# Homework 2 + challenge problems

Pratyay Dutta (Codeforces ID : pdutt005)

February 3, 2024

## Contents

# 1 Challenge Problems

## 1.1 Time turner : Submission ID - 243333816

**Explanation**: I employed a vanilla greedy approach at first. After sorting the items list according to the end times, I did two runs of the same algorithm which is:

- Select all the possible class timings with the greedy choice being the class with the earliest finish time
- Remove all the classes that has been covered in the previous run.
- Do the same run on the leftover items in the list.

However, this did not yield correct results for all test cases and i realised that the greedy choice wasnt right because i was not taking into account the starting time of the next class. I needed to include those classes which not only has the earliest end time but also the difference between the start time of the class and the end time of the previous class should be minimized.
To do this, After sorting the items list according to the end times, I initiated two *counter* variables which will keep track of the number of classes Hermoine is taking per run respectively. The first class is always going to be taken at first run. Initialising $c1 = 1$ and $c2 = 0$(the counter for the second run) we loop through the items list and do the following:

- As we encounter the first element that cant go into the first run, we put it into the second run and increase c2 by 1
- For every other element, we check if it can go into 1 or 2
  - If it can go into 1 and not 2, we put in 1 and increase c1 by 1
  - If it can go into 2 and not 1, we put in 2 and increase c2 by 1
  - If it can go into both, we check the difference between the start time of the current element and the end times of both the previous elements of 1 and 2. We then, put it into the one where the difference is less.

**Time complexity**: Since we are looping through the list of items only once, the time complexity of our algorithm is O(n)

## 1.2 Share candies : Submission ID - 244347084

**Explanation**: We realise that the maximum number of candies that everyone can have is the mean of the total candies(we call this point, right) and the minimum number of candies each person can have is the minimum candies a particular person possesses(we call this point, left). Our solution lies in between this interval definitely. I employed a binary search that finds the midpoint of this range(mid) and checks if having that number of candies by distribution is possible or not. There can be two cases:

- It is possible. Then the optimal solution lies between this value and the maximum value, so we change the the left to the current value i.e mid.
- It is not possible. Then the optimal solution lies between this value and the minimum value, so we do right=mid.

We continue doing this till the interval converges to a single number. For checking if a certain solution(number of candies) is possible or not, i made a function 'possible' that takes the argument as the list of candies and the current solution for checking.
I first sort the items in decreasing order of distance of each person. The idea is that if someone has excess candies then it passes over to the person closest to him (the next item in the list). Since the distance is always getting added when transferring candies from one person to another, it does not matter if a person C in A,B,C,D,E,F gets candies directly from F or from F through E and D. If someone has less candies then the deficit passes over in a similar way.
To implement this, we create two variables 'excess' and 'deficit'. While looping throught the list we check if the person has more candies than the argument *mid*:

- If yes, then it is relaxed to *mid* candies and the excess is the difference between its number of candies and *mid*. Now we check the existing deficit.
  - If the existing deficit is less than or equal to the current excess. Then we relax the deficit and pass on the excess to the next person after subtracting the distance. If the distance is more than the excess candies, then excess just becomes 0. The deficit, being met, becomes 0.
  - If the existing deficit is more than the current excess, then the deficit is met as much as possible from the current excess. The excess becomes 0 and the deficit, since its not fully met passes on to the next person after adding the distance. We add the distance to keep into consideration that if a deficit exists then it has to be covered by any future excess that we encounter and it has to come back through the path.
- If no, then the deficit increments by (numcandies-*mid*). Now we check the existing excess:
  - If the existing excess is greater than or equal to the deficit. Then we compensate this deficit with the existing excess and pass on the remaining excess to the next person aftersubtracting the distance and if the distance is more than the remaining excess then the excess becomes 0. The deficit,being met, becomes 0.
  - If the existing excess is less than the deficit, then the deficit is met as much as possible from the excess. The excess becomes 0 and the deficit, since its not fully met passes on to the next person after adding the distance.

After we finish the loop, we check the final excess and deficit. If the deficit is larger than the excess then it is not possible for everyone to have *mid* candies due to shortage of candies. If the excess is larger that means it is possible for everyone to have *mid* candies. The function returns a boolean after which, the binary search, as previously explained, continues.
**Time Complexity**:The binary search takes O(logn) time. In every iteration of the binary search, we are looping over all the elements in our list which is O(n). Therefore, the time complexity of our algorithm is O(nlogn).

