CHAPTER

17

GREEDY ALGORITHMS

17.1 Introduction

Let us start our discussion with simple theory that will give us an understanding of the Greedy technique. In the game of *Chess*, every time we make a decision about a move, we have to also think about the future consequences. Whereas, in the game of *Tennis* (or *Volleyball*), our action is based on the immediate situation. This means that in some cases making a decision that looks right at that moment gives the best solution (*Greedy*), but in other cases it doesn't. The Greedy technique is best suited for looking at the immediate situation.

17.2 Greedy Strategy

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some *local best* is chosen. It assumes that a local good selection makes for a global optimal solution.

17.3 Elements of Greedy Algorithms

The two basic properties of optimal Greedy algorithms are:

1) Greedy choice property
2) Optimal substructure

Greedy choice property

This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solutions to solve larger problems.

17.4 Does Greedy Always Work?

Making locally optimal choices does not always work. Hence, Greedy algorithms will not always give the best solutions. We will see particular examples in the *Problems* section and in the *Dynamic Programming* chapter.

17.5 Advantages and Disadvantages of Greedy Method

The main advantage of the Greedy method is that it is straightforward, easy to understand and easy to code. In Greedy algorithms, once we make a decision, we do not have to spend time re-examining the already computed values. Its main disadvantage is that for many problems there is no greedy algorithm. That means, in many cases there is no guarantee that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

17.6 Greedy Applications

- Sorting: Selection sort, Topological sort
- Priority Queues: Heap sort
- Huffman coding compression algorithm
- Prim's and Kruskal's algorithms
- Shortest path in Weighted Graph [Dijkstra's]
- Coin change problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as an approximation algorithm for complex problems

17.7 Understanding Greedy Technique

For better understanding let us go through an example.

Huffman Coding Algorithm

Definition

Given a set of n characters from the alphabet A [each character $c \in A$] and their associated frequency $freq(c)$, find a binary code for each character $c \in A$, such that $\sum_{c \in A} freq(c)|binarycode(c)|$ is minimum, where $|binarycode(c)|$ represents the length of binary code of character c. That means the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation].

The basic idea behind the Huffman coding algorithm is to use fewer bits for more frequently occurring characters. The Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. When reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character *'e'* is used 10 times more frequently than the character *'q'*. It would then be advantageous for us to instead use a 7 bit code for e and a 9 bit code for *q* because that could reduce our overall message length.

On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters. Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

An Example

Let's assume that after scanning a file we find the following character frequencies:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

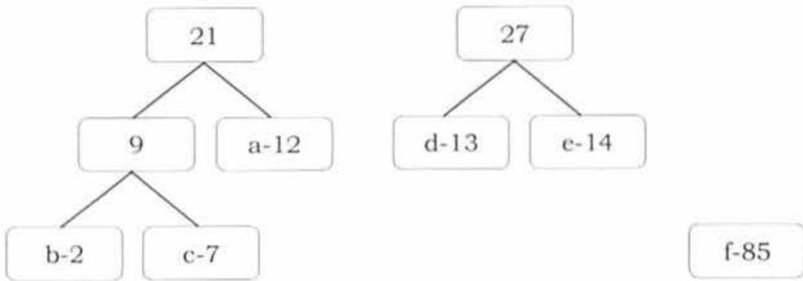
Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

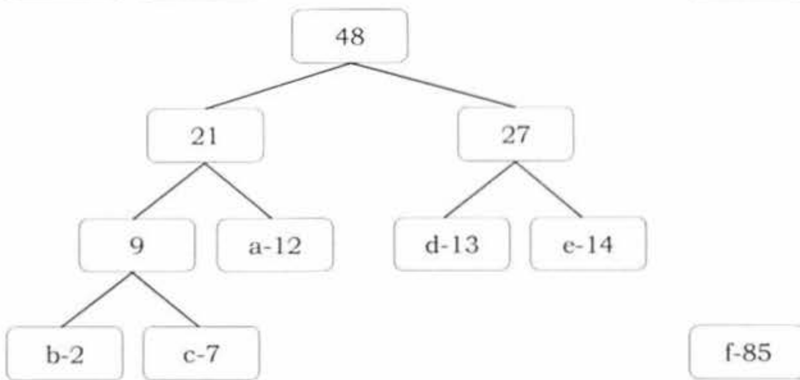
| b-2 | c-7 | a-12 | d-13 | e-14 | f-85 |

