

TABLE OF CONTENTS

TABLE OF CONTENTS.....	1
FIRST: DESIGN PROBLEMS [with solution]	2
Problem1: Task Scheduling.....	2
Problem2: Max Sequence Sum.....	3
Problem3: Highest Increase Pair.....	4
Problem4: Burn Image DVDs.....	5
Problem5: Coin Change.....	6
Problem6: Task Scheduling II	8
Problem7: Parents Gift.....	8
Problem8: FCIS Activities.....	9
Problem9: Restaurant Placement I	10
SECOND: DESIGN PROBLEMS [without solution]	12
Problem1: Office Organization	12
Problem2: Bread Transportation	13
Problem3: Candy & Sugar Rush	14
Problem4: Dividing Coins	15
Problem5: Mobile Base Stations.....	16
THIRD: OTHER QUESTIONS.....	17
QUESTION1: Huffman Code.....	17
QUESTION2: Knapsack Variation	18
QUESTION3: Activity Selection Variation	18

FIRST: DESIGN PROBLEMS [with solution]

Problem1: Task Scheduling

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Each task must run non-preemptively, that is, once task a_i is started, it must run continuously for p_i units of time.

Let c_i be the *completion time* of task a_i , that is, the time at which task a_i completes processing.

Your goal is to minimize the average completion time, that is, to minimize $\frac{1}{n} \sum_{i=1}^n c_i$.

For example, suppose there are two tasks, a_1 and a_2 , with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1 . Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$.

Give an algorithm that schedules the tasks so as to minimize the average completion time. State the running time of your algorithm?

SOLUTION:

Greedy-choice:

- Chose the process with minimum processing time and schedule it first.
- Since we want to minimize average completion time, $\frac{1}{n} \sum_{i=1}^n c_i$. Thus, we need to minimize completion time of each process c_i .
- Since $c_i = c_{i-1} + p_i$, where, c_{i-1} is completion time of previous process ($i-1$) and p_i is processing time of current process i .

To minimize c_i , we need to minimize both c_{i-1} and p_i .

But $c_{i-1} = c_{i-2} + p_{i-1}$, to minimize it, we need to minimize both c_{i-2} and p_{i-1} .

And $c_{i-2} = c_{i-3} + p_{i-2}$, to minimize it, we need to minimize both c_{i-3} and p_{i-2} .

...

And so on. Ending with $c_1 = c_0 + p_1 = 0 + p_1$

- This means to minimize c_1 , choose, among all processes (n), the process with **minimum processing time** (p_1).

Then to minimize c_2 , choose, among remaining processes ($n-1$), the process with **minimum processing time** (p_2).

And so on.

Sub-problem:

Schedule the remaining $(n-1)$ processes in optimal way (i.e. repeat step (i) again)

pseudo-code

1. Sort processes according to their **processingTime** in ascending order
(Merge/Quick)
2. Keep executing the processes in their order until finishing them

```
curTime = 0
totalCompletionTime = 0
for j = 1 to N {
    curTime = curTime + Processes[j].processingTime
    Processes[j].completionTime = curTime
    totalCompletionTime += curTime
}
Return totalCompletionTime / N
```

Deduce the complexity

1. Sort processes according to their **processingTime** in ascending order
→ $\Theta(N \log(N))$
 2. Keep executing the processes in their order until finishing them
→ $\Theta(N)$
- Total = → $\Theta(N \log(N))$

Problem2: Max Sequence Sum

Given the array of real numbers $x[n]$, compute the maximum sum found in any contiguous sub-array.

Example: If the input array is

31 -41 59 26 -53 58 97 -93 -23 84

 ↑ ↑

then the program returns the sum of $x[2..6] = 187$

Give an efficient algorithm to solve this problem? State the running time of your algorithm?

SOLUTION:

Greedy-choice:

- Start summing the elements from the beginning of the array
- If current sum is **+ve**, then continue (as there's possibility to increase this sum by adding further elements)

Else if current sum is **-ve**, then reset the sum to 0 and restart from next element (as starting from 0 is better (i.e. give us greater sum) than starting from -ve number)

Sub-problem:

- Continue the process of calculating current sum on the remaining elements.
- Keep track of the **maxSoFar** sum during your process.

