Algorithm - A sequence of steps to carry out a specific task

Characteristics -
input : value or a set of values
output : value or a set of values
finiteness : Terminates after finite no. of steps
definiteness : Each step should be a basic operation
unambiguity : Each step is clearly defined.

Why to analyze an Algorithm ?
- To understand the scalability, That is to evaluate the behaviour for large inputs
- To estimate the resource usage

Resources - CPU, Memory } bounded resources

$1 GHz = 1 \times 10^3$ clock/sec

Running time of an algorithm is a function of its input size n

$f(n)$ is a function on n
$f(n) = 5n + 7$
$f(n) = 2n^2 + 6n - 2$

Eg:- Quick sort { $O(n \log n)$ - time, $O(n)$ - space

There is a trade off between time and space

Queue
Priority queue - (Array) - $O(n)$
Priority queue - (Heap data structure) - $O(\log_2 n)$

Scanned by CamScanner

How to analyze an Algorithm?

- Running time depends on hardware, software (compiler)
- Analyze algorithm independent of the hardware
- RAM model (sequential analysis)
- We use pseudo code to write algorithm
  ↓
  high level discription

Operations { Arithmetic +, -
Logical &&, ||, !
relational <, >, <=, >= }

Eg:-

```
j=0 ; sum=0;                    — 2
while (j<n) do                  — n+1 times
    sum = sum+1   →2  } — n times   n(2+2) = 4n
    j = j+1       →2  }
print sum;                      — 1
```

Total = 2 + (n+1) + 4n + 1
= 5n + 4 = O(n)

Insertion sort (A, n)

```
Begin
for (j=2 to n)                          — n c1
    i=j-1 ; key = A[j];                 — (n-1) c2
    while (i>0 && A[i] > key)           — tj c3
        A[i+1] = A[i]   // shift right   — (tj-1) c4
        i = i-1;                        — (tj-1) c5
    A[i+1] = key;                       — (n-1) c6
End
```

Let total time be T(n)

Scanned by CamScanner

$$T(n) = nc_1 + (n+1)c_2 + t_j c_3 + (t_j - 1)(c_4 + c_5) + (n-1)c_6$$

Best case -

$t_j = 1$

$$T(n) = nc_1 + (n-1)c_2 + c_3 + (n-1)c_6$$

$$= (\quad)n + (\quad)$$

$$= nc_1 + nc_2 - c_2 + c_3 + nc_6 - c_6$$

$$= n(c_1 + c_2 + c_6) + (c_3 - c_2 - c_6)$$

$= O(n)$

Worst case

$$t_j = \sum_{j=2}^{n} j \qquad \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

$$= \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

$$T(n) = nc_1 + (n-1)c_2 + \left(\frac{n(n+1)}{2} - 1\right)c_3 + \left(\frac{n(n+1)}{2} - 2\right)(c_4 + c_5) + (n-1)c_6$$

$$= nc_1 + nc_2 - c_2 + \frac{n^2}{2}c_3 + \frac{n}{2}c_3 - c_3 +$$

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

$= O(n^2)$

Average case -

$= O(n^2)$ because only constants change

Scanned by CamScanner

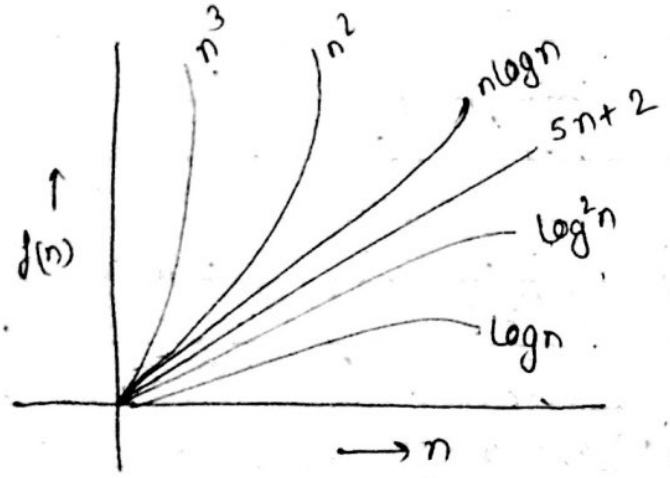
Asymptotic analysis

Growth of functions

$5n+2$
$6\log n + 6$
$n\log n + 10$
$5n^2 + 1$
$3n^3 - 2n + 1$
$\log^2 n = (\log n)^2$

$6\log n + 6$
$\log^2 n$
$5n+2$
$n\log n + 10$
$5n^2 + 1$
$3n^3 - 2n + 1$

↓ Gro... fast



Q For an input of size n insertion sort runs while merge sort runs $64n\log n$. For wh value of n does insertion sort beat m

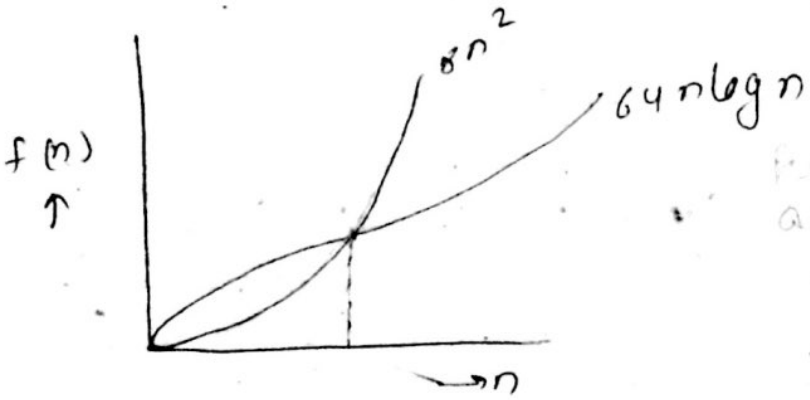
Let $n = 5$

$8n^2 = 8 \times 5 \times 5$
$= 200$

$64n\log n = 64 \times 5 \times$
$= 320 \times$
$=$

For small value of n insertion sort is b

Scanned by CamScanner



$n = 6$

$8n^2 = 288$ < $64n\log n = 64 \times 6 \times \log 6 = 298.8100$

$n = 7$

$8n^2 = 392$ > $64n\log n = 378.603$

$\therefore n = 7$

8 Consider two machine A and B. Machine A has speed of 10^9 inst/sec. Machine B has speed of 10^7 inst/sec. There are two algorithm

Algorithm $I_1 = 2n^2$

Algorithm $I_2 = 50\, n\log n$

Given $n = 1 \times 10^6$. How much time it takes for machine A and B for algorithm I_1 and algo I_2

		I_1	I_2
mac	A	?	?
mac	B	?	?

Scanned by CamScanner

machine A $= 10^9$ inst/sec

1 sec $= 10^9$ ins

$l_1 = 2n^2 = 2(10^6)^2$

$= \frac{2 \times 10^{12}}{10^9} = 2 \times 10^3$

$= 2000$

Machine B $- 10^7$ inst/sec

1 sec $= 10^7$ inst

$= 2(10^6)^2$

$= \frac{2 \times (10^6)^2}{10^7} = 2 \times 10^5$

machine A

$l_2 = 50 n \log n$

$= 50 \times 10^6 \times 6 \log 10$

$= 50 \times 10^6 \times 6$

$= \frac{50 \times 10^6 \times 6}{10^9} = \frac{50 \times 6}{10^3} = 3 \times 10^{-1}$

Machine B

$= \frac{50 \times 10^6 \times 6}{10^7} = 5 \times 6 = 30$

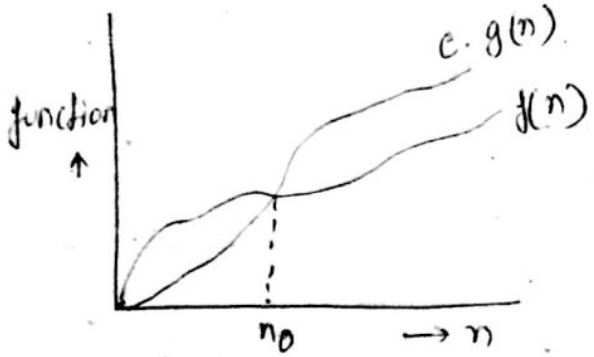
Scanned by CamScanner

Asymptotic Notations

1. Big-oh Notation $O()$
   - Asymptotic upper bound

Definition : Given 2 functions $f(n)$ and $g(n)$ we write $f(n) = O(g(n))$ if there exists positive constants c and n_0 such that

$0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$



asymptotic upper bound

worst case time

Eg:

Prove that $3n^2 + n = O(n^2)$

$f(n) = 3n^2 + n$, $g(n) = n^2$

$3n^2 + n \leq c \cdot n^2$

Let $c = 4$; $n = 1$

$n=1$ $\quad 3 + 1 \leq 4$ ✓

$n=2$ $\quad 12 + 2 \leq 4 \cdot 4$ ✓

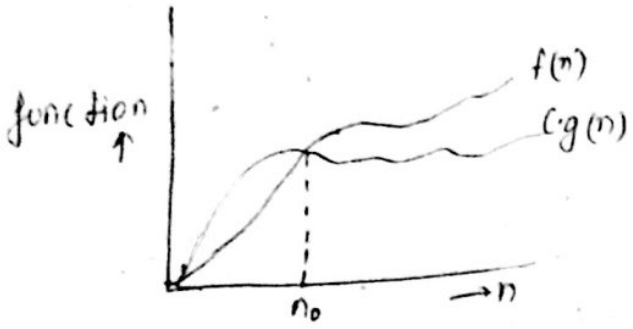
$\therefore c = 4 \quad n_0 = 1$

$3n^2 + n = O(n^2)$

2. Big Omega $\Omega()$
   - Asymptotic lower bound

Defi : Given $f(n)$ and $g(n)$ we write $f(n) = \Omega(g(n))$ if there exists ~~two~~ +ve constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

Scanned by CamScanner



asymptotic lower bound

Best case time

Eg:- Prove that $3n^2+n = \Omega(n^2)$

$3n^2+n \geq c \cdot n^2$

$c=1$, $n=1$

$n=1$ $\quad 3+1 \geq 1.1$

$n=2$ $\quad 12+2 \geq 1.4$

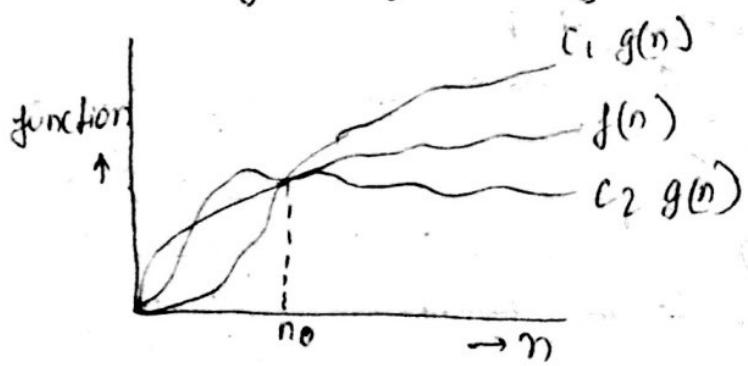
$\therefore c=1$, $n_0=1$

3. Big-Theta $\Theta(\ )$

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

$f(n) = \Theta(g(n))$ if there exists positive constants c_1, c_2 and n_0 such that

$c_1 g(n) \geq f(n) \geq c_2 g(n)$ for $n \geq n_0$



asymptotic tight bound

Average case time

1.
2.
3.

