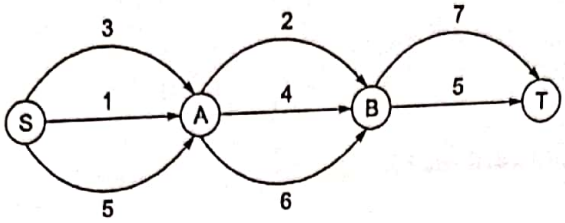
i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
①	0	0	3	3	3	3
②	0	0	3	4	4	7
3	0	0	3	4	5	(7)
4	0	0	3	4	5	(7)

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)

Solution: {1, 1, 0, 0}

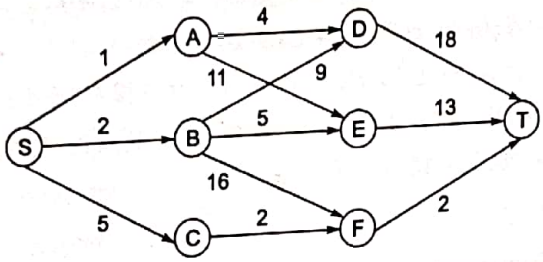
6.7 Multistage Graph

Multistage graph a special case of shortest path problem where vertex set is divided into stages. To find a shortest path in a multi-stage graph



Apply the greedy method: The shortest path from S to T : 1 + 2 + 5 = 8

The shortest path in multistage graphs

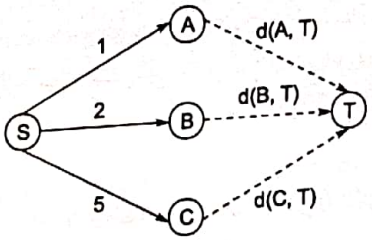


The greedy method can not be applied to this case: (S, A, D, T) 1 + 4 + 18 = 23

The real shortest path is: (S, C, F, T) 5 + 2 + 2 = 9

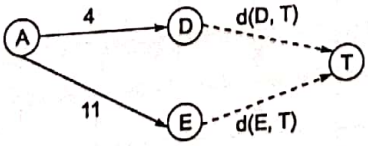
Dynamic programming approach

Dynamic programming approach (forward approach):



$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 5 + d(C, T)\}$

$d(A, T) = \min \{4 + d(D, T), 11 + d(E, T)\} = \min \{4 + 18, 11 + 13\} = 22$

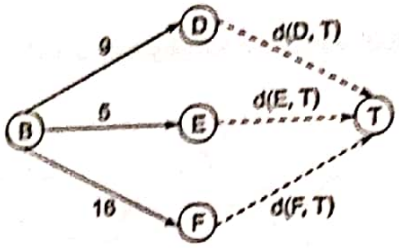


$d(B, T) = \min \{9 + d(D, T), 5 + d(E, T), 16 + d(F, T)\} = \min \{9 + 18, 5 + 13, 16 + 2\} = 18.$

$d(C, T) = \min \{2 + d(F, T)\} = 2 + 2 = 4$

$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 5 + d(C, T)\} = \min \{1 + 22, 2 + 18, 5 + 4\} = 9.$

The above way of reasoning is called **backward reasoning**.



Backward approach (forward reasoning)

$d(S, A) = 1$

$d(S, B) = 2$

$d(S, C) = 5$

$d(S, D) = \min \{d(S, A) + d(A, D), d(S, B) + d(B, D)\} = \min \{1 + 4, 2 + 9\} = 5$

$d(S, E) = \min \{d(S, A) + d(A, E), d(S, B) + d(B, E)\} = \min \{1 + 11, 2 + 5\} = 7$

$d(S, F) = \min \{d(S, A) + d(A, F), d(S, B) + d(B, F)\} = \min \{2 + 16, 5 + 2\} = 7$

$d(S, T) = \min \{d(S, D) + d(D, T), d(S, E) + d(E, T), d(S, F) + d(F, T)\}$

$= \min\{ 5 + 18, 7 + 13, 7 + 2 \} = 9$

Principle of Optimality

Principle of optimality: Suppose that in solving a problem, sequence of we have to make a sequence is decisions $D_1, D_2, \ldots, D_n$. If this optimal, then the last k decisions, $1 < k < n$ must be optimal.