Write the pseudo-code

```
maxSoFar = 0
curSum = 0
for i = 1 to n
{
    curSum += x[i]
    if curSum < 0 then
        curSum = 0

    maxSoFar = max(maxSoFar, curSum)
}
```

Deduce the complexity

⇒ **$O(N)$**

Extra Requirement

- **Modify** the above code to return the **start and end positions** of the found sequence?

Problem3: Highest Increase Pair

Suppose that we need to find in a given a sequence of numbers x_1, \dots, x_n the highest increase, i.e., some pair with $i < j$, $x_i < x_j$ and maximum difference $x_j - x_i$. A naïve algorithm can do that in $O(n^2)$ time. Give a more clever algorithm that needs only $O(n)$ time. Clearly explain the time bound.

SOLUTION:

Greedy-choice:

Always select the min of the two values as the first value (x_i), and then look for 2nd value (x_j) that leads to the max difference.

1. Let x_i = 1st item in the array
2. Check it with the next item
3. If $x_i <$ next item, then calculate their difference and update the **MaxSoFar** value
4. Else, set x_i to the next item

Sub-problem:

- Look for max difference in remaining items (each time: the number of items is reduced by 1)

Write the pseudo-code

```

MaxSoFar = 0
i = 0
For (j = 1 to N)
    If (A[i] > A[j]) Then
        i = j
    Else
        Diff ← A[j] - A[i]
        If (Diff > MaxSoFar) Then
            MaxSoFar = Diff
            1stItem = i
            2ndItem = j
        End If
    End If
End For

```

$\Theta(N)$

Deduce the complexity

$\Rightarrow \Theta(N)$

Problem4: Burn Image DVDs

Suppose there are a set of **N** bitmap images, all of the same size **S**. Each image has one of possible **R** ranks (1: min rank, R: max rank). There are **K** blank DVDs, each with a capacity **C**. It's desired to burn **each DVD** with the **most valuable images** (i.e. with **max total rank**). Following is an example:

Input	N = 9 Images, S = 2 GB, K = 2 DVDs, C = 4.5 GB									
	Image	1	2	3	4	5	6	7	8	9
	rank r[i]	3	2	5	1	2	2	2	4	3
Output	DVD#1: total rank = 9, Images are: 3, 8									
	DVD#2: total rank = 6, Images are: 1, 9									

Given:

- **N** images, each of size **S** and its own rank **r[i]**,
- **K** DVDs each with capacity **C**

Required:

- Calculate **max total rank** for **each of the K DVDs**, in efficient way?
1. What's the **Greedy choice** for this problem?
 2. Write the pseudo-code of an **efficient** solution to solve?
 3. What's the complexity of this efficient solution?

SOLUTION:

For each DVD, select the image with max rank

1. Sort the images according to their rank (descending)
2. Total[1...K] = 0
3. x = 1
4. NumPerDVD = floor(C / S)
5. For d = 1 to K
6. For i = 1 to NumPerDVD
 - a. Total[d] += r[x]
 - b. x++
7. Return Total

$\Theta(N \log(N))$

Problem5: Coin Change

Consider the problem of making change for **n** cents using the fewest number of coins. Assume that each coin's value is an integer.

Describe a greedy algorithm to make change consisting of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent)? What's the complexity your algorithm?

Explain when to use DP and when to use Greedy in Coin Change?

1. For each coin **c_i**,
 - i. Incrementally add its value to itself until the result just exceed the next coin **c_{i+1}**,
 - ii. if you can change the value you reached in (1) by using the next coin (**c_{i+1}**) together with other smaller coins so that the total number of used coins is less (or equal) number of coins in (1), then you have a **greedy choice** for this coin
2. If you get a greedy choice for all coins, then solve it **greedy**
3. Else, solve it **dynamic programming**



Examples:

1. Coins = {1, 5, 20, 30, 50}

For coin 1: we need 6 coins to **exceed** next coin ($6 \times 1 > 5$), we can reach 6 in less number of coins by using one coin from 5 and one from 1 → greedy-choice

For coin 5: we need 5 coins to **exceed** next coin ($5 \times 5 > 20$), we can reach 25 in less number of coins by using one coin from 20 and one from 5 → greedy-choice