Eg:- Prove that $3n^2+n = \Theta(n^2)$

$c_1 = 4$, $c_2 = 1$, $n_0 = 1$

$c_1 \cdot n^2 \geq 3n^2+n \geq c_2 \cdot n^2$

Scanned by CamScanner

Prove that $3\log n + \log\log n = \Theta(\log n)$

$f(n) = 3\log n + \log\log n$, $g(n) = \log n$

$c_1 \cdot \log n \geq 3\log n + \log\log n \geq c_2 \log n$

$f(n) \leq c \cdot g(n)$

$c_1 = 4$, $n = 2$

$3\log 2 + \log\log 2 \leq 4 \log 2$

$3.1 + 0 \leq 4.1$

$c_2 = 2$

$2.1 \leq 3.1 + 0 \leq 4.1$

$\therefore \quad c_1 = 4, \; c_2 = 2, \; n = 2$

$f(n) = 5n^2 + 3n - 6$

$f(n) = \Theta(n^2)$

$f(n) = 6n\log n$ (poly logarithm function)

Algorithm Design Techniques

Divide and Conquer

3 steps

1. Divide → Divide the problem into subproblems
2. Solve each sub problem (conquer)
3. Combine the solution of subproblems.

Eg:- Merge_sort (A, p, r)

if (p < r)

$q = \lfloor (p+r)/2 \rfloor$

Merge_sort (A, p, q)

Scanned by CamScanner

(11) Analyzing divide & conquer algorithm → $T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ a\,T\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$

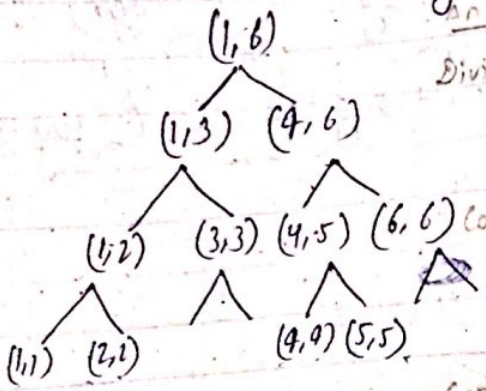
Merge-sort (A, p+1, r)
Merge (A, p, q, r)

Analysis of merge sort

Divide: The divide step just computes the middle of the subarray, which takes constant time. $D(n) = \Theta(1)$

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

Combine: merge sort procedure on an n-element subarray takes time $\Theta(n)$ so $C(n) = \Theta(n)$



Merge (A, p, q, r)
- L_1 [p .... q]
- L_2 [q+1 ... r]
- combine L_1 & L_2 } $\Theta(n)$

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \end{cases}$$

Let $T(n)$ be the time for merge sort

Recurrence equation

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

$$T(n) = 2\,T\left(\frac{n}{2}\right) \quad \text{if } n > 1$$

Recurrence Equation -

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$\lceil \frac{n}{2} \rceil$ $\lfloor \frac{n}{2} \rfloor$

General form

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Scanned by CamScanner

$a \rightarrow$ no. of subproblems

$\frac{n}{b} \rightarrow$ size of each subproblem

$f(n)$ represents the compilation

Solving recurrences

1. Substitution method
2. Recursion tree method
3. Master method

1. Substitution method

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Guess $T(n) \leq c\, n \log n$

Assume that guess is true for some $k < n$

$$T(k) \leq ck \log k \qquad k < n$$

$$T(n/2) \leq c \cdot (n/2) \log(n/2)$$

Substitute

$$T(n) \leq 2 \cdot c \cdot \frac{n}{2} \log\left(\frac{n}{2}\right) + cn$$

$$\leq cn(\log n - \log 2) + cn$$

$$\leq cn\log n - cn + cn$$

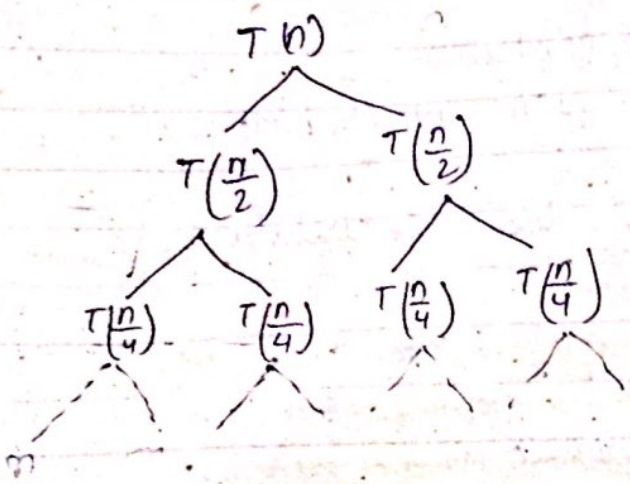
$\therefore T(n) \leq cn\log n$ for some $c > 1$

$$T(n) = O(n \log n)$$

2. Recursion Tree method

Solve $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ using recursion tree method

Scanned by CamScanner



	Cost
$2^0 = 1$	cn
$2^1 = 2$	$\frac{cn}{2} + \frac{cn}{2} = cn$
$2^2 = 4$	$\frac{4cn}{4} = cn$
⋮	
$2^{\log n} = n$	cn

Total cost $= cn(\log n + 1)$

$T(n) = cn(\log n + 1)$

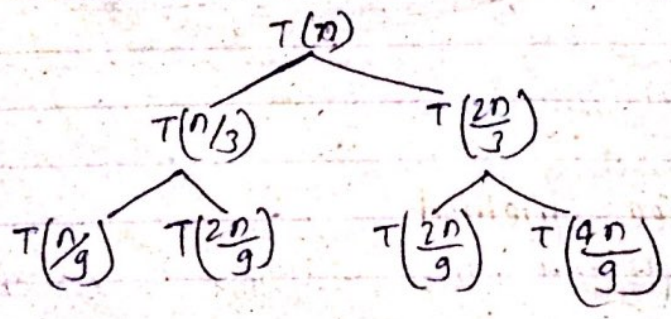
$\leq cn \log n$

$T(n) = O(n \log n)$

$n = 2^k$

2^k

Solve $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$ using recursion tree and verify using substitution method



cn

$\frac{cn}{3} + \frac{2cn}{3} = cn$

$= cn$

$\log_3 n$

Scanned by CamScanner

Total cost $\leq cn \log_{3/2} n$

$T(n) \leq cn \log_2 n$

$T(n) = O(n \log n)$

Verify:- Guess $T(n) = O(n \log n)$

$T(k) \leq c \cdot k \log k$ for $k < n$

Substitute

$$T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{3}\right)\log\left(\frac{n}{3}\right) + c\frac{2n}{3}\log\left(\frac{2n}{3}\right) + dn$$

$O(n) \leq d \cdot n$

$$\leq c\frac{n}{3}(\log n - \log 3) + \frac{2cn}{3}(\log 2n - \log 3) + dn$$

$$\leq \frac{cn}{3}\log n - \frac{cn}{3}\log 3 + \frac{2cn}{3}\log 2n + \frac{2cn}{3}\log n - \frac{2cn}{3}\log + dn$$

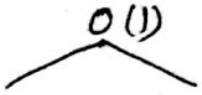
$$= n\log n\left(\frac{c}{3} + \frac{2c}{3}\right) + cn\left(\frac{2}{3} - \frac{1}{3}\log 3 - \frac{2}{3}\log 3\right) + dn$$

$T(n) \leq cn\log n$ for some c. so $T(n) = O(n \log n)$

Solve (i) $T(n) = T\left(\frac{n}{2}\right) + O(1)$ $O(\log n)$

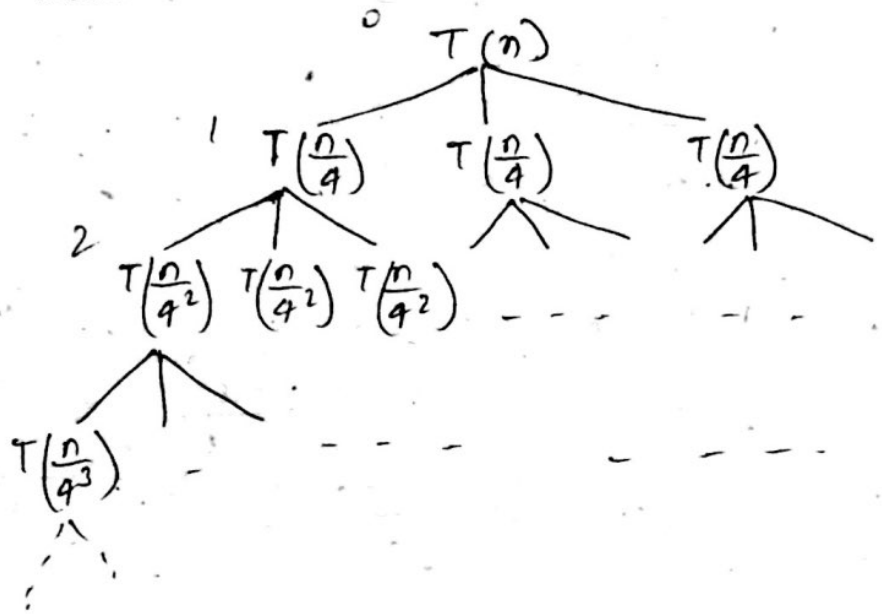
(ii) $T(n) = 2T\left(\frac{n}{2}\right) + O(1)$ O

(i) $T(n) = T\left(\frac{n}{2}\right) + O(1)$



Scanned by CamScanner

Q Solve $T(n) = 3T(n/4) + \Theta(n^2)$ using recursion tree



The no. of nodes at level $i = 3^i$ $(i=0, 1, \ldots)$

The subproblem size at level $i = \frac{n}{4^i}$ $(i=0, 1, \ldots)$

The subproblem hits 1 when $\frac{n}{4^i} = 1$

$$\frac{n}{4^i} = 1$$

$$n = 4^i$$

$$\log n = i \log 4$$

$$i = \log_4 n \qquad \therefore i = 0, 1, \ldots \log_4 n$$

The cost of each subproblem at level $i = c \cdot \left(\frac{n}{4^i}\right)^2$

Cost at level i = No. of nodes at level i × cost of eac

$$= 3^i \times c\left(\frac{n}{4^i}\right)^2$$

$$= \left(\frac{3}{16}\right)^i \cdot cn^2$$

Scanned by CamScanner

Total cost is $\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c n^2$ + cost of last level

cost of last level $= 3^i \cdot 1 = 3^{\log_4 n}$

$= n^{\log_4 3}$

Total cost $= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c n^2 + (n)^{\log_4 3}$

$1 + x^1 + x^2 + \cdots x^n = \frac{1 - x^{n+1}}{1 - x}$ for $x > 1$

$1 + x^1 + x^2 + \cdots \infty = \frac{1}{1 - x}$ $|x| < 1$

Total cost = from formula here $x = \frac{3}{16}$

$\therefore T(n) \leq \sum_{0}^{\infty} \left(\frac{3}{16}\right)^i c n^2 + n^{\log_4 3}$

