**DSAD Assignment 2**

**PS5 Design Document**

**1. Problem Statement**

You are a part of a class. Your teacher has given a set of questions to the class. Each question takes 1 day to finish the task. Each problem has a deadline, if finished before the deadline you get the extra bonus marks. If you don’t finish the problem before the deadline, you won’t get the bonus. The formulation of the problem is as follows

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | Problem1 | Problem2 | Problem3 | Problem4 | Problem5 |
| **Deadlines** | 1 | 2 | 3 | 1 | 4 |
| **Bonuses** | 20 | 40 | 10 | 10 | 20 |

Maximise the profit by correctly scheduling the problem using greedy-method.

**2. Data Structure Model & Approach to the Problem**

Every problem is represented as a class having three instance variables – Name, Bonus and Deadline.

We read the input and create a list of these problems objects.

Now, we need to use greedy approach to schedule the problems. We do it as follows.

First, we sort the problems in non-increasing order of their bonuses (We are using merge sort algorithm to perform the sorting). We want to be greedy and always consider the problem with highest bonus first and then the next highest and so on.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | Problem2 | Problem1 | Problem5 | Problem3 | Problem4 |
| **Deadlines** | 2 | 1 | 4 | 3 | 1 |
| **Bonuses** | 40 | 20 | 20 | 10 | 10 |

After this, we iterate over this sorted list and schedule the problems in such a way that we don’t miss the deadline but always do it just before that individual problem’s deadline. This means we will only schedule as many problems as the highest unit of time in the *Deadlines* row. Since anything done after deadline is not going to result in any profit, we do not bother to solve those problems even if they remain unsolved.  
  
In the above example, the scheduling is done as follows in steps

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Time Slots** | 0-1 | 1-2 | 2-3 | 3-4 |
| **Step1** |  | Problem2 |  |  |
| **Step2** | Problem1 |  |  |  |
| **Step3** |  |  |  | Problem5 |
| **Step4** |  |  | Problem3 |  |

- As Problem2 gives maximum bonus, we first try to schedule that problem. Since it has a deadline of 2 units of time and it only needs 1 unit of time for completion, we schedule it in the time slot of 1-2.

- Next problems with the highest bonus are Problem1 and Problem5. Both have the same bonus. However, problem1 has a deadline of 1 unit of time and Problem5 has deadline of 4 units of time. Hence, we schedule them in first and last time slots respectively.

- Finally, we see that Problem3 and Problem4 both have a same bonus; however, we can no longer schedule Problem4 because we have already scheduled jobs before a deadline of 1 unit of time. But Problem3 has 3 units of time as deadline and time slot 2-3 is vacant, which means we can schedule it to happen in that time interval.

We model the problem in a greedy way as shown above and perform the problem scheduling. The bonus which is accrued is given by the sum of bonuses of the problems which could be scheduled to solve in the respective time slots shown above. For this example,

Bonus = Problem2\_Bonus + Problem1\_Bonus + Problem5\_Bonus + Problem3\_Bonus = 40 + 20 + 20 + 10 = 90

**3. Run time analysis**

Firstly, we need to sort the problem list in non-increasing order of their bonuses. While there are several algorithms to do this, the most efficient algorithms for doing this problem run in O(nlogn) complexity. We have selected merge sort for achieving this objective.  
  
Merge sort divides the problem into halves until there’s only one element in each step and eventually, we combine these results to get a sorted sequence of elements. The recurrence relation for such problems can be given by  
  
T(n) = 2T(n/2) + O(n)  
  
We can solve this by master theorem   
Here, a = b = 2, c = 1

Hence the time complexity of average case problem is given using θ(nlogn) for sorting the problems in the decreasing order of their bonuses.

Subsequently, we go over this sorted list and schedule the problems as described in the table above which is simply a linear pass over the list with some constant number of comparisons.

So, the complexity of this step is constant \* n = O(n)

Things like initializing variables and doing some housekeeping stuff like writing the output of bonuses or reading the input from the command-line are independent of n, so those will take some constant time c.   
  
We end up with the final complexity as  
  
O(nlogn) + O(n) + O(1) = O(nlogn)  
  
Our algorithm will take nlogn steps as n asymptotically approaches infinity.

**4. Alternate way of solving the Problem**

We have used a greedy way for solving this problem but there could be several other ways to model this problem. We can solve this problem in a couple of other ways.

1. *Brute Force method.*  
   Since we know that there will be no advantage/bonus if we exceed the deadline, we can find the highest deadline in the group of problems, let’s call it as “d”  
     
   Once we know this, we can list down all the permutations/schedules of n problems in d time-slots. There will be nPd such permutations.  
     
   For each permutation we can compute the bonus and we can then select that permutation which has the largest bonus amongst these set of schedules.  
     
   Finding the highest deadline will be O(n) because we will have to make a linear pass over problem list to find maximum.  
   For each permutation, computing the bonus will need a linear pass over the scheduled problem string. In worst case, let’s say we have d = n. Then we will have nPn = n! Permutations and each permutation will need n operations (one linear pass) so, this step will require n!\*n operations in the worst case.  
     
   So our time complexity in the worst case will be O(n!\*n).
2. *Generic Algorithm*  
   In this method, similar to the permutations we generated in the previous method, we start with an initial set of problem schedules.   
     
   We call these problem schedules as candidates of generation 0 and we find out their respective bonuses. The idea is to evolve these candidates over several generations such that the bonus value increases in every subsequent generation.  
     
   Those candidates in generation 0 who have high bonuses are referred to as best fit candidates and they have a high probability of being sampled and those candidates with low bonuses are not so fit candidates and they have low probability of being sampled.  
     
   Once we have sampled best fit candidates from generation 0, they are mutated in some way. In our case, consider two of the problems in a sequence are swapped (This mutation could be custom and needs to be smartly decided to converge quickly). We also sample some items with low fitness from previous generation because we don’t know if some mutation in these can lead to a very fit candidate and we don’t want to discard that possibility (Exploration/Exploitation).   
     
   Then we find the best fit candidates in the new generation we created from above, and we do this over and over across generations till there’s little or no improvement in the subsequent generations (We can define a tolerance based on the problem at hand).  
     
   Here if we assume that we do this for g generations and there are n problems to be scheduled and m permutations are considered at each step, we can give an approximate measure of time complexity to be O(g(nm + nm + n)). So, the time complexity of this algorithm will be O(gnm).