For coin 20: we need 2 coins to **exceed** next coin ($2 \times 20 > 30$), we **can't** reach 40 in less number of coins since by using one coin from 30, we require additional two coins from 5 (total of 3 coins) → **NO** greedy-choice

→ Solve by DP

2. Coins = {1, 5, 10, 25}

For coin 1: we need 6 coins to **exceed** next coin ($6 \times 1 > 5$), we can reach 6 in less number of coins by using one coin from 5 and one from 1 → greedy-choice

For coin 5: we need 3 coins to **exceed** next coin ($3 \times 5 > 10$), we can reach 15 in less number of coins by using one coin from 10 and one from 5 → greedy-choice

For coin 10: we need 3 coins to **exceed** next coin ($3 \times 10 > 25$), we **can** reach 30 in less number of coins by using one coin from 25 and one from 5 → greedy-choice

→ Solve by Greedy

SOLUTION:

- Greedy-choice: determine the largest coin whose value is less than or equal to n . Let this coin have value c . Give one such coin,
- Sub-problem: making change for $n - c$ cents using the same set of coins
- pseudo-code: **Assumption:** coins are sorted in ascending order

```
index = numCoins
count = 0
while (value != 0)
{
    count += floor(value / coins[index])
    value = value % coins[index]
    index --
}
return count
```

- Complexity

⇒ $\Theta(N)$

Problem6: Task Scheduling II

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. You have one computer with a single CPU on which to run these tasks. Suppose also that the tasks are not all available at once. That is, each task has a **release time** r_i before which it is not available to be processed. In addition, **preemption** is allowed, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. We say that the completion time of a_i is 15.

Let c_i be the *completion time* of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $\sum_{i=1}^n c_i$.

SOLUTION:

Greedy-choice:

- At each newly released process, preempt the execution and select the process with minimum remaining time and schedule it next.
- Since we want to minimize average completion time, $\frac{1}{n} \sum_{i=1}^n c_i$. Thus, we need to minimize completion time of each process c_i .
- Since $c_i = p_i + w_i$, where, w_i is the waiting time of process (i) and p_i is its processing time.
- To minimize c_i , we need to minimize its waiting time. Since its waiting time is dependent on the previously scheduled processes. So, to minimize it, we need to select the process with the min remaining time at each newly released process.
- If the newly released process has the min remaining time among others, so scheduling it will ensure that it completes with ZERO wait and the other processes will continue with the min waiting time for each of them.
- Otherwise, selecting a process with the min remaining time will ensure that all other processes will continue with the min waiting time for each of them.

Sub-problem:

- Schedule the remaining $(n - 1)$ processes in optimal way

Problem7: Parents Gift

A loving daughter decided to prepare a precious gift for her parents. She bought **one basket** and decided to fill it with the most expensive fruits from the supermarket. Basket can be filled with a maximum weight **W**.

In the supermarket; there're 'N' fruits. For each one, the supermarket displays the **full remaining weights** 'weights[i]' with their total prices 'prices[i]'. It's required to make the daughter fill her basket with the most expensive collection of fruits. Following are two examples:

CASE#1 (Solved)	CASE#2 (Required?)																																										
W = 40	W = 7																																										
<table><tr><th>Item</th><th>Weight</th><th>Price</th></tr><tr><td>1</td><td>30</td><td>10\$</td></tr><tr><td>2</td><td>10</td><td>50\$</td></tr><tr><td>3</td><td>20</td><td>20\$</td></tr><tr><td>4</td><td>40</td><td>80\$</td></tr><tr><td>5</td><td>25</td><td>200\$</td></tr><tr><td>6</td><td>50</td><td>150\$</td></tr></table>	Item	Weight	Price	1	30	10\$	2	10	50\$	3	20	20\$	4	40	80\$	5	25	200\$	6	50	150\$	<table><tr><th>Item</th><th>Weight</th><th>Price</th></tr><tr><td>1</td><td>1</td><td>10\$</td></tr><tr><td>2</td><td>25</td><td>50\$</td></tr><tr><td>3</td><td>4</td><td>20\$</td></tr><tr><td>4</td><td>50</td><td>15\$</td></tr><tr><td>5</td><td>40</td><td>80\$</td></tr><tr><td>6</td><td>25</td><td>100\$</td></tr></table>	Item	Weight	Price	1	1	10\$	2	25	50\$	3	4	20\$	4	50	15\$	5	40	80\$	6	25	100\$
Item	Weight	Price																																									
1	30	10\$																																									
2	10	50\$																																									
3	20	20\$																																									
4	40	80\$																																									
5	25	200\$																																									
6	50	150\$																																									
Item	Weight	Price																																									
1	1	10\$																																									
2	25	50\$																																									
3	4	20\$																																									
4	50	15\$																																									
5	40	80\$																																									
6	25	100\$																																									
Max Profit = 265\$ (from items 2,5,6)	Max Profit = ??																																										