$= c n^2 \frac{1}{\left(1 - \frac{3}{16}\right)} + n^{\log_4 3}$

$= \frac{16}{13} c n^2 + n^{\log_4 3}$

$\therefore T(n) = O(n^2)$

Q Solve $T(n) = 4\,T\left(\frac{n}{2}\right) + n^3$

Scanned by CamScanner

Master theorem

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Let $a > 1$ and $b > 1$ be constant

$T(n)$ can be bounded asymptotically as follows:

(i) if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$

then $T(n) = \Theta\left(n^{\log_b a}\right)$

(ii) if $f(n) = \Theta\left(n^{\log_b a}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$

(iii) if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$

$a \cdot f\left(\frac{n}{b}\right) \le c \cdot f(n)$ for some $c < 1$ and

sufficiently large n then $T(n) = \Theta(f(n))$

Q Solve $T(n) = 9\,T\left(\frac{n}{3}\right) + n$

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

$n^{\log_b a} = n^{\log_3 9} = n^2$ and $f(n) \neq n^2$

$f(n) = n^{\log_3 9 - 1} = n^{2-1}$ $\quad \epsilon = 1 > 0$

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

Q Solve $T(n) = 3\,T\left(\frac{2n}{3}\right) + 1$

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

Scanned by CamScanner

$\log_3 a = \log_{3/2} 3 = \frac{\log 3}{\log 1.5} \approx 2$

$f(n) = n^{2-2} \qquad \epsilon = 2$

$\therefore \Theta(n^2)$

Q Solve $T(n) = T\left(\frac{2n}{3}\right) + 1$

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

$a = b^k$
$1 > \frac{3}{2}^0 \quad p = 0$

$\log_a^b = 0$

$n^{\log_a^b} \quad n^0 = 1 = f(n)$

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

$\Theta(\log n)$

Q Solve $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

$a = 3, b = 4, k = 1, p = 1$
$a < b^k$
$3 < 4^1 \quad p > 0$
$\Theta(n^k \log^p n)$
$\Theta(n \log n)$

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

$n^{\log_b a} = n^{\log_4 3} = n^{0.78}$

$3 \cdot f\left(\frac{n}{4}\log\left(\frac{n}{4}\right)\right) \leq c \cdot n\log n$

$3 \cdot f\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \left(\frac{3}{4}\right) n\log n$

$c = \frac{3}{4} < 1$

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

Scanned by CamScanner

Q Solve $T(n) = 5T\left(\frac{n}{2}\right) + \theta(n^2)$

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

$\log_b a = \log_2 5$

$n^{\log_2 5} > \log_2 5\ n^2$

case 1 applies $T(n) = \theta(n^{\log_2 5})$

Q Solve $T(n) = 27.T\left(\frac{n}{3}\right) + \theta(n^3 \log n)$

$a = 27$, $b = 3$, $\log_3 27 = 3$

$f(n) = n^3 \log n$ and n^3

case 2 (b) $f(n) = n^{\log_b a} \log^k n$

then $T(n) = \theta\ n^{\log_b a} \log^{k+1} n$

$T(n) = \theta(n^3 \log^2 n)$

Q Solve $T(n) = 5T\left(\frac{n}{2}\right) + \theta(n^3)$

$a = 5$, $b = 2$, $f(n) = n^3 = n^{\log_b a}$

$n^{\log_2 5} < n^3$

case 3 $T(n) = \theta(n^3)$

Q solve $T(n) = 27T\left(\frac{n}{3}\right) + \theta\left(\frac{n^3}{\log n}\right)$

$a = 27$, $b = 3$, $f(n) = n^3 \log^{-1} n$

$f(n) \neq \theta(n^3 \log^k n)$ for $k \geq 0$

Scanned by CamScanner

Master method cannot be applied

a. $f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ (in prev. ques.)

s. $f\left(\frac{n}{2}\right) \leq c \cdot f(n)$

s. $\left(\frac{n^3}{8}\right) \leq c \cdot n^3$ true for $c = \frac{1}{8} < 1$

$f(n)$ is polynomially larger.

Divide and Conquer
basics : Lower bounds

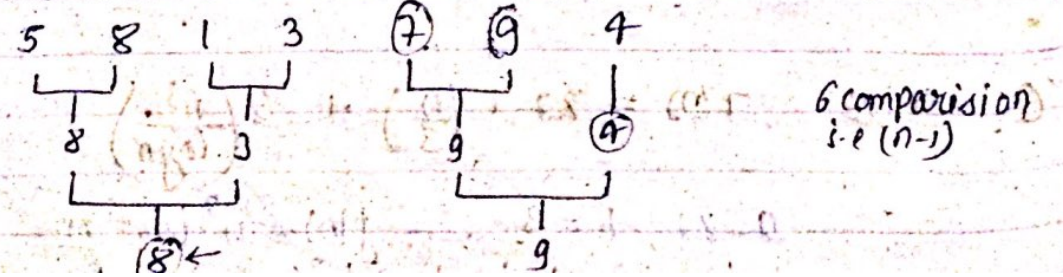
Q Find minimum in an unsorted array of size 'n'
$= n-1$

Q Find both minimum and maximum in an unsorted array of size 'n'
$(n-1) + (n-2) = 2n-3$

small { } $= n/2$
big { } $= n/2$

$\left(\frac{n}{2}-1\right) + \left(\frac{n}{2}-1\right) + \frac{n}{2} = \frac{3n}{2} - 2$

Find the second maximum element



6 comparision
i.e $(n-1)$

Scanned by CamScanner

height of tree is $\lceil \log n \rceil$

Total no. of comparisions for (9) $= (n-1) + (\lceil \log n \rceil - 1)$

$= n + \lceil \log n \rceil - 2$

Two types of sorting

(i) Comparision based sorting

(ii) Linear time sorting

(i) Comparision based sorting
$= n \log n$ (No restriction)

(ii) Linear time sorting
Eg:- Counting sort
Radix sort

Finding i^{th} order statistics
Find the i^{th} smallest element

Median : middle element after sorting

Find $\lfloor \frac{n}{2} \rfloor^{th}$ element

(i) sort - $O(n \log n)$

Algorithm - i^{th} smallest (A, n, i);
sort (A, n);
return A[i];
} $O(n \log n)$

Scanned by CamScanner

1. Partition the array
2. Check the no. of elements in lower partition (say k)

if (i<k) ....
else (i>k)

ith Order Statistics

```
ith smallest (A, n, i) {
        sort (A, n) ;        // select a good sorting method   O(n log n)   } O(n log n)
        return A[i] ;        O(1)
}
```

OR

```
Random_select (A, p, r, i) {   // select ith smallest (in
                               // p is start index
                               // r is end index
    if (p < r) {
        q = Random_partition (A, p, r)
        k = q - p + 1 ;        // no. of elements in lower
                               // partition
        if (i < k)
            Random_select (A, p, q-1, i) ;
        else
            Random_select (A, q+1, r, i-k) ;
    }
}
```

Random(a,b) returns an integer between a & b inclusive (with each such integer being equally likely).
Eg: Random(0,1) returns 0 or 1 each with probability ½ and Random(1,2) returns 1 or 2 each with probability ½.

```
Random_partition (A, p, r) {
    i = Random (p, r) ;                    O(1)
    Exchange (A[r], A[i])                  O(1)
    x = A[r] ;
    i = p - 1 ;
    for (j = j+1 to r-1)                   O(n)
        if (A[j] ≤ x) {
            i = i + 1
            exchange (A[i], A[j]) ;
```

Scanned by CamScanner

exchange (A(r), A(i+1));
return i+1;
}

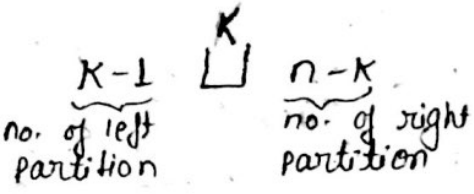
(A, 1, 6, 2)
(A, 1, 4, 2)
(A, 2, 4, 1)
(A, 2, 2, 1)

Worst case running time

$T(n) = T(n-1) + O(n)$
$= T(n-1) + c \cdot n$
$\doteq O(n^2)$

Analysis of Random-select

max size of each partition
$= \max(n-k, k-1)$



say n is odd (n=5)

k	k-1	n-k
1	0	4
2	1	3
3	2	2
4	3	1
5	4	0

say n is even (n=6)

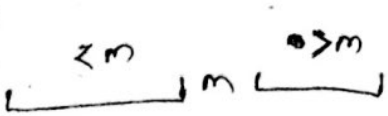
k	k-1	n-k
1	0	5
2	1	4
3	2	3
4	3	2
5	4	1
6	5	0

Scanned by CamScanner

Selection in worst case linear time

Find minimum no. of comparison to sort 5 elements or find median.

- median method

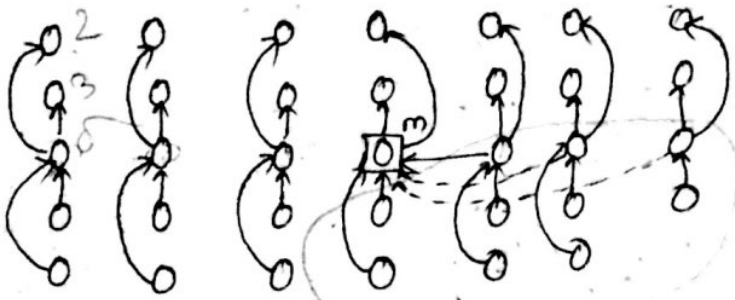


Algorithm SELECT (A, n, i)
{
1. Divide the n elements into $\lfloor n/5 \rfloor$ groups of 5 element each.
   A last group may have less than 5 elements $n \pmod 5$
2. Find the median of each group of size 5 (take lower median if size is even for last group) so we get $\lceil \frac{n}{5} \rceil$ medians.
3. Repeat steps 1 and 2 until we get a single median say m.
4. Partition Array A around m using a modified PARTITION algorithm.
5. Let k be the no. of elements in the lower partition including m
6. if (i == k) return A[i];
7. if (i < k) call SELECT(A, k, i);
   else
   SELECT(A, n-k, i-k)
}

Scanned by CamScanner

Grouping should be >4
$n \geq 140$

Analysis, $n = 34$



Total no. of groups $= \lceil \frac{n}{5} \rceil$

The no. of groups that contribute at last 3 elements $> m$ is $\left(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2\right)$

The no. of elements greater than m is

$$>= 3\left(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2\right)$$

(at least) $>= 3\left(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2\right) = \frac{3n}{10} - 6$

No. of elements smaller than m is at most

$$n - \left(\frac{3n}{10} - 6\right)$$

$$<= \frac{7n}{10} + 6$$

Recurrence is $T(n) = T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$ —①

for $n >$ —

$T(n) = 1$ for $n <$ —

Guess $T(n) = O(n)$, $T(n) < cn$

so, $T(k) \leq ck$ for $k < n$

substitute in ①

Scanned by CamScanner

$T(n) \leq c\left[\frac{n}{5}\right] + c\left(\frac{7n}{10}+6\right) + an$

$= c\left(\frac{n}{5}+1\right) + c\left(\frac{7n}{10}+6\right) + an$

$= \frac{cn}{5} + c + \frac{7cn}{10} + 6c + an$

$= \frac{9cn}{10} + 7c + an$

$T(n) = cn - \frac{1}{10}cn + 7c + an$

$= cn - \left(\frac{1}{10}cn - 7c - an\right)$

$T(n) \leq cn$ if $\frac{cn}{10} - 7c - an \geq 0$

$\frac{cn}{10} - an \geq 7c$

$cn - 10an \geq 70c$

$n(c-10a) \geq 70c$

$n \geq \frac{70c}{c-10a}$

$\frac{cn}{10} \geq 7c + an$

$cn \geq 70c + 10an$

$c(n-70) \geq 10an$

$c \geq \frac{10an}{n-70}$ Therefore $n > 70$

Take $n = 140$, $c \geq 20a$

$\therefore T(n) = T\left[\frac{n}{5}\right] + T\left(\frac{7n}{10}+6\right) + O(n)$ for $n \geq 140$