Example: The shortest path problem.

If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j. In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

NOTE: If the recurrence relations are formulated using the forward approach then the relations are solved backwards i.e., beginning with the last decision. On the other hand if the relations are formulated using the backward approach, they are solved forwards.

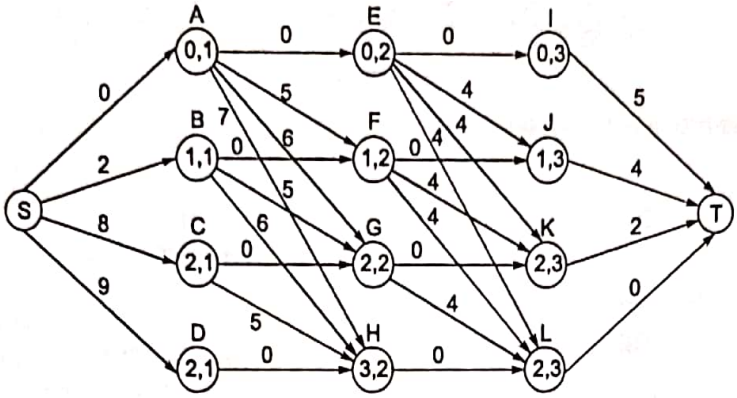
The Resource Allocation Problem

m resources, n projects profit $p(i, j)$: j resources are allocated to project i.

Maximize the total profit.

Resource Project	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

The multistage graph solution



The resource allocation problem can be described as a multistage graph. (i, j) : i resources allocated to projects 1, 2, . . . , j e.g. node H = (3, 2) : 3 resources allocated to projects 1, 2.

Find the longest path from S to T:

(S, C, H, L,T), 8 + 5 + 0 + 0 = 13

2 resources allocated to project 1.

1 resource allocated to project 1.

0 resource allocated to projects 3, 4.

6.8 Traveling-salesman Problem

In the traveling salesman problem, which is closely related to the hamiltonian-cycle problem, a salesman must visit n cities. We can define problem as: the salesman whishes to make a *tour*, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman takes a non-negative integer cost $c(i, j)$ to travel from city i to city j, and the salesman whishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

Traveling-salesman Problem

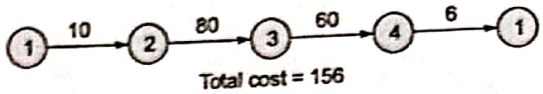
Let TSP (A, R) be the minimum cost required to go from vertex 'A' to all the remaining vertices present in R exactly one and coming back to the source vertex. Let $c(x, y)$ be the cost required to go from vertex 'x' to vertex 'y'. Consider S to be the source vertex hence, the recurrence relation for the problem will be,

$$\text{TSP}(A, R) = \begin{cases} c(A, S) & \text{if } R = \phi \\ \min\{c(A, k) + \text{TSP}(k, R - k) \forall k \in R & \text{if } R \geq 1 \end{cases}$$

Consider the following example consisting of 4 cities, where the starting city is marked as 'A' and the distance between each pair of cities is shown in following adjacency matrix.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 3 \end{array} \\ \begin{bmatrix} 0 & 10 & 20 & 30 \\ 5 & 0 & 80 & 90 \\ 7 & 40 & 0 & 60 \\ 6 & 11 & 21 & 0 \end{bmatrix} \end{array}$$

If greeddy approach is used to solve the problem, then the cost of the path will be as follows:



Total cost = 156

Solving the same problem by dynamic approach

$$TSP(1,\{2,3,4\}) = \min\begin{cases} c(1,2)+TSP(2,\{3,4\}) & \text{...(1)} \\ c(1,3)+TSP(3,\{2,4\}) & \text{...(2)} \\ c(1,4)+TSP(4,\{2,3\}) & \text{...(3)} \end{cases}$$

$$TSP(2,\{3,4\}) = \min\begin{cases} c(2,3)+TSP(3,\{4\}) & \text{...(4)} \\ c(2,4)+TSP(4,\{3\}) & \text{...(5)} \end{cases}$$

$$TSP(3,\{4\}) = c(3,4) + TSP(4,\{\phi\})$$
$$= c(3,4) + c(4,1) = 60 + 6 = 66 \quad \text{...(6)}$$

$$TSP(4,\{3\}) = c(4,3) + TSP(3,\{\phi\})$$
$$= c(4,3) + c(3,1) = 21 + 7 = 28 \quad \text{...(7)}$$