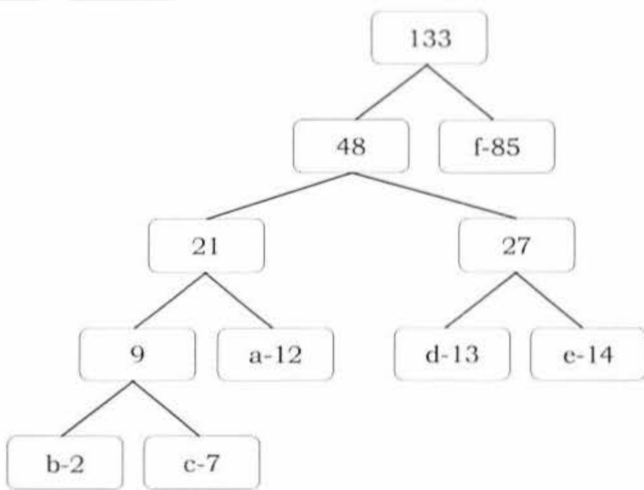
The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes.

Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like this:

Repeat this process until only one tree is left:







Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Calculating Bits Saved

Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that the number of bits that are used to store the data using the Huffman code. In the above example, since we have six characters, let's assume each character is stored with a three bit code. Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is $3 * 133 = 399$. Using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved $399 - 238 = 161$ bits, or nearly 40% of the storage space.

```
from heapq import heappush, heappop, heapify
from collections import defaultdict
def HuffmanEncode(characterFrequency):
    heap = [[freq, [sym, ""]] for sym, freq in characterFrequency.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
inputText = "this is an example for huffman encoding"
characterFrequency = defaultdict(int)
for character in inputText:
    characterFrequency[character] += 1
huffCodes = HuffmanEncode(characterFrequency)
print "Symbol\tFrequency\tHuffman Code"
for p in huffCodes:
    print "%s\t\t\t%s\t\t\t%s" % (p[0], characterFrequency[p[0]], p[1])
```

Time Complexity: $O(nlogn)$, since there will be *one* build_heap, $2n - 2$ delete_mins, and $n - 2$ inserts, on a priority queue that never has more than n elements. Refer to the *Priority Queues* chapter for details.

17.8 Greedy Algorithms: Problems & Solutions

Problem-1 Given an array F with size n. Assume the array content $F[i]$ indicates the length of the i^{th} file and we want to merge all these files into one single file. Check whether the following algorithm gives the best solution for this problem or not?

Algorithm: Merge the files contiguously. That means select the first two files and merge them. Then select the output of the previous merge and merge with the third file, and keep going...

Note: Given two files A and B with sizes m and n, the complexity of merging is $O(m + n)$.

Solution: This algorithm will not produce the optimal solution. For a counter example, let us consider the following file sizes array.

$$F = \{10,5,100,50,20,15\}$$

As per the above algorithm, we need to merge the first two files (10 and 5 size files), and as a result we get the following list of files. In the list below, 15 indicates the cost of merging two files with sizes 10 and 5.

$$\{15,100,50,20,15\}$$

Similarly, merging 15 with the next file 100 produces: $\{115,50,20,15\}$. For the subsequent steps the list becomes

$$\{165,20,15\}, \{185,15\}$$

Finally, $\{200\}$

The total cost of merging = Cost of all merging operations = $15 + 115 + 165 + 185 + 200 = 680$.

To see whether the above result is optimal or not, consider the order: $\{5, 10, 15, 20, 50, 100\}$. For this example, following the same approach, the total cost of merging = $15 + 30 + 50 + 100 + 200 = 395$. So, the given algorithm is not giving the best (optimal) solution.