$T(n) = 1$ for $n < 140$

An app

Rankin

Collat

Inver

if

so

Here

Eas

How

→ use

Div

Q Consi

an

such

there

and c

Scanned by CamScanner

An application - counting the no. of inversions

Ranking Application

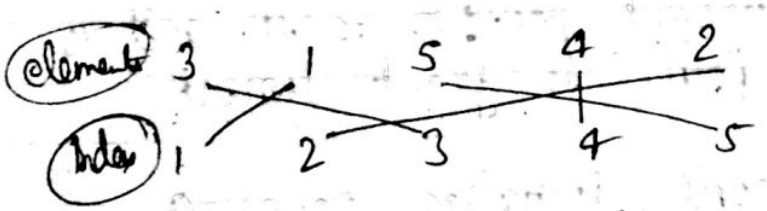
Collaborative filtering -

	my	other
A	1	3
B	2	1
C	3	5
D	4	4
E	5	2

ranking of movies

Inversion - For 2 indices $i<j$, it is called inversion if $A[i] > A[j]$

so other's inversions are
(3,1), (3,2), (5,4), (5,2), (4,2)



Here wherever crossing occurs, that is an inversions

Easy solution - $O(n^2)$
How to do it in $O(n \log n)$?
→ use divide and conquer

Divide the input into 2 subarrays of size $n/2$.

Q Consider an array $a_1 \ldots a_n$. A pair (a_i, a_j) is called an inversion if $a_i > a_j$ for two indices i and j such that $i<j$. For example in the sequence (2,4,1,3,5) there are 3 inversions (2,1), (4,1) and (4,3). Design a divide and conquer based algorithm to count the no. of inversions

Scanned by CamScanner

```
Sort-and-count(L)
{
  if |L| == 1 then there is no inversion //return
  else
     Divide the list into 2 halves, say A & B
     (r_A, A) = sort-and-count(A)
     (r_B, A) = sort-and-count(B)
     (r, L) = Merge-and-count(A, B)
  end if
  Return r = r_A + r_B + r and sorted list L
}
```

```
Merge-and-count(A, B)
{
  Maintain a current pointer to A and B
  initialized to point to front element.

  Maintain a count, initialize count = 0;
  while (A & B are not empty)
  {
     Let a_i and b_j be the elements pointed
     by current pointer..

     Append the smaller of these to the
     output list
     If b_j is smaller then
        increment count by the no. of elements
        remaining in A.
     end if
     Advance current pointer in the list from
     which smallest is selected.
```

Scanned by CamScanner

once one list is empty - append the remainder of the other to the output list.
Return count and merge list.
}

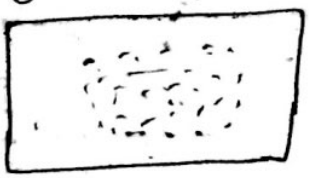
Recurrence:-

$$T(n) = \begin{cases} 2T(n/2) + a(n) & \text{for } n \geq 2 \\ 1 & \text{for } n < 2 \end{cases}$$

$$T(n) = O(n \log n)$$

Some problems in divide and conquer

(i) Closest pair of points

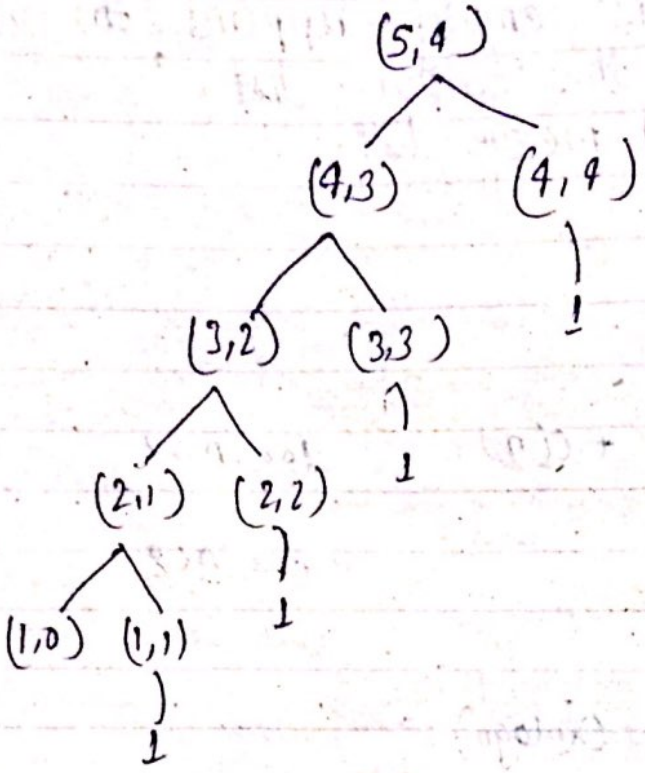


(ii) Selection (i^{th} smallest)

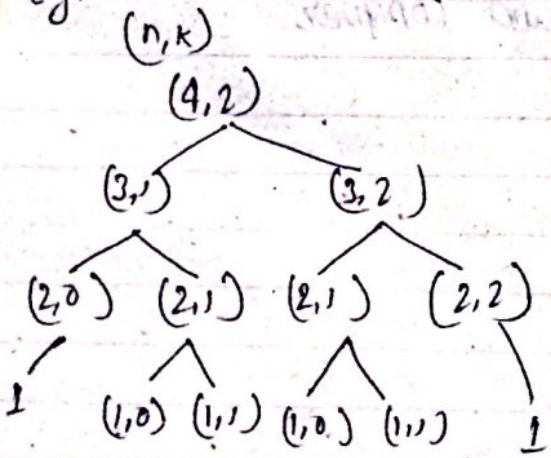
Binomial coefficient

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = n \text{ or } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & \text{otherwise} \end{cases}$$

Scanned by CamScanner



Eg:-



	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	

```
Bin (n,k)
{
    if (k==n) || (k==0)
        return 1
    else
        return (Bin(n-1,k-1) + Bin(n-1,k));
}
```

Scanned by CamScanner

Dynamic Programming

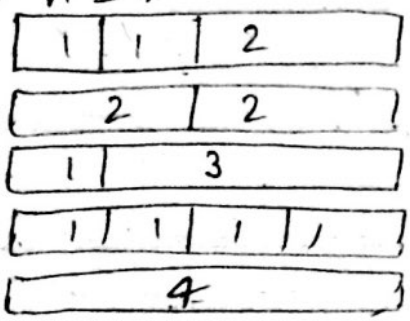
Eg:- Rod cutting problem

Problem statement :- Given a rod of length n inches, and a table of prices p_i for $i = 1, \ldots n$, determine the maximum revenue r_n obtainable by cutting up the rods and selling the prices

Eg:-

length (i)	1	2	3	4	5	6	7	8	9	10
price (p_i)	1	5	8	9	10	17	17	20	24	30

Let $n = 4$



$1+1+5=7$

$5+5=10$ → Maximum revenue

$1+8=9$

$1+1+1+1=4$

9.

$r_n = r_4 = 10$

$r_n = p_{i_1} + p_{i_2} + \cdots p_{i_K}$, where $i_1, i_2, \ldots i_K$ are the lengths of pieces

$r_4 = p_2 + p_2 = 5+5 = 10$

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_{n-1} + r_1\right)$$
$$r_n = \max\left(p_i + r_{n-j}\right) \quad 1 \le j \le n$$

$$\begin{cases} r_4 = \max(p_4, r_1+r_3, r_2+r_2, r_3+r_1) \\ r_3 = \max(p_3, r_1+r_2, r_2+r_1) \\ r_2 = \max(p_2, r_1+r_1) \\ r_1 = \max(p_1) = p_1 \end{cases}$$

$r_0 = 0$

$r_4 = \max(p_1+r_3, p_2+r_2, p_3+r_1, p+r_0)$

Scanned by CamScanner

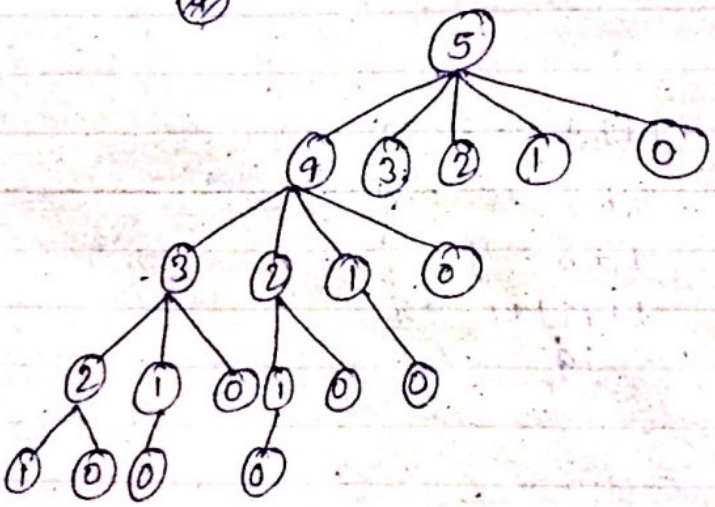
Recursive solution :-

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Recursive top-down implementation

```
cut_rod (p, n)
{
  if (n == 0)   return 0
  q = -∞
  for i = 1 to n
        q = max (q, p[i] + cut_rod (p, n-i));
  return q;
}
```

(4)



$$T(n) = 1 + \sum_{j=1}^{n-1} T(j)$$

Prove that $T(n) = 2^n$

Memoization -
- recursive solution with look up table
- top down approach

Using dynamic programming for optimal rod cutting
① top-down with memoization
bottom-up method

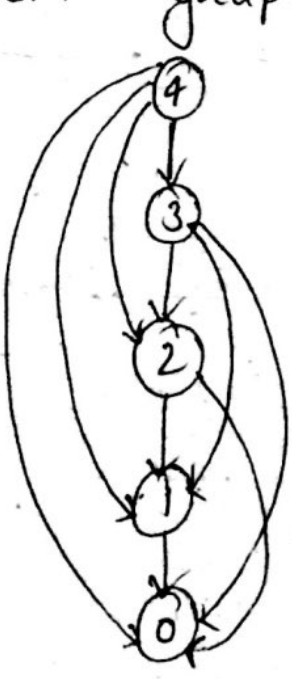
Scanned by CamScanner

① top-down CUT-ROD procedure with memoization

```
Memoized-cut-rod (p,n)
  { Let r[0...n] be a new array
    for (i = 0 to n)
        r[i] = -∞
    return Aux-memoized-cut-rod (p, n, r)
  }

Aux-memoized-cut-rod (p,n,r)                    O(n²)
  {
    if (r[n] ≥ 0)  return r[n];
    if (n == 0)    q = 0;
    else { q = -∞
           for (i = 1 to n)
                q = max (q, p[i] + Aux-memoized-cut-rod
                                   (p, n-i, r))
         }
    r[n] = q;
    return q;
  }
```