- Based on the problem specifications, find the solution of the second case (BOTH profit & item(s))?
- What's the time complexity of this efficient solution?

SOLUTION:

1. $10 + 20 + 2 \times 4 = 38$ from items 1, 3, 6

2. $O(N \log(N))$

Problem8: FCIS Activities

Suppose there're a set of activities **A[1...N]** to schedule among a large number of halls, where any activity can take place in any hall. Each activity has start time **S[i]** and finish time **F[i]**. It's required to **schedule all** the activities using as **few halls** as possible. Design an **efficient** algorithm to determine the **scheduled activities in each hall**. Start with the following function declaration:

Res = Sched(A[], S[], F[], N) ; where **Res** contains the scheduled activities in each hall

Following are examples: Solve the 3rd case?

Cases	Case#1					Case#2					Case#3 [REQUIRED??]					
Input	s[]	1	7	4		s[]	1	1	5	7	s[]	1	2	3	5	7
	f[]	5	10	8		f[]	4	6	10	9	f[]	4	7	6	9	9
Output	2					2					???					

1. Calculate the output of the missing case in the given example?
2. Specify the suitable **design paradigm** that lead to an efficient solution?
3. Based on the **CHOSEN paradigm**, answer the **corresponding requirements**:

D&C	DP	Greedy
Divide:.....	Formulate subproblem of each trial:	Greedy choice:
Conquer:.....	Extra storage size:	Describe subproblem:
Combine:.....	Complexity:	Complexity:
Recurrence: $T(\dots)=\dots$	Faster Sol.: TOP-DOWN OR BOTTOM-UP	

1. **3**
2. **Greedy**
3. Greedy choice: **Chose the activity with min start time and check it against the opened hall(s).**
Describe subproblem: **schedule the remaining activities using the min number of halls**
Complexity: **$O(N^2)$ using arrays OR $O(N \log(N))$ using priority queue**

Problem9: Restaurant Placement I

Smiley Donuts, a new donuts restaurant chain, wants to build restaurants on many street corners with the goal of maximizing their total profit.

The street network is described as an undirected graph $G = (V, E)$, where the potential restaurant sites are the vertices of the graph. Each vertex u has a nonnegative integer value p_u , which describes the potential **profit** of site u . Two restaurants cannot be built on adjacent vertices (to avoid self-competition).

You are supposed to design an algorithm that outputs the chosen set $U \subseteq V$ of sites that maximizes the total profit $\sum_{u \in U} p_u$. Suppose that the street network G is acyclic, i.e., a tree.

- a. Consider the following “greedy” restaurant-placement algorithm: Choose the highest-profit vertex u_0 in the tree (breaking ties according to some order on vertex names) and put it into U . Remove u_0 from further consideration, along with all of its neighbors in G . Repeat until no further vertices remain. Give a counterexample to show that this algorithm does not always give a restaurant placement with the maximum profit?
- b. Suppose that, in the absence of good market research, DD decides that all sites are equally good, so the goal is simply to design a restaurant placement with the largest number of locations. Give a simple greedy algorithm for this case, and prove its correctness?



SOLUTION:

- a. the tree could be a line (9, 10, 9)
- b. pick any node u_0 as the root of the tree and sort all the nodes following a depth-first search (DFS) and store the sorted nodes in array N . For each valid node in N , in the reverse order, include it and remove its parent from N .