Problem-2 Similar to Problem-1, does the following algorithm give the optimal solution?

Algorithm: Merge the files in pairs. That means after the first step, the algorithm produces the $n/2$ intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going.

Note: Sometimes this algorithm is called 2-way merging. Instead of two files at a time, if we merge K files at a time then we call it K-way merging.

Solution: This algorithm will not produce the optimal solution and consider the previous example for a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the third pair of files (20 and 15). As a result we get the following list of files.

$$\{15, 150, 35\}$$

Similarly, merge the output in pairs and this step produces [below, the third element does not have a pair element, so keep it the same]:

$$\{165,35\}$$

Finally, $\{185\}$

The total cost of merging = Cost of all merging operations = $15 + 150 + 35 + 165 + 185 = 550$. This is much more than 395 (of the previous problem). So, the given algorithm is not giving the best (optimal) solution.

Problem-3 In Problem-1, what is the best way to merge *all the files* into a single file?

Solution: Using the Greedy algorithm we can reduce the total time for merging the given files. Let us consider the following algorithm.

Algorithm:

1. Store file sizes in a priority queue. The key of elements are file lengths.
2. Repeat the following until there is only one file:
    a. Extract two smallest elements X and Y.
    b. Merge X and Y and insert this new file in the priority queue.

Variant of same algorithm:

1. Sort the file sizes in ascending order.
2. Repeat the following until there is only one file:
    a. Take the first two elements (smallest) X and Y.
    b. Merge X and Y and insert this new file in the sorted list.

To check the above algorithm, let us trace it with the previous example. The given array is:

$$F = \{10,5,100,50,20,15\}$$

As per the above algorithm, after sorting the list it becomes: $\{5, 10, 15, 20, 50,100\}$. We need to merge the two smallest files (5 and 10 size files) and as a result we get the following list of files. In the list below, 15 indicates the cost of merging two files with sizes 10 and 5.

$$\{15,15,20,50,100\}$$

Similarly, merging the two smallest elements (15 and 15) produces: $\{20,30,50,100\}$. For the subsequent steps the list becomes

{50,50,100} //merging 20 and 30
{100,100} //merging 20 and 30

Finally, {200}

The total cost of merging = Cost of all merging operations = $15 + 30 + 50 + 100 + 200 = 395$. So, this algorithm is producing the optimal solution for this merging problem.

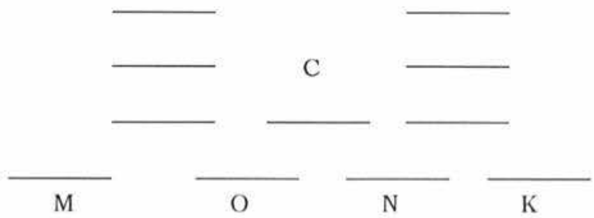
Time Complexity: $O(nlogn)$ time using heaps to find best merging pattern plus the optimal cost of merging the files.

Problem-4 **Interval Scheduling Algorithm:** Given a set of n intervals $S = \{(\text{start}_i, \text{end}_i) | 1 \le i \le n\}$. Let us assume that we want to find a maximum subset S' of S such that no pair of intervals in S' overlaps. Check whether the following algorithm works or not.

Algorithm: while (S is not empty) {

Select the interval I that overlaps the least number of other intervals.
Add I to final solution set S'.
Remove all intervals from S that overlap with I.
}

Solution: This algorithm does not solve the problem of finding a maximum subset of non-overlapping intervals. Consider the following intervals. The optimal solution is $\{M, O, N, K\}$. However, the interval that overlaps with the fewest others is C, and the given algorithm will select C first.