Subproblem graph



Scanned by CamScanner

② bottom-up method

```
Button_up_cut_rod(p,n)
  {
    define r[0...n] as new array
    r[0] = 0;
    for j = 1 to n
        q = +∞
        for i = 1 to j
            if (q < p[i] + r[j-i])
                q = p[i] + r[j-i];
        r[j] = q;
    return r[n]
  }
```

O(n²)

r[j] is the value of optimal solution

Bottom-up-cut-Rod-solution (p,10)

How to construct the optimal solution ?

r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

```
Bottom_up_cut_Rod_solution (p,n)
  { Let r[0...n] and s[0...n] be new array
    r[0] = 0;
    for j = 1 to n;
        q = -∞;
        for i = 1 to j
            if (q < p[i] + r[j-i])
              {
                q = p[i] + r[j-i];
                s[j] = i;
              }
        r[j] = q;
    return s and r;
```

Scanned by CamScanner

A call to cut-rod-solution (p, n) would print the cuts 1 and 6 corresponding to the first optimal decomposition for 8_2

call with n = 2

```
cut-rod-solution (p, n)
  {
    (r, s) = Bottom-up-cut-Rod-extended (p, n);
     while (n > 0)
        {
           print s[n];
           n = n - s[n];
        }
  }
```

Steps
(i) Indentify the optimal substructure
(ii) (a) Recursive formulation
(b) Recursive top-down solution with memorization
(iii) Bottom up solution (subproblem graph)
(iv) Construct the optimal solution

Steps in Dynamic programming
1. Identify the optimal substructure
2. Formulate Recursive solution
3. Establish a top-down / memorization based solution to get the value of optimal solution
4. Construct the optimal solution

Q Differentiate between the divide and conquer and Dynamic problem techniques for solving problems.
A

Scanned by CamScanner

Matrix chain Multiplication

$A_1, \ldots A_n$

A_1, A_2, A_3

$10 \times 5, \quad 5 \times 20, \quad 20 \times 12$

$10 \times 20 \qquad 20 \times 12$

$A_1A_2A_3 \qquad 10 \times 12$

Obtain a parenthesization of $A_1, \ldots A_n$ that results in minimum number of scalar multiplications.

$((A_1A_2)A_3)$

$(A_1(A_2A_3))$

Eg: A_1, A_2, A_3, A_4

1. $((A_1A_2)(A_3A_4))$
2. $((A_1(A_2A_3))A_4)$
3. $(((A_1A_2)A_3)A_4)$
4. $(A_1((A_2A_3)A_4))$
5. $(A_1(A_2(A_3A_4)))$

$P(1) = 1$

$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k)$

$P(2) = \sum P(1) P(1) = 1$

$P(3) = P(1) P(2) - P(2) P(1)$

$= 1 + 1 = 2$

$P(4) = P(1) P(3)$

$= 2$

Scanned by CamScanner

A_1, A_2, A_3

$10 \times 5, 5 \times 15, 15 \times 4$

$((A_1 A_2) A_3)$ → No. of scalar multiplications.

$10 \times 5 \times 15$ + cost of multiplying $(10 \times 15)(15 \times 4)$

$= 10 \times 5 \times 15 + 10 \times 15 \times 4$

$= 750 + 600 = 1350$

$(A_1 (A_2 A_3))$ -

$5 \times 15 \times 4 + 10 \times 5 \times 4$

$= 300 + 200 = 500$

$A_1 \ldots A_j = (A_1 \ldots A_k)(A_{k+1} \ldots A_j)$

$i \times k \quad (p_{j-1} p_k)(p_k \times p_j)$

The parenthesization of prefix chain $(A_j \to A_k)$ within an optimal parenthisization of $A_j \ldots A_j$ is also optimal

Recursive solution

$m[i,j]$ represents cost of multiplying chain $A_j, A_{j+1}, \ldots A_j$.

$m[i,i] = 0$

$m[i,j]$ is defined when $i<j$

$$m[i,j] = m[i,k] + m[k+1, j] + p_{j-1} p_k p_j$$

$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} m(i,k) + m(k+1,j) + p_{j-1} p_k p_j, & \text{if } i<j \\ 0 & \text{if } i=j \end{cases}$$

	1	2	3	4
1	0	—	—	—
2		0	—	—
3			0	—
4				0

Scanned by CamScanner

Q Find optimal parenthesization for the product A_1, A_2, A_3 Given $p = \langle 10, 3, 20, 15 \rangle$

m

	1	2	3
1	0		
2		0	
3			0

s

	1	2
1	1	
2	2	
3		

$m[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2 = 600$

$m[2,3] = m[2,2] + m[3,3] + p_1 p_2 p_3 = 900$

$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{cases}$

$= \min \begin{cases} 0 + 900 + (10 \cdot 3 \cdot 15) \\ 600 + 0 + (10 \cdot 20 \cdot 15) \end{cases}$

$= \min (4500 + 900, 600 + 3000)$

$= \min (1350, 3600)$

$= 1350$

$(A_1 (A_2 A_3))$

Scanned by CamScanner

Subsequence

A subsequence of a sequence / string $X = \langle x_1, x_2, \ldots x_m \rangle$ is a sequence obtained by deleting 0 or more elements from X.

Example:
"Sudan" is a subsequence of "sesquipedalian".
So is "equal".

There are 2^m subsequences of X

A common subsequence Z of two sequences X and Y is a subsequence of both.

Example: "ua" is a common subsequence of "sudan" and "equal".

Longest Common Subsequence

Input: $X = \langle x_1, x_2, \ldots x_m \rangle$
$Y = \langle y_1, y_2, \ldots y_n \rangle$

Output: a longest common subsequence (LCS) of X and Y

Example.

(a) X = abcbdab Y = bdcaba
LCS_1 = bcba LCS_2 = bdab

(b) X = enquiring Y = sequipedalian
LCS = equin

(c) X = empty bottle Y = nematode knowledge
LCS = emt ole

Scanned by CamScanner

Brute-force Algorithm

- For every subsequence of X, check if it's a subsequence of Y.

($2^n m$)

Worst-case running time: $(n2)^m$.

- 2^m subsequences of X to check
- Each check takes $O(n)$ time - scanning Y for first element, then from there for next element

Optimal Substructure of LCS

X: x_1 | x_2 | ... | x_m

Y: y_1 | y_2 | ... | y_n

Q. Explain the princip of optimality in LCS problem.

LCS 2: z_1 | z_2 | ... | z_k | x_m

Case 1: $x_m = y_n$

Then $z_k = x_m = y_n$

Otherwise, LCS has length $\geq k+1$, a contradictio

Case 2: $x_m \neq y_n$

X: x_1 | x_2 | ... | x_m

Y: y_1 | y_2 | ... | y_n

Either LCS of

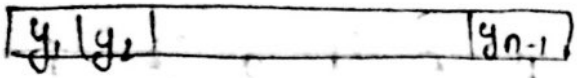
x_1 | x_2 | ... | x_{m-1}

y_1 | y_2 | ... | y_n

or LCS of

x_1 | x_2 | ... | x_m

Scanned by CamScanner



A Recursive formula

$X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$

$c[i,j]$ = length of an LCS of $\langle x_1, \ldots x_i \rangle$ and $\langle y_1, \ldots y_j \rangle$

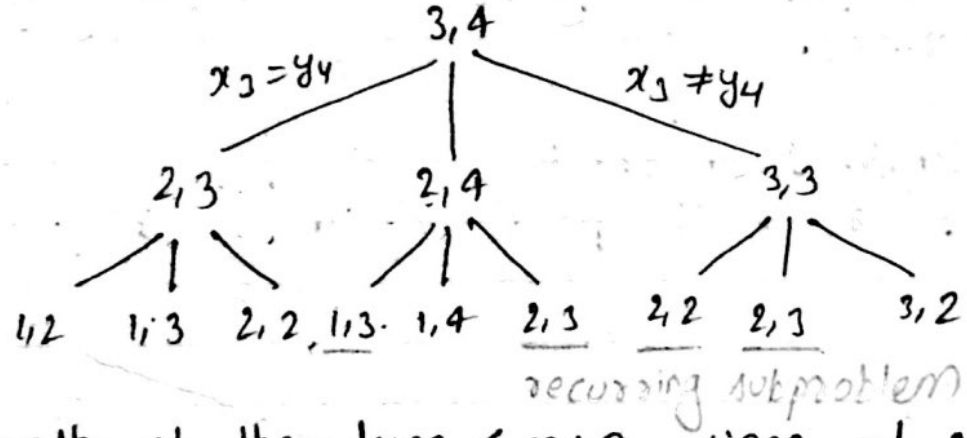
Then $c[m,n]$ = length of an LCS of X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c(i-1, j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

A total of $(m+1)(n+1)$ subproblems.

Recursion Tree

$m=3$ and $n=4$.



Depth of the tree $\leq m+n$ since at each level i and/or j reduce by 1

Branches by at most 3 at each node.

Amount of work for top-down recursion: $3^{\Theta(m+n)}$

Constructing an LCS

1. Initialize entries $c(i,0)$ and $c(0,j)$

Scanned by CamScanner

C[,]		s	e	s	q	u	i	p	e	d	a	l	i	a	n
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	0	0													
n	0														
q	0														
u	0														
i	0														
r	0														
i	0														
n	0														
g	0														

m=9

n=14

$c[1,2] = 1 \rightarrow c[2,1] = 1$ $\quad c[1,2] = \max(c[1,1], c[2,1]) = 1$

2. Fill out entries row by row. Save pointers in $b[1 \ldots m, 1 \ldots n]$.

LCS length determined

C[,]		s	e	s	q	u	i	p	e	d	a	l	i	a	n
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
n	0	0	1	1	1	1	1	1	1	1	1	1	1	1	2
q	0	0	1	2	2	2	2	2	2	2	2	2	2	2	2
u	0	0	1	1	2	3	3	3	3	3	3	3	3	3	3
i	0	0	1	1	2	3	4	4	4	4	4	4	4	4	4
r	0	0	1	1	2	3	4	4	4	4	4	4	4	4	4
i	0	0	1	1	2	3	4	4	4	4	4	4	5	5	5
n	0	0	1	1	2	3	4	4	4	4	4	4	5	5	6
g	0	0	1	1	2	3	4	4	4	4	4	4	5	6	6

LCS has length 6

Scanned by CamScanner

Reconstructing an LCS

3. Starting from the lower-right corner, follow the pointers in b[,]. Whenever encounters an upper-left pointer, print the labeling char.

		s	e	s	q	u	i	p	e	d	a	l	i	a	n	Prin
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
e	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	"e"
n	0	0	1	1	1	1	1	1	1	1	1	1	1	1	2	
q	0	0	1	1	2	2	2	2	2	2	2	2	2	2	2	"q"
u	0	0	1	1	2	3	3	3	3	3	3	3	3	3	3	"u"
i	0	0	1	1	2	3	4	4	4	4	4	4	4	4	4	"i"
r	0	0	1	1	2	3	4	4	4	4	4	4	4	4	4	
i	0	0	1	1	2	3	4	4	4	4	4	4	5	5	5	"i"
n	0	0	1	1	2	3	4	4	4	4	4	4	5	5	6	"n"
g	0	0	1	1	2	3	4	4	4	4	4	4	5	5	6	