### 1.3 Security system : Submission ID - 244639110

**Explanation**: The high level idea which I used to solve the problem: Till there is no rows blocked, we remove the largest furniture which is blocking the maximum number of blocked rows.
I maintain a list *excess* which tracks the number of blockages in each row.
The dictionary $f$ keeps track of which rows are being intersected by each furniture.
The *blocking* dictionary keeps track of the number of blocked rows intersected by each furniture. If all the values of this dictionary is 0, it means that no rows are blocked anymore. So we keep removing furnitures till all the values of this dictionary are 0 and we keep track of how many times we are removing a furniture with the *removed* variable.
We find the maximum number of blocked rows intersected by each furniture and we have to remove one of those furnitures. The remove function:

- Finds the largest furniture which is intersecting the max number of blocked rows.
- We remove that furniture and update our *excess* list such that the blockages decrease by one in those rows which were being intersected by the removed furniture.
- A furniture getting removed means that the furniture in dictionary $f$ gets an empty set i.e. that furniture no longer blocks anymore rows.
- Recomputes the *blocking* dictionary from the updated $f$ dictionary and *excess* list.
- Returns the updated *excess* and *blocking*.

# 2 Deadlines again

## 2.1 Deadline first

Let the distribution be as follows:

| Assignment | Points | Deadline |
|---|---|---|
| A | 5 | 1 |
| B | 8 | 2 |
| C | 9 | 2 |

Given the data, if we choose a greedy method that chooses the earliest deadline, then we would first complete assignment A and then choose between assignment B and C according to any technique to deal with ties. In that case, the total points would be $5 + 8 = 13$ or $5 + 9 = 14$. We can see that neither of them are the optimal solution since we can complete assignments B and C to get 17 points which would be optimal. Therefore, the earliest deadline being the greedy choice is wrong.

## 2.2 Highest first

Let the distribution be as follows:

| Assignment | Points | Deadline |
|---|---|---|
| A | 9 | 3 |
| B | 8 | 2 |
| C | 2 | 1 |

Given the distribution, the greedy method of choosing the highest point assignment first would choose A first and then B. In that case the deadline of C would be passed so C will not be done. Total points would be 17. But optimal choice would be C,B and A which would give a total point of 19. Hence the highest first greedy strategy is not optimal.

## 2.3 Optimal solution must contain assignment with highest points

Let there exist an optimal solution $S_1 = a_1, a_2, ..., a_n$.
The points for the respective assignment are $p_1, p_2, ...., p_n$.
Therefore, the total number of points scored $= p_1 + p_2 + .... + p_n = T_1$.
Let there be an assignment $a_j$ with deadline $1 < d_j < d_n$ not in $S_1$ which has the highest point $= p_j$.
Since all the assignments are done in a day, we can create another solution $S_2 = a_1, a_2, ..., a_j, ...a_n$ by swapping out any assignment $a_i$ with deadline $d_i <= d_j$.
This substitution is valid because the assignment $a_j$ is being done before its respective deadline and doing it does not affect the other assignment completions because each assignment is done independently in one day. Therefore $T_2$ for solution $S_2 = T_1 - p_i + p_j$.
Now $p_j > p_i$ since $p_j$ is the highest point.
Therefore $T_2 > T_1$.
Therefore $S_1$ is not optimal which means our original assumption was wrong and hence it is proven that the optimal solution always has to contain the assignment with the highest point.

## 2.4 Greedy algorithm

Our greedy approach to solve this problem will be : **Pick the work with highest points and do it on the day of the deadline or on the day closest to the deadline**. Our algorithm is as follows:

- Pick the work with highest points.
- Check if the day of the deadline is empty. If empty, we do the work then.
- If deadline day is not empty, we iterate back from the day of the deadline checking for a free day and we do the work on the day we find free.
- If we reach the start of the list without finding an empty date, it means that we cannot do the work for getting the optimal solution.
- Repeat with the next highest point work.

This will guarantee that we get the most number of points i.e it is the optimal solution. The proof of the optimality of this technique is given in part 6 and 7 of this question.

## 2.5 Example test case to test our algorithm

Given distribution:

| Assignment | Points | Deadline |
|---|---|---|
| A | 10 | 13 |
| B | 8 | 2 |
| C | 7 | 2 |
| D | 15 | 5 |
| E | 1 | 7 |
| F | 12 | 3 |
| G | 4 | 4 |
| H | 5 | 5 |

Table 1: Data distribution

I coded the proposed algorithm in python and the output is as follows:

## Deadlines again

In [40]:

```
points = [10,8,7,15,1,12,4,5]
deadlines = [13,2,2,5,7,3,4,5]
items = [(points[i], deadlines[i]) for i in range(len(points))]

#Sorting the list in terms of points in decreasing order
items.sort(key=lambda x: x[0], reverse=True)
arr = [None for i in range(13)]

for i in range(len(items)):
    curr = items[i][0]                      #Highest point work chosen
    for j in range(items[i][1]-1, -1, -1):
        if arr[j]==None:
            arr[j] = curr                   #Put in the spot closest to the deadline
            break

p = 0
for i in range(len(arr)):
    if arr[i]:
        print(f'Work with points {arr[i]} done on day {i+1}')
        p+=arr[i]

print(f'\nTotal points scored = {p}')
```

```
Work with points 7 done on day 1
Work with points 8 done on day 2
Work with points 12 done on day 3
Work with points 5 done on day 4
Work with points 15 done on day 5
Work with points 1 done on day 7
Work with points 10 done on day 13

Total points scored = 58
```

Figure 1: Output for the given data

The algorithm sees the work with 15 points first and puts it on day 5.
It picks 12 and puts it on day 3.
It picks 10 and puts it on day day 13.
It picks 8 and puts it on day 2.
It picks 7. Now deadline of 7 is day 2 but day is already occupied with work worth 8 points. So algorithm puts 7 in day 1.
It picks 5. Deadline is day 5 but it is already taken by 15. So it checks day 4 which is empty and places it there.
It picks 4. Deadline is day 4 but it is already taken by 5. It checks day 3 which is taken by 12. It checks day 2 which is taken by 8. It checks day 1 which is taken by 7. Therefore, we cannot accomodate 4.
It picks 1 and puts it on day 7.
Total points scored $= 7 + 8 + 12 + 5 + 15 + 1 + 10 = 58$

## 2.6 Greedy choice proof

Let there be a solution $S_1 = a_1, a_2, ...a_n$ which is optimal and does not include the greedy choice $a_j$ having points $p_j$ with deadline $d_j$. we have to show that there exists an optimal solution that starts with our greedy choice. There can be 2 cases:

1. $a_j$ is in our solution $S_1$ : In this case, it is true that there exists an optimal solution that starts with our greedy choice.
2. $a_1 != a_j$ : Let $S_2$ be a solution made by swapping $a_j$ and $a_1$, the first element replaced by the greedy choice. We have to show that $S_2$ is a valid solution to the problem:
   - Since $a_j$ is our greedy choice, $a_j > a_i$ for $a_i$ in solution $S_1$. We can swap with any element in the solution because it always gets highest priority to be done since $p_j$ is highest.
   - The total points $T_2 > T_1$ since $p_j$ is the highest point being the greedy choice. Therefore, $S_2$ is optimal and it always contains the greedy choice.

Therefore, it is always safe to consider the greedy choice.

## 2.7 Optimal Substructure definition and proof

**Definition**: Optimal substructure means that the optimal solution of a problem must contain the optimal solutions of the subproblems derived from it. If a greedy solution is an optimal solution then the greedy solutions of the subproblems will also be the optimal solutions to them.
**Proof**: Let there be an optimal solution $S = (a_1, d_1), (a_2, d_2), ....(a_n, d_n)$ on O (O is the set of all assignments) where $a_n$ is the assignment name and $d_n$ is the day that they are being done. Let $S' = S - (a_1, d_1)$ be a set of all the assignments created by removing the first choice of S. Then S' must be optimal to $O' = 0-$All work that coincides or overlaps with $(a_1, d_1)$.
We make an assumption that S' is not optimal on O'. Then there must exist a solution A' which is optimal on O' such that $|A'| > |S'|$
Therefore $|A' + (a_1, d_1)| > |S' + (a_1, d_1)|$
or $|A| > |S|$ (A = solution formed after adding $(a_1, d_1)$ to A').
This is a contradiction since we know that S is the optimal solution.
Therefore our assumption was wrong. S' is optimal on O' which proves the existence of an optimal substructure.

# 3 Don't wait too long

Given S = 60, E = 20
**Optimal strategy** : Don't wait for elevator. Always use stairs.
**Our strategy** : W=40. We wait for 40 seconds. If elevator comes before 40 seconds, we take elevator. Else we take the stairs. Therefore:

- If elevator comes after t seconds where $t < W$ : Time $= t + 20$ which is less than 60. Therefore it is optimal.
- If elevator comes at W=40 seconds: Time $= 40 + 20 = 60$ which is equal to the optimal strategy. Therefore it is optimal.
- If elevator does not come before W=40 seconds : We take the stairs. Therefore time $= 40 + 60 = 100$.
  $c = 100/60 = 1.667$

For this problem, with the given times for lift and stairs, our algorithm is a c-approximate algorithm where c = 1.667
Generalising :
W = S-E

- If elevator comes : time = S-E+E = S (which is our optimal choice)
- If elevator does not come : time = S-E+S = $2S - E$
  c = $(2S - E)/S = 2 - E/S$

With our given approach we will never take time which is more than $2 - E/S$ times our optimal approach.