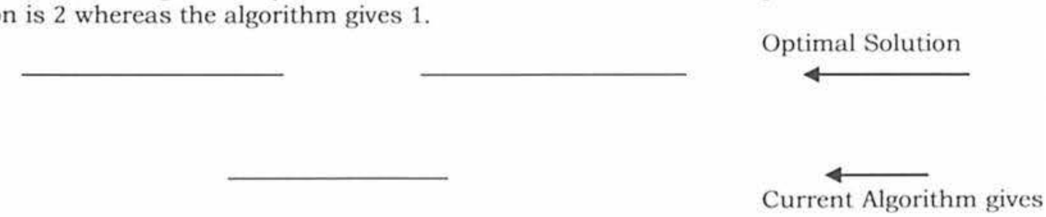


Problem-5 In Problem-4, if we select the interval that starts earliest (also not overlapping with already chosen intervals), does it give the optimal solution?

Solution: No. It will not give the optimal solution. Let us consider the example below. It can be seen that the optimal solution is 4 whereas the given algorithm gives 1.

Problem-6 In Problem-4, if we select the shortest interval (but it is not overlapping the already chosen intervals), does it give the optimal solution?

Solution: This also will not give the optimal solution. Let us consider the example below. It can be seen that the optimal solution is 2 whereas the algorithm gives 1.



Problem-7 For Problem-4, what is the optimal solution?

Solution: Now, let us concentrate on the optimal greedy solution.

Algorithm:

Sort intervals according to the right-most ends [end times];
for every consecutive interval {
- If the left-most end is after the right-most end of the last selected interval then we select this interval
- Otherwise we skip it and go to the next interval

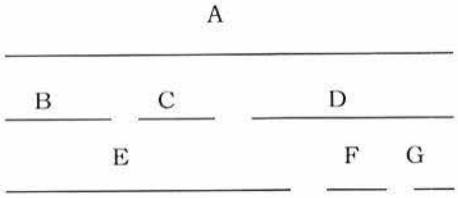
}

Time complexity = Time for sorting + Time for scanning = $O(nlogn + n) = O(nlogn)$.

Problem-8 Consider the following problem.
Input: $S = \{(start_i, end_i) | 1 \le i \le n\}$ of intervals. The interval $(start_i, end_i)$ we can treat as a request for a room for a class with time $start_i$ to time end_i.
Output: Find an assignment of classes to rooms that uses the fewest number of rooms.
Consider the following iterative algorithm. Assign as many classes as possible to the first room, then assign as many classes as possible to the second room, then assign as many classes as possible to the third room, etc. Does this algorithm give the best solution?

Note: In fact, this problem is similar to the interval scheduling algorithm. The only difference is the application.

Solution: This algorithm does not solve the interval-coloring problem. Consider the following intervals:



Maximizing the number of classes in the first room results in having $\{B, C, F, G\}$ in one room, and classes A, D, and E each in their own rooms, for a total of 4. The optimal solution is to put A in one room, $\{ B, C, D \}$ in another, and $\{E, F, G\}$ in another, for a total of 3 rooms.

Problem-9 For Problem-8, consider the following algorithm. Process the classes in increasing order of start times. Assume that we are processing class C. If there is a room R such that R has been assigned to an earlier class, and C can be assigned to R without overlapping previously assigned classes, then assign C to R. Otherwise, put C in a new room. Does this algorithm solve the problem?

Solution: This algorithm solves the interval-coloring problem. Note that if the greedy algorithm creates a new room for the current class c_i, then because it examines classes in order of start times, c_i start point must intersect with the last class in all of the current rooms. Thus when greedy creates the last room, n, it is because the start time of the current class intersects with $n - 1$ other classes. But we know that for any single point in any class it can only intersect with at most s other class, so it must then be that $n \leq S$. As s is a lower bound on the total number needed, and greedy is feasible, it is thus also optimal.

Note: For optimal solution refer to Problem-7 and for code refer to Problem-10.