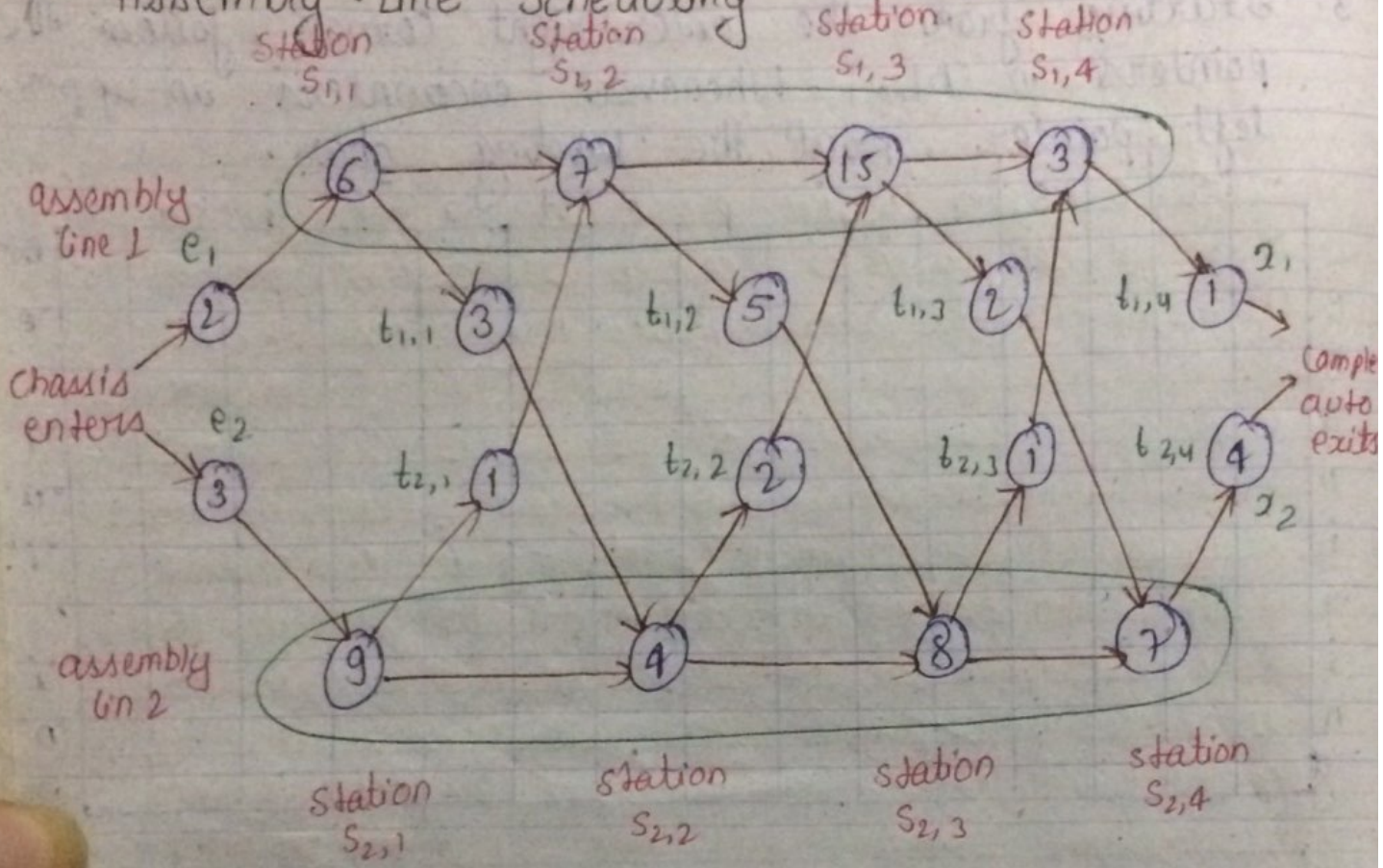
Q Obtain the LCS of $X = \langle b\,c\,b\,d\,a\,b\,c\,c\,a \rangle$, $Y = \langle b\,a\,b\,a\,c\,a \rangle$
Construct the subproblem and show the length of LCS

Soln:-

		y	b	a	b	a	c	a	
		0	1	2	3	4	5	6	
x	0	0	0	0	0	0	0	0	
b	1	0	1	↖ 1	1	← 1	← 1	← 1	
c	2	0	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2	← 2	
b	3	0	↖ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2	"b"
d	4	0	↑ 1	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	
a	5	0	↑ 1	↖ 2	↑ 2	↖ 3	↑ 3	↖ 3	"a"
b	6	0	↑ 1	↑ 2	↖ 3	↑ 3	↑ 3	↑ 3	"b"
c	7	0	↑ 1	↑ 2	↑ 3	↑ 3	↖ 4	← 4	
c	8	0	↑ 1	↑ 2	↑ 3	↑ 3	← 4	↑ 4	"c"
a	9	0	↑ 1	↑ 2	↑ 3	↑ 4	↑ 4	↖ 5	"a"

Scanned by CamScanner

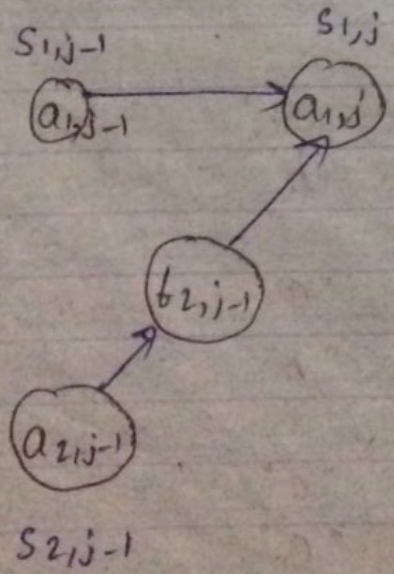
Assembly-Line Scheduling



Minimize the total time through the factory for o
auto

Optimal substructure

$f_i(j)$: the fastest possible time to get a chassis from the starting point through station $S_{i,j}$

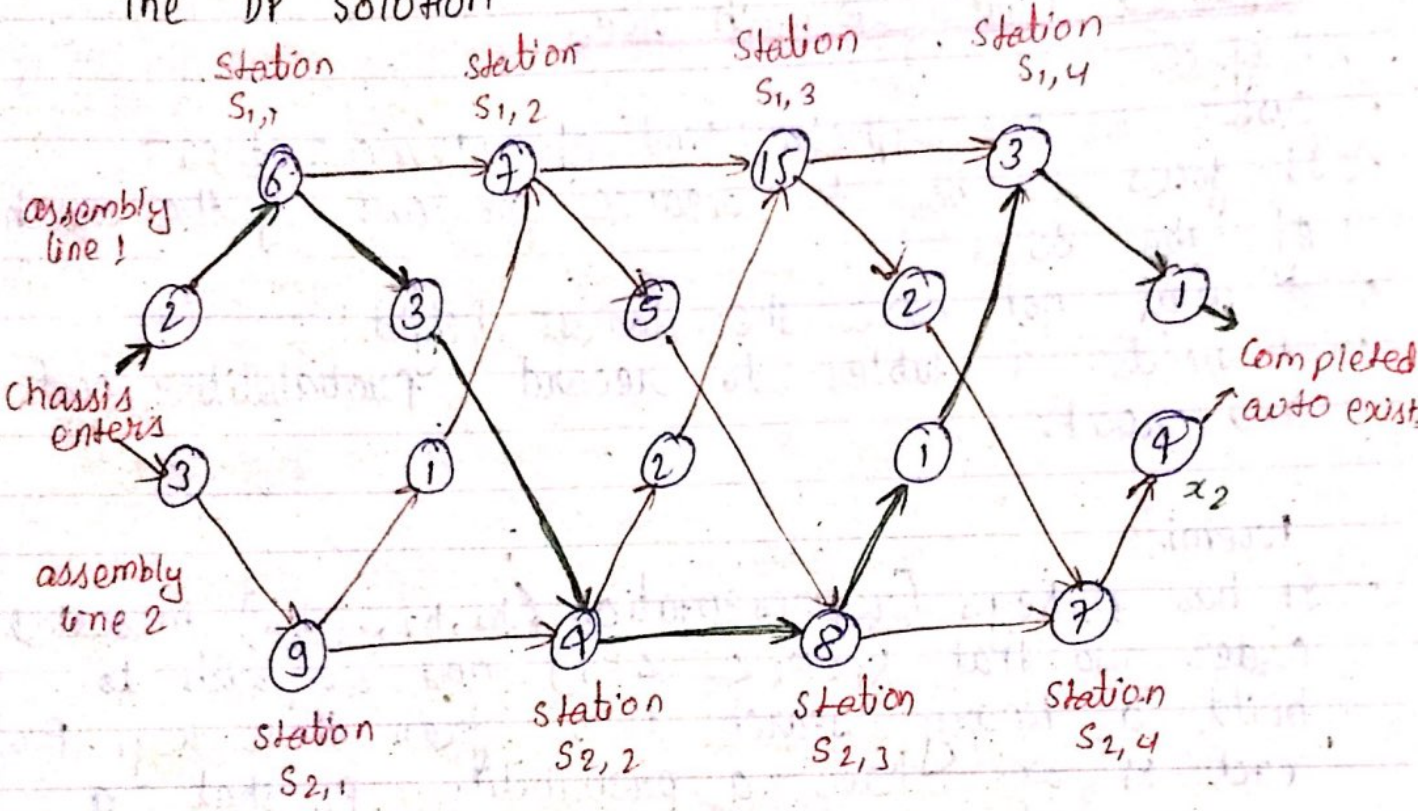


$$f_1(j) = \begin{cases} e_1 + a_{1,1} & \text{, if } j = 1 \\ \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + \right. & \\ \left. \text{, if } j > \right. & \end{cases}$$

$$f^* = \min\left[f_1[n] + x_1,\ f_2[n] + x_2\right]$$

Scanned by CamScanner

The DP solution



j	1	2	3	4
$f_1[j]$	8	15	30	27
$f_2[j]$	12	15	23	30

j	2	3	4	
$l_1[j]$	1	1	2	← from line 2
$l_2[j]$	1	2	2	

$f^* = 28$

Optimal substructure of an LCS

Let $X = \{x_1, x_2, x_3, \ldots x_m\}$ and $Y = \{y_1, y_2, y_3, \ldots y_n\}$ be sequences, and let $Z = \{z_1, z_2, z_3, \ldots z_k\}$ be any LCS of X and Y

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Scanned by CamScanner

Optimal Binary Search Tree.

Preface

- OBST is one special kind of advanced tree.
- It focus on how to reduce the cost of the search of the BST.
- It may not have the lowest height!
- It needs 3 tables to record probabilities, cost and root.

Premise

- It has n keys (representation $(k_1, k_2, \ldots k_n)$ in sorted order (so that $k_1 < k_2 < \ldots < k_n$), and we wish to build a binary search tree from these keys. For each k_i, we have a probability p_i that a search will be for k_i.
- In contrast of, some searches may be for values not in k_i, and so we also have $n+1$ "dummy keys" $d_0, d_1, \ldots, d_n$ representating not in k_i.
- In particular, d_0 represents all values less than k_1, and d_n represents all values greater than k_n, and for $i = 1, 2, \ldots n-1$, the dummy key d_i represents all values between k_i and k_{i+1}.