Correctness:

If there exists an optimal placement O that excludes some leaf v . Simply add v to O and if necessary, remove its parent. The result is no worse than placement O . Repeat for all leaves until we have an optimal placement with all the leaves included.

Due to DFS sorting, the first valid node in N , in reverse order, must be a leaf node. So, it can be included in the optimal placement and its parent excluded. When nodes from N are processed in reverse order, each processed node is the last valid one in the current N and is thus a leaf. And including the leaf is part of an optimal solution for the corresponding subtree. Overall, an optimal placement is derived for the original tree.

Analysis:

Sorting nodes using DFS takes $O(n)$ time. The greedy algorithm also takes $O(n)$ time since it processes each node once. Overall the algorithm has $O(n)$ complexity.



SECOND: DESIGN PROBLEMS [without solution]

Problem1: Office Organization

Your office becomes full of documents. You currently have **N** units of documents on your office, and your father demands that you have exactly **M** units of documents left by the end of the day. The only hope for you now is to ask help from your brother and sister.

- Your sister offers that she can reduce your documents **by half** for **\$A** (rounding down when necessary – e.g. 25 → 12).
- Your brother offers that he can reduce your entire documents **by one unit** for **\$B**

Note that work can never be reduced to less than 0.

Given N, M, A and B, your task is to find the minimum costs in **MOST EFFICIENT WAY** to organize your office to meet your father needs.

Complexity

The complexity of your algorithm should be **less than $O(N)$**

Examples

- | | | |
|----|---------------------------------|-------------|
| 1. | $N = 100, M = 5, A = 10, B = 1$ | Output = 37 |
| 2. | $N = 100, M = 5, A = 5, B = 2$ | Output = 22 |



Problem2: Bread Transportation

City X consists of one street, and every inhabitant in the city is a bread salesman. Simple enough: everyone buys bread from other inhabitants of the city. Every day each inhabitant decides how much bread he wants to buy or sell. Interestingly, demand and supply is always the same, so that each inhabitant gets what he wants.

There is one problem, however: Transporting bread from one house to another results in work, since all bread is equally good, the inhabitants of City X don't care which persons they are doing trade with, they are only interested in selling or buying a specific amount of bread. They are clever enough to figure out a way of trading so that the overall amount of work needed for transports is minimized.

In this problem you are asked to reconstruct the trading during one day in City X. For simplicity we will assume that the houses are built along a straight line with equal distance between adjacent houses. Transporting one loaf of bread from one house to an adjacent house results in one unit of work.

Input Specification

Your function will get an array each cell represents a house. Each cell will contain either a positive number (wants to sell bread) or a negative number (wants to buy bread) the sum of the array is zero.

Output Specification

The minimum amount of work units needed so that every inhabitant has his demand fulfilled.

Examples

Example1: 5 -4 1 -3 1 (house 1 wants to sell 5 loafs of bread, house 2 wants to buy 4, house 3 wants to buy 1..)

Output: 9

Example 2: -1000 -1000 -1000 1000 1000 1000, output 9000

The answer should include the code, the trace of the above 2 examples and the algorithm complexity

Problem3: Candy & Sugar Rush

You have a bag full of candy (N pieces), you decided to eat all of that candy as fast as possible!. You start to eat one piece at a time. However, after having eaten some of the candy (C pieces) you get a sugar rush, enabling you to eat more than one piece at a time.

From previous experience, you noticed that you start eating one piece of candy at a time (i.e. one piece per mouthful). However, after having eaten C pieces, the sugar rush kicks in and you can eat twice as many pieces of candy at a time. Then after having eaten another C pieces, the amount you can eat at a time is doubled yet again! This continues until you finish all of your candy!

Write an algorithm to calculate how many mouthfuls you need to eat N pieces given that you get a sugar rush every C pieces

Input

N number of candy you have, C and the Count of pieces necessary for a sugar rush.

Output

Number of mouthfuls

Sample

Example	Input	Output
1	$N=10, C=1$	10
2	$N=10, C=5$	8

Sample explanation: In first case, the number of pieces doubles after each one. The entire batch is finished as $1 + 2 + 4 + 3 = 10$, just 4 steps. The last step could have been up to 8 pieces large, but only 3 pieces of candy remained at that point.