Problem-10 Suppose we are given two arrays $Start[1..n]$ and $Finish[1..n]$ listing the start and finish times of each class. Our task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $Start\ [i] > Finish[j]$ or $Start\ [j] > Finish\ [i]$

Solution: Our aim is to finish the first class as early as possible, because that leaves us with the most remaining classes. We scan through the classes in order of finish time, and whenever we encounter a class that doesn't conflict with the latest class so far, then we take that class.

```
def LargestTasks(Start, n,  Finish):
    sort Finish[]
    rearrange Start[] to match
    count = 1
    X[count]  = 1
    for i in range(2,n):
        if(Start[i] > Finish[X[count]]):
            count  = count + 1
            X[count] = I
    return X[1:count]
```

This algorithm clearly runs in $O(nlogn)$ time due to sorting.
This algorithm clearly runs in $O(nlogn)$ time due to sorting.

Problem-11 Consider the making change problem in the country of India. The input to this problem is an integer M. The output should be the minimum number of coins to make M rupees of change. In India, assume the available coins are $1, 5, 10, 20, 25, 50$ rupees. Assume that we have an unlimited number of coins of each type.

For this problem, does the following algorithm produce the optimal solution or not? Take as many coins as possible from the highest denominations. So for example, to make change for 234 rupees the greedy algorithm would take four 50 rupee coins, one 25 rupee coin, one 5 rupee coin, and four 1 rupee coins.

Solution: The greedy algorithm is not optimal for the problem of making change with the minimum number of coins when the denominations are $1, 5, 10, 20, 25$, and 50. In order to make 40 rupees, the greedy algorithm would use three coins of $25, 10$, and 5 rupees. The optimal solution is to use two 20-shilling coins.

Note: For the optimal solution, refer to the *Dynamic Programming* chapter.

Problem-12 Let us assume that we are going for a long drive between cities A and B. In preparation for our trip, we have downloaded a map that contains the distances in miles between all the petrol stations on our route. Assume that our car's tanks can hold petrol for n miles. Assume that the value n is given. Suppose we stop at every point. Does it give the best solution?

Solution: Here the algorithm does not produce optimal solution. Obvious Reason: filling at each petrol station does not produce optimal solution.

Problem-13 For problem Problem-12, stop if and only if you don't have enough petrol to make it to the next gas station, and if you stop, fill the tank up all the way. Prove or disprove that this algorithm correctly solves the problem.

Solution: The greedy approach works: We start our trip from A with a full tank. We check our map to determine the farthest petrol station on our route within n miles. We stop at that petrol station, fill up our tank and check our map again to determine the farthest petrol station on our route within n miles from this stop. Repeat the process until we get to B.

Note: For code, refer to *Dynamic Programming* chapter.

Problem-14 **Fractional Knapsack problem:** Given items $t_1, t_2, \ldots, t_n$ (items we might want to carry in our backpack) with associated weights $s_1, s_2, \ldots, s_n$ and benefit values $v_1, v_2, \ldots, v_n$, how can we maximize the total benefit considering that we are subject to an absolute weight limit C?

Solution:

Algorithm:

1) Compute value per size density for each item $d_i = \frac{v_i}{s_i}$.
2) Sort each item by its value density.
3) Take as much as possible of the density item not already in the bag

Time Complexity: $O(nlogn)$ for sorting and $O(n)$ for greedy selections.

Note: The items can be entered into a priority queue and retrieved one by one until either the bag is full or all items have been selected. This actually has a better runtime of $O(n + clogn)$ where c is the number of items that actually get selected in the solution. There is a savings in runtime if $c = O(n)$, but otherwise there is no change in the complexity.

Problem-15 **Number of railway-platforms:** At a railway station, we have a time-table with the trains' arrivals and departures. We need to find the minimum number of platforms so that all the trains can be accommodated as per their schedule.

Example: The timetable is as given below, the answer is 3. Otherwise, the railway station will not be able to accommodate all the trains.

Rail	Arrival	Departure
Rail A	0900 hrs	0930 hrs
Rail B	0915 hrs	1300 hrs
Rail C	1030 hrs	1100 hrs
Rail D	1045 hrs	1145 hrs

Solution: Let's take the same example as described above. Calculating the number of platforms is done by determining the maximum number of trains at the railway station at any time.