The dummy keys are leaves (external nodes) and the data keys mean internal nodes.

Formula & Prove

- The case of search are two situations, one is success, and the other, without saying, is failure
- We can get the first statement:

$$(i = 1 \sim n) \sum p_i + (i = 0 \sim n) \sum q_i = 1$$

Scanned by CamScanner

- Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T. Let us assume that the actual cost of a search is the number of nodes examined, i.e. the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is : (The second statement)
- E[search cost in T]

$$= \sum_{i=1}^{n} p_i \cdot (\text{depth}_T(k_i) + 1) + \sum_{i=0}^{n} q_i \cdot (\text{depth}_T(d_i) + 1)$$

$$= 1 + \sum_{i=1}^{n} p_i \cdot \text{depth}_T(k_i) + \sum_{i=0}^{n} q_i \cdot \text{depth}_T(d_i)$$

where depth_T denotes a node's depth in the tree T.

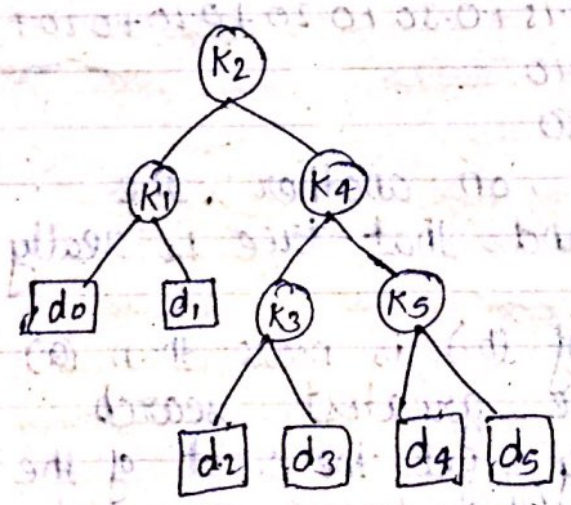


Figure (a)

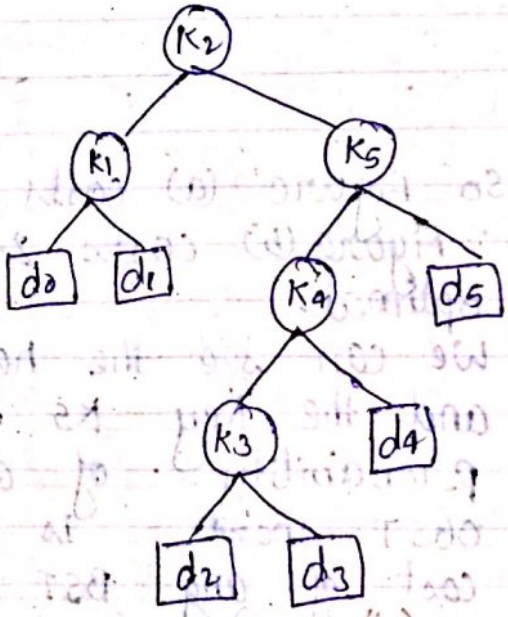


Figure (b)

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Scanned by CamScanner

By figure (a), we can calculate the expected search cost node by node:

Node #	Depth	probability	cost
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	3	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40

Cost = Probability * (Depth + 1)

- And the total cost = 0.30 + 0.10 + 0.15 + 0.20 + 0.60 + 0.15 + 0.30 + 0.20 + 0.20 + 0.20 + 0.40
  = 2.80
- So figure (a) costs 2.80, on another, the figure (b) costs 2.75 and that tree is really optimal.
- We can see the height of (b) is more than (a) and the key k_5 has the greatest search probability of an key, yet the root of the OBST shown is k_2. (The lowest expected cost of any BST with k_5 at the root is 2.85)

Step 1: The structure of an OBST

- To characterize the optimal substructure of OBST, we start with an observation about subtrees. Consider any subtree of a BST. It must contain

Keys
some
contain
leaves

- We ne
show
to th
subpr
these
of
these
will
dum
subt
the
exa
1 ≤
sea
tho
that

- Ther
sub
k_i,
arg
k_i,
seq
to
key
dee
syn

Scanned by CamScanner

keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, \ldots k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$

- We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i, \ldots, k_j$, one of these keys, say k_r $(i \le r \le j)$, will be the root of an optimal subtree ~~of the root~~ containing these keys. The left subtree of the root k_r will contain the keys $(k_i, \ldots, k_{r-1})$ and the dummy keys $(d_{i-1}, \ldots, d_{r-1})$, and the right subtree will contain the keys $(k_{r+1}, \ldots, k_j)$ and the dummy keys $(d_r, \ldots, d_j)$. As long as we examine all candidate roots k_r, where $i \le r \le j$, and we determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, we are guaranteed that we will find an OBST.

- There is one detail worth nothing about "empty" subtrees. Suppose that in a subtree with keys $k_i, \ldots, k_j$, we select k_i as the root. By the above argument, k_i's left subtree contains the keys $k_i, \ldots k_{i-1}$. It is natural to interpret this sequence as containing no keys. It is easy to know that subtree also contain dummy keys. The sequence has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if we select k_j as the root,

Scanned by CamScanner

then k_j's right subtree contains the keys, $k_{j+1}, \ldots, k_j$; this right subtree contains no actual keys, but it does contain the dummy key d_j.

Step 2: A recursive solution

- We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an OBST containing the keys $k_i, \ldots, k_j$, where $j \geq 1$, $j \geq n$ and $j \geq i-1$. (It is when $j = i-1$ that ther are no actual keys; we have just the dummy key d_{i-1}.)
- Let us define $e[i,j]$ as the expected cost of searching an OBST containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1,n]$.
- The easy case occurs when $j = i-1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.
- When $j \geq i$, we need to select a root k_r from among $k_i, \ldots, k_j$ and then make an OBST with keys $k_i, \ldots, k_{r-1}$ its left subtree and an OBST with keys $k_{r+1}, \ldots, k_j$ its right subtree
- By the time, what happens to the expected search cost of a subtree when it becomes a subtree of a node? The answer is that the depth of each node in the subtree increases by 1.
- By the second statement, the excepted search cost of this subtree increases by the sum of all the probabilities in the subtree. For a

subtree with keys $k_i, \ldots, k_j$ let us denote this sum of probabilities as

$$w(i,j) = (l = i \text{ to } j) \sum p_l + (l = j-1 \text{ to } j) \sum q_l$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots k_j$, we have

$$E[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

Nothing that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$

- We rewrite $e[i,j]$ as.

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$$

The recursive equation as above assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$E[i,j] =$

case 1: if $i \le j$, $i \le r \le j$

$$e[i,j] = \min\{e[i,r-1] + e[r+1,j] + w(i,j)\}$$

case 2: if $j = i-1$

$$e[i,j] = q_{i-1}$$

- The $e[i,j]$ values give the expected search costs in OBST. To help us keep track of the structure of OBST, we define root$[i,j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an OBST containing keys $k_i, \ldots, k_j$.

Scanned by CamScanner

Step 3: Computing the expected search cost of an OBST

- We store the $e[i,j]$ values in a table. $e[1..n+1, 0...n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n, we will need to compute and store $e[n+1,n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, we will need to compute and store $e[1,0]$. We will use only the entries $e[i,j]$ for which $j \geq i-1$. We also use a table $root[i,j]$, for recording the root of the subtree containing keys $k_i, \ldots, k_j$. This table uses only the entries for which $1 \leq i \leq j \leq n$.

- We will need one other table for efficiency. Rather than compute the value of $w(i,j)$ from scratch every time we are computing $e[i,j]$ — we store these values in a table $w[1...n+1, 0...n]$. For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n$.

- For $j \geq i$ we compute:
  $w[i,j] = w[i,j-1] + p_j + q_j$

OPTIMAL-BST(p, q, n)

Scanned by CamScanner

```
OPTIMAL-BST (p, q, n)
  For i ← 1 to n+1
      do e[i, i-1] ← q_{i-1}
      do w[i, i-1] ← q_{i-1}
  For l ← 1 to n
      do for i ← 1 to n-l+1
            do j ← i+l-1
            e[i, j] ← ∞
            w[i, j] ← w[i, j-1] + p_j + q_j
            For r ← i to j
                  do t ← e[i, r-1] + e[r+1, j] + w[i, j]
                     if t < e[i, j]
                     then e[i, j] ← t
                     root[i, j] ← r
  Return e and root
```

e

	5	4	3	2	1
6					
5	0.10				
4	0.05	0.05			
3	0.05	0.05	0.05		
2	0.10	0.30	0.25	0.40	
1	0.50	0.30	0.60	0.70	0.90
0	0.90	1.30	1.20	1.75	2.75

w

	5	4	3	2	1
6					
5	0.10				
4	0.05	0.05			
3	0.05	0.05	0.05		
2	0.10	0.20	0.15	0.25	
1	0.35	0.30	0.35	0.45	0.70
0	0.50	0.60	0.50	0.55	0.80	1.00

root

	5	4	3	2	1
5	5				
4	5	5			
3	4	5	5		
2	2	2	4	4	
1	2	2	2	3	3

The table e[i,j], w[i,j] and root[i,j] computed by optimal-BST

Scanned by CamScanner

Advanced Proof-1

- All keys (including data keys and dummy keys) of the weight sum (probability weight) and that can get the formula:
- $\sum_{i=1}^{n} k_i + \sum_{i=0}^{n} d_i$
- Because the probability of k_i is p_i and d_i is q_i;
- Then rewrite that
- $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$ ..... formula (1)

Advanced Proof-2

- We first focus on the probability weight; but not in all, just for some part of the full tree. That means we have $k_i, \ldots, k_j$ data, and $1 \le i, \le j \le n$, and ensures that $k_i, \ldots, k_j$ is just one part of the full tree. By the time, we can rewrite formula (1) into
- $w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1} q_l$
- For recursive structure, maybe we can get another formula for $w[i,j] = w[i,j-1] + p_j + q_j$
- By this, we can struct the weight table

Advanced Proof-3

- Finally, we want to discuss our topic, without saying, the cost, which is expected to be the optimal one.

Scanned by CamScanner

- Then define the recorsive structore's cost e[i,j]. which means $k_i, \dots, k_j$, $1 \le i \le j \le n$ cost.
- And we can divide into root, left subtree and right subtree.

Advanced Proof -4

- The final cost formula:

$$e[i,j] = p_r + e[i, r-1] + w[i, r-1] + e[r+1, j] + w[r+1, j]$$

Nothing that: $p_r + w[i, r-1] + w[r+1, j] = w[i,j]$

so, $e[i,j] = (e[i, r-1] + e[r+1, j]) + w[i,j]$

↑ Get the minimal set

And we use its to struct the cost table!

P.S. Nether weight nor cost calculating, if $k_i, \dots k_j$, but $j = i-1$, it means that the sequence have no actual key, but a dummy key.

Exercise

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Scanned by CamScanner

0-1 Knapsack problem

Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

Item #	Weight	Value
1	4	8
2	3	6
3	5	5

There are two versions of the problem

1. 0-1 knapsack problem
   Items are indivisible, you either take an item or not. Some special instances can be solved with dynamic programming.