The second case has only a single sugar rush that matters, the one after 5 pieces. So the solution is $1 + 1 + 1 + 1 + 1 + 2 + 2 + 1 = 10$, which is 8 steps.



Problem4: Dividing Coins

You have a set of coins to split with your brother while he is sleeping. So, you've decided to stick to the following strategy to avoid suspicions: you take the minimum number of coins, whose sum of values is strictly more than the sum of values of the remaining coins. On this basis, determine what minimum number of coins you need to take to divide them in the described manner.

Input: an array of coin values; 1 2 3 (you have 3 coins first coin of value 1 second of value 2 ...)

Output: minimum number of coins to take

Input	Output	Explanation
3 3	2	2 coins (you and your brother have sums equal to 6, 0 correspondingly). If you take 1 coin, you get sums 3, 3. If you take 0 coins, you get sums 0, 6. Those variants do not satisfy you as your sum should be strictly more than your brother's sum.
2 1 2	2	In the second sample one coin isn't enough for us, too. You can pick coins with values 1, 2 or 2, 2. In any case, the minimum number of coins equals 2

Problem5: Mobile Base Stations

Let's consider a long river, along which **N** houses are scattered. You can think of this river as an axis, and the **houses[]** contains their coordinates on this axis in a sorted order. Your company wants to place mobile phone base stations **at certain points along the river**, so that every house is **within 4 kilometers** of one of the base stations. Give an efficient algorithm that minimizes the number of base stations used? Start with this function declaration:

Min # Stations = Place(houses[], N)

Following are examples: (4th Case is REQUIRED?)

Cases	Case#1	Case#2	Case#3	Case#4 [REQUIRED??]
Input	1, 10, 20, 30	5, 7, 8, 9, 11	2, 7, 10, 20, 21, 23, 24	4, 5, 6, 8, 11, 13, 16, 19, 25, 31, 35
Output	4	1	2	????

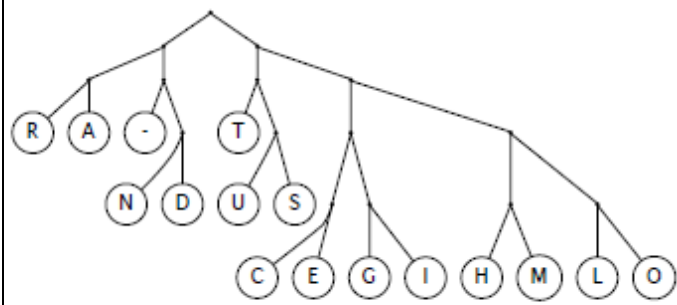
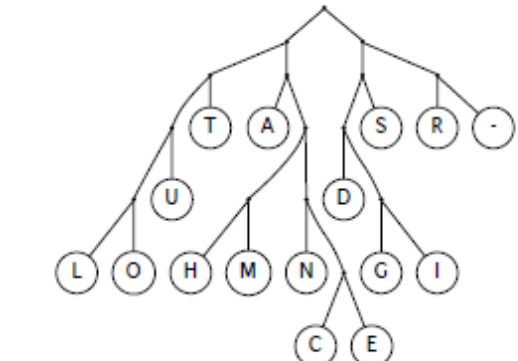
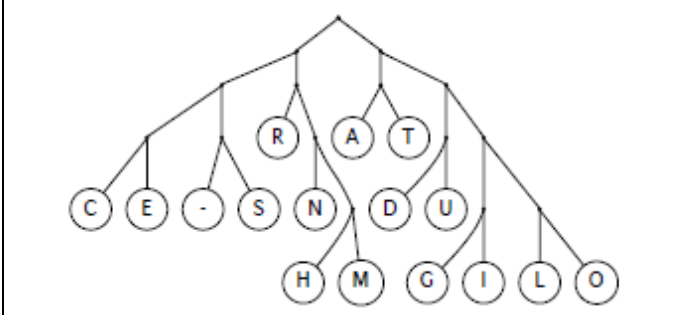
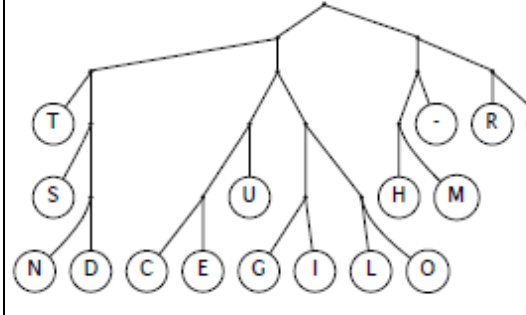
1. Calculate the output of the missing case in the given example?
2. Specify the suitable **design paradigm** that lead to an efficient solution?
3. Based on the **CHOSEN** paradigm, answer the **corresponding requirements**:

D&C	DP	Greedy
Divide:.....	Formulate subproblem of each trial:	Greedy choice:
Conquer:.....	Extra storage size:	Describe subproblem:
Combine:.....	Faster solution: Top-Down OR Bottom-up	Complexity:
Recurrence: T(...)=.....		
Complete pseudocode	Complete pseudocode	Complete pseudocode

THIRD: OTHER QUESTIONS

QUESTION1: Huffman Code

Consider the string "DATA-STRUCTURES-AND-ALGORITHMS": which of the following trees is/are considered optimal prefix-free code for this input string?

a)		b)			
c)		d)			
e)	a) and b)		f)	a) and c)	

Frequency of each char:

Char	A	T	S	R	-	D	U	C	E	N	L	G	O	I	H	M	TOTAL
Freq	4	4	3	3	3	2	2	1	1	1	1	1	1	1	1	1	
a)	3	3	4	3	3	4	4	5	5	4	5	5	5	5	5	5	40+32+42=114
b)	3	3	3	3	3	4	4	6	6	5	5	5	5	5	5	5	12+35+16+51=114
c)	3	3	4	3	4	4	4	4	4	4	5	5	5	5	5	5	30+52+33=115
d)	3	3	4	3	3	5	4	5	5	5	5	5	5	5	4	4	45+28+42=115

Correct choice: (e)

QUESTION2: Knapsack Variation

If you apply the **0-1 Knapsack algorithm** to a set of **equally weighted** items but with different costs. What's the suitable design paradigm that can be used to solve this problem? What's the complexity of such solution?

(**N**: number of items, **W**: Knapsack weight)

SOLUTION:

Greedy & $O(N \log(N))$

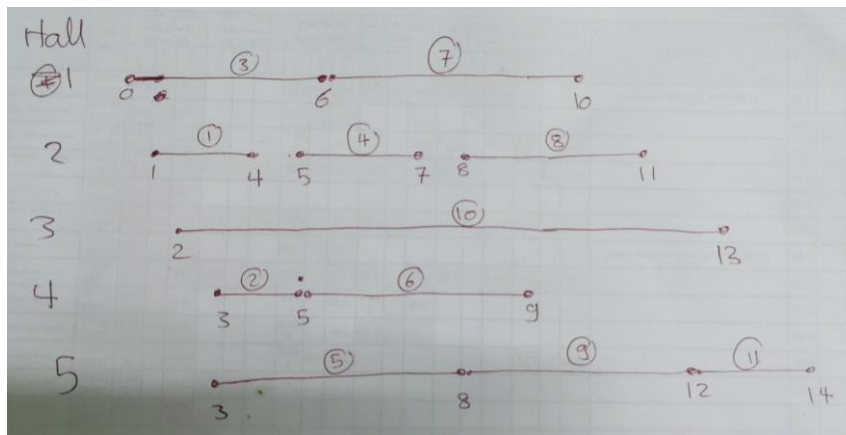
QUESTION3: Activity Selection Variation

In **Activity Selection II Problem**, given an infinite number of resources and a set of N activities, each with start time s_i and finish time f_i , it's required to schedule ALL activities using a minimum number of resources. Answer the following:

- For the following set of activities, what's the min number of resources to schedule ALL of them?

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

5



- The suitable Greedy choice to solve this problem is to select the activity with...

a. Min duration	b. Earliest start time	c. min num of overlaps	d. Earliest finish time	e. None of choices
-----------------	------------------------	------------------------	-------------------------	--------------------