First, sort all the arrival(A) and departure(D) times in an array. Then, save the corresponding arrivals anddepartures in the array also. After sorting, our array will look like this:

0900	0915	0930	1030	1045	1100	1145	1300
A	A	D	A	A	D	D	D

Now modify the array by placing 1 for A and -1 for D. The new array will look like this:

1	1	-1	1	1	-1	-1	-1

Finally make a cumulative array out of this:

1	2	1	2	3	2	1	0

Our solution will be the maximum value in this array. Here it is 3.

Note: If we have a train arriving and another departing at the same time, then put the departure time first in the sorted array.

Problem-16 Consider a country with very long roads and houses along the road. Assume that the residents of all houses use cell phones. We want to place cell phone towers along the road, and each cell phone tower covers a range of 7 kilometers. Create an efficient algorithm that allow for the fewest cell phone towers.

Solution:

The algorithm to locate the least number of cell phone towers:

1) Start from the beginning of the road
2) Find the first uncovered house on the road
3) If there is no such house, terminate this algorithm. Otherwise, go to next step
4) Locate a cell phone tower 7 miles away after we find this house along the road
5) Go to step 2

Problem-17 **Preparing Songs Cassette:** Suppose we have a set of n songs and want to store these on a tape. In the future, users will want to read those songs from the tape. Reading a song from a tape is not like reading from a disk; first we have to fast-forward past all the other songs, and that takes a significant amount of time. Let $A[1..n]$ be an array listing the lengths of each song, specifically, song i has length $A[i]$. If the songs are stored in order from 1 to n, then the cost of accessing the k^{th} song is:

$$C(k) = \sum_{i=1}^{k} A[i]$$

The cost reflects the fact that before we read song k we must first scan past all the earlier songs on the tape. If we change the order of the songs on the tape, we change the cost of accessing the songs, with the result that some songs become more expensive to read, but others become cheaper. Different song orders are likely to result in different expected costs. If we assume that each song is equally likely to be accessed, which order should we use if we want the expected cost to be as small as possible?

Solution: The answer is simple. We should store the songs in the order from shortest to longest. Storing the short songs at the beginning reduces the forwarding times for the remaining jobs.

Problem-18 Let us consider a set of events at *HITEX* (*Hyderabad Convention Center*). Assume that there are n events where each takes one unit of time. Event i will provide a profit of **P[i]** rupees (**P[i]** > 0) if started at or before time $T[i]$, where $T[i]$ is an arbitrary number. If an event is not started by $T[i]$ then there is no benefit in scheduling it at all. All events can start as early as time 0. Give the efficient algorithm to find a schedule that maximizes the profit.

Solution:

Algorithm:

- Sort the jobs according to floor($T[i]$) (sorted from largest to smallest).
- Let time t be the current time being considered (where initially t = floor($T[i]$)).
- All jobs i where floor($T[i]$) = t are inserted into a priority queue with the profit $\mathbf{g_i}$ used as the key.
- A *DeleteMax* is performed to select the job to run at time t.
- Then t is decremented and the process is continued.

Clearly the time complexity is $O(nlogn)$. The sort takes $O(nlogn)$ and there are at most n insert and DeleteMax operations performed on the priority queue, each of which takes $O(logn)$ time.

Problem-19 Let us consider a customer-care server (say, mobile customer-care) with n customers to be served in the queue. For simplicity assume that the service time required by each customer is known in advance and it is w_i minutes for customer i. So if, for example, the customers are served in order of increasing i, then the i^{th} customer has to wait: $\sum_{j=1}^{n-1} w_j$ *minutes*. The total waiting time of all customers can be given as $= \sum_{i=1}^{n} \sum_{j=1}^{i-1} w_j$. What is the best way to serve the customers so that the total waiting time can be reduced?

Solution: This problem can be easily solved using greedy technique. Since our objective is to reduce the total waiting time, what we can do is, select the customer whose service time is less. That means, if we process the customers in the increasing order of service time then we can reduce the total waiting time.

Time Complexity: $O(nlogn)$.