2. Fractional knapsack problem
   Items are divisible; you can take any fraction of an item.

0-1 knapsack problem

- Given a knapsack with maximum capacity W and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)

Problem: How to pack the knapsack to achieve maximum total value of packed items?

Scanned by CamScanner

Problem, in other words, is to find
$\max \sum_{i \in T} b_i$ subject to $\sum_{i \in T} w_i \leq W$

The problem is called a 0-1 problem, because each item must be entirely accepted or rejected.

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 maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

dynamic programming approach

- we can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)

Problem:

How to pack the knapsack to achieve maximum total value of packed items?

we can do better with an algorithm based on

Scanned by CamScanner

dynamic programming.

- We need to carefully identify the subproblems
  Let's try this:
  If items are labeled 1...n, then a subproblem would be to find an optimal solution for
  $S_k = \{$items labeled $1, 2, \ldots k\}$

- If items are labeled 1..n, then a subproblem would be to find an optimal solution for
  $S_K = \{$items labeled $1, 2, \ldots, K\}$

- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_K)?
- Unfortunately, we can't do that

$w_1=2$	$w_2=4$	$w_3=5$	$w_4=3$	
$b_1=3$	$b_2=5$	$b_3=8$	$b_4=4$	

Max weight: $W=20$
For S_4:
Total weight: 14
Maximum benefit: 20

	Item #	Weight w_i	Benefit b_i
	1	2	3
	2	4	5
S_4	3	5	8
S_5	4	3	4
	5	9	10

$w_1=2$	$w_2=4$	$w_3=5$	$w_5=9$
$b_1=3$	$b_2=5$	$b_3=8$	$b_5=10$

For S_5:
Total weight: 20
Maximum benefit: 26

Solution for S_4 is not part of the solution for S_5!!!

Scanned by CamScanner

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)

Problem:
How to pack the knapsack to achieve maximum total value of packed item?

- Let's add another parameter: w, which will represent the maximum weight for each subset of items.

- The subproblem then will be to compute V[k,w], i.e, to find an optimal solution for S_k = {items labeled 1, 2, ... k} in a knapsack of size w.

Resursive formula for subproblems
Assuming knowing V[i,j], where $j = 0, 1, 2, \ldots k-1$, $j = 0, 1, 2, \ldots w$, how to derive V[k,w]?

Recursive formula for subproblems:

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

It means that the best subset of S_k that has total weight w is:
1) the best subset of S_{k-1} that has total weight $\leq w$,

Scanned by CamScanner

2) the best subset of S_{k-1} that has total weight $\leq w$ plus the item k

Recursive formula

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

- The best subset of S_k that has the total weight $\leq w$ either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value.

0-1 knapsack Algorithm

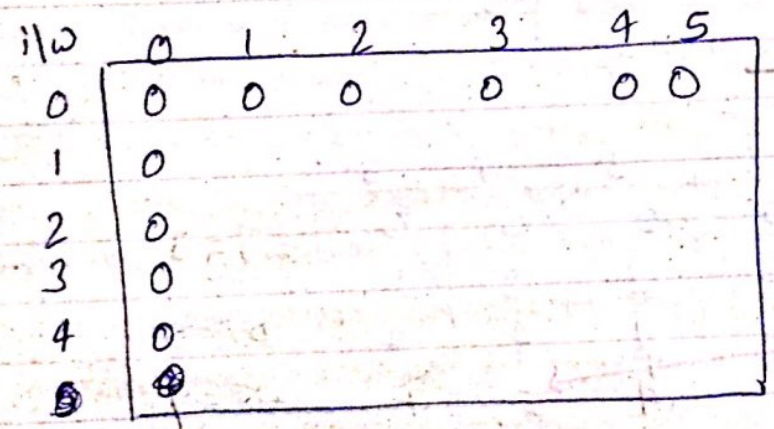
```
for w=0 to W
    V[0,w] = 0                          O(W)
for i=1 to n
    V[i,0] = 0
for i=1 to n                            Repeat n times
    for w=0 to W                        O(W)
        if wi <= w  // item i can be part of the solution
            if{bi + V[i-1, w-wi] > V[i-1,w]}
                V[i,w] = bi + V[i-1, w-wi]
            else
                V[i,w] = V[i-1, w]
        else
            V[i,w] = V[i-1,w]   // wi > w
```

Scanned by CamScanner

What is the running time of this algorithm?
$O(n*W)$

Remember that the brute-force algorithm takes $O(2^n)$

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

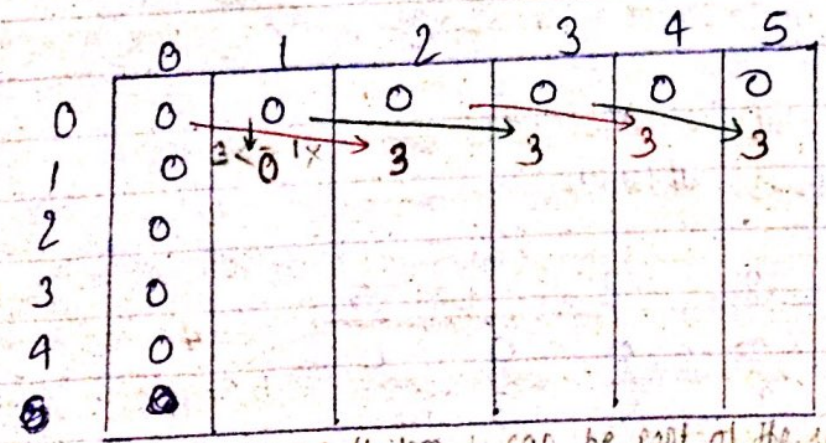


for $w=0$ to W
$V[0,w]=0$

for $i=1$ to n
$V[i,0]=0$

w_i b_i
w_i (weight) value

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



$i=1$

$b_i=3$
$w_i=2$
$w=1\ 2\ 3\ 4\ 5$
$w-w_i=-1\ 0\ 1\ 2\ 3$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + V[i-1, w-w_i] > V[i-1, w]$
    $V[i,w] = b_i + V[i-1, w-w_i]$
  else $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Scanned by CamScanner

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					
5	0					

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

$i = 2$

$b_i = 4$

$w_i = 3$

$w = 1\ 2\ 3\ 4\ 5$

$w - w_i = -2\ -1\ 0\ 1\ 2$

if $w_i <= w$.
  if $b_i + V[i-1, w-w_i] > V[i-1, w]$
⑤ ④ ③ $V[i,w] = b_i + V[i-1, w-w_i]$
  else
    $V[i,w] = V[i-1,w]$
② ① else $V[i,w] = V[i-1,w]$

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					
5	0					

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

$i = 3$

$b_i = 5$

$w_i = 4$

$w = 1\ 2\ 3\ 4\ 5$

$w - w_i = -3\ -2\ -1\ 0\ 1$

if $w_i <= w$
  if $b_i + V[i-1, w-w_i] > V[i-1, w]$
    ① $V[i,w] = b_i + V[i-1, w-w_i]$
  else
    ⑤ $V[i,w] = V[i-1,w]$
③ ② ③ else $V[i,w] = V[i-1,w]$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

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

$i = 4$

$b_i = 6$

$w_i = 5$

$w = 1\ 2\ 3\ 4\ 5$

$w - w_i = -4\ -3\ -2\ -1\ 0$

Scanned by CamScanner

If $w_i <= w$
  if $b_i + V[i-1, w-w_i] > V[i-1, w]$
    $V[i,w] = b_i + V[i-1, w-w_i]$
  else
    (5) $V[i,w] = V[i-1,w]$
(4) (3) (2) (1) else $V[i,w] = V[i-1,w]$

Exercise

Q Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

, capacity $W = 6$

How to find out which items are in the optimal subset?

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	25	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

5 3
5 1

(5) (3)
5 + 1 = 6
50 + 15 = 65

Scanned by CamScanner

This algorithm only finds the max possible value that can be carried in the knapsack
i.e the value in V[n,W]
To know the items that make this maximum value an addition to this algorithm is necessary

How to find actual knapsack items

- V[n,W] is the maximal value of items that can be placed in the knapsack.
- Let i=n and k=W
  - if V[i,k] ≠ V[i-1,k] then
    mark the ith item as in the knapsack
    i = i-1, k = k-wi
  - else
    i = i-1 // Assume the ith item is not in the knapsack // could it be in the optimally packed knapsack?

Finding the item

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

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

	②	③	④	⑤
	①	③	④	⑤
i=	4	3	2	1
k=	5	5	5	2
bi=	6	5	4	3
wi=	5	4	3	2
V[i,k]=	7	7	7	3
V[i-1,k]=	7	7	3	0
k-wi =			2	0

① i=n, k=W
while i,k>0
  if V[i,k] ≠ V[i-1,k] then
  ⑤ ⑦ mark the ith item as in the knapsack
    i = i-1, k = k-wi
  else
  ③ ① i = i-1

i = 2-1 ; k = 5-3 = 2

Scanned by CamScanner

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
5	0	0	3			

$i=0$
$K=0$

The optimal knapsack should contain {1,2}

$i=n, K=W$

(5) while $i,K>0$
  if $V(i,K) \neq V(i-1,K)$ then
    mark the n^{th} item as in the knapsack
    $i=i-1, K=K-w_i$
  else
    $i=i-1$

Memorization (Memory Function Method)

- Solve only subproblems that are necessary and solve it only once.
- Memorization is another way to deal with overlapping subproblems in dynamic programming.
- With memorization, we implement the algorithm recursively
  - → If we encounter a new subproblem, we compute and store the solution.
  - → If we encounter a subproblem we have seen, we look up the answer.
- Most useful when the algorithm is easiest to implement recursively
  - → Especially if we do not need solutions to all subproblems.

Scanned by CamScanner

0-1 Knapsack Memory Function Algorithm

```
for i=1 to n
    for ω=1 to W
        V[i,ω] = -1

for ω=0 to W
    V[0,ω] = 0
for i=1 to n
    V[i,0] = 0

MFknapsack(i,ω)
    if V[i,ω] < 0
        if ω < ωi
            value = MFknapsack(i-1,ω)
        else
            value = max(MFknapsack(i-1,ω),
                        bi + MFknapsack(i-1, ω-ωi))
    V[i,ω] = value
    return V[i,ω]
```

0-1 knapsack problem : $O(\omega \times n)$ vs $O(2^n)$

Scanned by CamScanner

Brute - Force Approach

```
public static int MCSS(int [] a)
{
    int max=0 , sum=0 , start=0 , end=0;
    // Cycle through all possible values of start and end
       indexes
    // for the sum
    for (i=0 ; i< a.length ; i++)
    {
        for (j=i ; j < a.length ; j++)
        {
            sum=0;
            //Find sum A(i) to A(j)
            for (k=i ; k<=j ; k++)
                sum += a[k];
            if (sum > max)
            {
                max = sum;
                start = i; // Although method doesn't return these
                end = j; // they can be computed
            }
        }
    }
    return max;
}
```

Dynamic Programming Approach

```
SMaxV(0) = 0
MaxV(0) = 0
for i=1 to n
    SMaxV(i) = max(SmaxV(i-1) + x[i] , 0)
    MaxV(i) = max(MaxV(i-1), SMaxV(i))
return MaxV(n)
```

Scanned by CamScanner