Using (4), (5), (6) and (7)

$$TSP(2,\{3,4\}) = \min\begin{cases} c(2,3)+66 = 80+66 = 146 \\ c(2,4)+28 = 90+28 = 128 \end{cases}$$

$$TSP(2,\{3,4\}) = 118$$

$$TSP(3,\{2,4\}) = \min\begin{cases} c(2,3)+TSP(2,\{4\}) & \text{...(8)} \\ c(3,4)+TSP(4,\{2\}) & \text{...(9)} \end{cases}$$

$$TSP(2,\{4\}) = c(2,4) + TSP(4,\{\phi\})$$
$$= c(2,4) + c(4,1) = 90 + 6 = 96 \quad \text{...(10)}$$

$$TSP(4,\{2\}) = c(4,2) + TSP(2,\{\phi\})$$
$$= c(4,2) + c(2,1) = 11 + 5 = 16 \quad \text{...(11)}$$

Using (8), (9), (10), (11)

$$TSP(3,\{2,4\}) = \min\begin{cases} c(3,2)+96 = 40+96 = 136 \\ c(3,4)+16 = 60+16 = 76 \end{cases}$$

$$TSP(3,\{2,4\}) = 76$$

$$TSP(4,\{2,3\}) = \min\begin{cases} c(4,2)+TSP(2,\{3\}) & \text{...(12)} \\ c(4,3)+TSP(3,\{2\}) \end{cases}$$

$$TSP(2,\{3\}) = c(2,3) + TSP(3,\{\phi\})$$
$$= c(2,3) + c(3,1) = 80 + 7 = 87 \quad \text{...(13)}$$

$$TSP(3,\{2\}) = c(3,2) + TSP(2,\{\phi\})$$
$$= c(3,2) + c(2,1) = 40 + 5 = 45 \quad \text{...(11)}$$

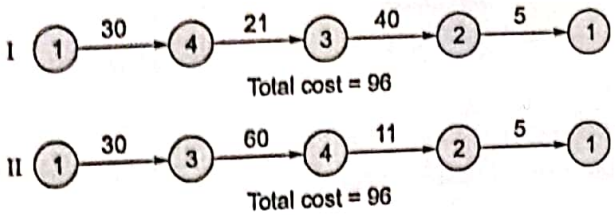
Using (12), (13) and (14)

$$TSP(4,\{2,3\}) = \min\begin{cases} 11+87 = 98 \\ 21+45 = 66 \end{cases}$$

$$TSP(4,\{2,3\}) = 66$$

Hence,
$$TSP(1,\{2,3,4\}) = \min\begin{cases} 10+118 = 128 \\ 20+76 = 96 \\ 30+66 = 96 \end{cases}$$

Hence the cost of path is 96 and two possible paths.



Analysis of Traveling salesman problem:

TSP (n) will generate n level n-array tree without using dynamic programming.

So, number of function call = $(n-1)(n-2)(n-3)\ldots(2)(1) = (n-1)!$

Time complexity = Number of function calls × Cost of each function calls

$= (n-1)! \times O(n) = n! = O(n^n)$

Space complexity = Input + Extra (in form of stack)

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

In above procedure number of function call is repeated, so number of distinct function call is equal to total number of function call.

So time complexity using dynamic programming:

$= (n-1)!$ function call $\times O(n)$ cost of each function call

$= O(n!) = O(n^n)$

Space complexity = Input + Extra (Table + Stack)

$= O(n^2) + n + n!$

$= O(n^n)$

Using dynamic programming TSP give same time complexity but more space complexity. It is one of the NP-complete problem.

Summary

- **Dynamic Programming (DP):**
  (a) DP ≈ recursion + memorization (i.e. reuse)
  (b) DP ≈ controlled brute force
- DP results in an efficient algorithm, if the following conditions hold:
  (a) The optimal solution can be produced by combining optimal solutions of subproblems;
  (b) The optimal solution of each subproblem can be produced by combining optimal solutions of sub-subproblems, etc;
  (c) The total number of subproblems arising recursively is polynomial.

Student's Assignments

Q.1 Let X = $\langle x_1, x_2, \ldots x_m \rangle$ and Y = $\langle y_1, y_2, \ldots y_n \rangle$ be sequences. What is the running time to find the longest common subsequence of X and Y using dynamic programming.

(a) $O(m+n)$ (b) $O(mn)$
(c) $O(m^n)$ (d) $O(n^m)$

Q.2 Let X = abcbdab and Y = bdcaba. Find the length of longest common subsequence (LCS) of X and Y.

(a) 3 (b) 4
(c) 5 (d) 6

Q.3 If the towers of hanoi problem is solved using divide and conquer approach then find the running time to solve the problem with 'n' disks?

(a) $O(n)$ (b) $O(n \log n)$
(c) $O(2^n)$ (d) $O(n^n)$

Q.4 Let G = (V, E) be a directed graph. Each edge of G is represented as (i, j) with length $l[i, j]$. If there is no edge from i to j then $l[i, j] = \infty$. Assume n vertices in V and $d^k_{i,j}$ is the length of shortest path from i to j that does pass through any vertex in {1, 2, . . ., n}.

$$d^k_{ij} = \begin{cases} l[i, j] & \text{if } k = 0 \\ \min\{A, B\} & \text{if } 1 \le k \le n \end{cases}$$

If the above $d^k_{i,j}$ computed recursively to find all pairs shortest path, identify A and B respectively?

(a) $d^{k-1}_{i,j}$ and $d^{j-1}_{i,j} + d^{j-1}_{k,j}$

(b) $d^{k-1}_{i,j}$ and $d^{k-1}_{i,k} + d^{k-1}_{k,j}$

(c) $d^k_{i,j}$ and $d^k_{i,k} + d^k_{k,j}$

(d) $d^k_{i,j}$ and $d^k_{i,k} + d^k_{j,k}$

Q.5 Floyd algorithm uses $n+1$ martices $D_0, D_1, \ldots D_n$ of dimensions $n \times n$. D_n contains the cost for all pairs shortest path.

Given $D_0 = \begin{bmatrix} 0 & 2 & 9 \\ 8 & 0 & 6 \\ 1 & \infty & 0 \end{bmatrix}$

Where

$d^0_{11} = 0, d^0_{12} = 2, d^0_{13} = 9$
$d^0_{21} = 8, d^0_{22} = 0, d^0_{23} = 6$
$d^0_{31} = 1, d^0_{32} = \infty, d^0_{33} = 0$

Find D^1 using the d^k_{ij} recursive equation.

(a) $\begin{bmatrix} 0 & 2 & 9 \\ 8 & 0 & 6 \\ 1 & 3 & 0 \end{bmatrix}$ (b) $\begin{bmatrix} 0 & 2 & 8 \\ 7 & 0 & 6 \\ 1 & 3 & 0 \end{bmatrix}$

(c) $\begin{bmatrix} 0 & 2 & 9 \\ 8 & 0 & 6 \\ 1 & \infty & 0 \end{bmatrix}$ (d) $\begin{bmatrix} 0 & 2 & 9 \\ 8 & 0 & 6 \\ 1 & 2 & 0 \end{bmatrix}$

Q.6 Running time of 0/1 Knapsack problem using dynamic programming is _____. Assume n is the number of items and w is the capacity of Knapsack.

(a) $O(n^2)$ (b) $O(n^w)$
(c) $O(w^n)$ (d) $O(nw)$

Q.7 Finding Hamiltonian cycle in a graph requires _____ running time.

(a) Polynomial (b) Non polynomial
(c) Both (a) and (b) (d) None of these

Answer Key:

1. (b) **2.** (b) **3.** (c) **4.** (b) **5.** (d)
6. (d) **7.** (b)