# TUTORIAT -6- P. A.

# GREEDY

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem*.

## 1. Planificarea optimă a unor spectacole într-o singură sală

Considerăm $n$ spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s, f)$, $[s, f)$, ... , $[s, f)$, toate dintr-o singură zi. Având la dispoziție o

singură sală, în care putem să planificăm un singur spectacol la un moment dat, să se

determine numărul maxim de spectacole care pot fi planificate fără suprapuneri. Un

spectacol $S$ poate fi programat după spectacolul $S$ dacă $s \geq f$ .

# 2. Planificarea unor spectacole folosind un număr minim de săli

Considerăm $n$ spectacole $S_1, S_2, \ldots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s1, f1), [s2, f2), \ldots, [sn, fn)$, toate dintr-o singură zi. Să se determine numărul minim de săli necesare astfel încât toate spectacolele să fie programate astfel încât să nu existe suprapuneri în nicio sală. Un spectacol $S$ poate fi programat după spectacolul $S$ dacă $sj \geq fi$.

# 3. Planificarea proiectelor cu profit maxim

Se consideră o mulțime de proiecte, fiecare având un termen limită și un profit asociat dacă proiectul este terminat până la termenul limită. Fiecare proiect este realizat într-o singură unitate de timp. Să se planifice proiectele (fără a se suprapune ca timp) astfel încât să se maximizeze profitul total.

Fișierul "proiecte.txt" conține, pe fiecare linie, numele, termenul limită și profitul asociat unui proiect. În fișierul "profit.txt" să se afișeze succesiunea de proiecte alese și profitul total obținut prin realizarea lor.

# 4. Minimizarea întârzierii maxime a unor activități

Fie o mulțime de **n** activități. Fiecare activitate are o anumită durată $\mathbf{d_i}$ (se execută într-un număr de unități de timp) și un termen limită $\mathbf{t_i}$ până la care ar trebui executată (un număr de unități de timp indicat de la începutul programării tuturor activităților $\mathbf{t_0 = 0}$). O activitate care începe la momentul $\mathbf{s_i}$ și se termină la momentul $\mathbf{f_i = s_i + d_i}$ are o întârziere de $\mathbf{h_i = max\{0, f_i - t_i\}}$ unități de timp. Se cere programarea activităților astfel încât să se minimizeze întârzierea maximă $\mathbf{H = max(h_i)}$.

Fișierul "activitati.txt" conține pe prima linie numărul de activități n, iar pe următoarele n linii conține câte două numere indicând durata și termenul limită al fiecărei activități. Să se afișeze în fișierul "intarzieri.txt" programarea activităților, pe câte o linie, astfel: intervalul ales ($s_i$ --> $f_i$), de lungime egală cu durata $d_i$, termenul limită $t_i$ și întârzierea $h_i$ pentru fiecare activitate. Pe ultima linie din fișier să se afișeze întârzierea maximă.

# 5. Plata unei sume folosind un număr minim de bancnote

Fie o mulțime de bancnote $\{B_0, B_1, \dots, B_n\}$ astfel încât $B_0 = 1$ (avem mereu bancnota unitate, pentru a putea plăti orice sumă) și $BB_{ii} \mid BB_{jj}$, $\forall\ 0 \leq i < j \leq n-2$ (cu excepția ultimelor 2 bancnote, toate valorile se divid cu toate valorile din listă mai mici decât ele). De exemplu se consideră bancnotele românești cu valorile $\{1, 5, 10, 50, 100, 200, 500\}$.

Pentru o sumă de bani S, să se determine o modalitate de a plăti suma S folosind un număr minim de bancnote. Fișierul "bani.txt" conține pe prima linie valorile bancnotelor disponibile (se consideră că avem la dispoziție un număr infinit din fiecare bancnotă), iar pe a doua linie valoarea sumei S. În fișierul "plata.txt" să se afișeze ce bancnote cu

valori diferite și câte din fiecare valoare s-au folosit pentru a achita suma S.

# 6. Turn de înălțime maximă format din cuburi

Se dă o mulțime de n cuburi. Fiecare cub este caracterizat prin lungimea laturii și culoare. Nu există două cuburi având aceeași dimensiune. Fișierul “cuburi.txt” conține pe prima linie un număr natural nenul n (numărul de cuburi), apoi pe următoarele n linii câte un număr natural nenul (lungimea laturii cubului) și un șir de caractere (culoarea cubului).

Să se construiască un turn de înălțime maximă astfel încât peste un cub cu latura L și culoarea K se poate așeza doar un cub cu latura mai mică strict decât L și culoare diferită de K. În fișierul “turn.txt” să se afișeze componența turnului de la bază spre vârf, pe câte un rând latura și culoarea cubului, apoi la final să se afișeze înălțimea totală a turnului.

# 7. Activity Selection Problem

*You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.*

**Example 1 :** Consider the following 3 activities sorted byby finish time. start[] = {10, 12, 20}; finish[] = {20, 25, 30};A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [ These are indexes in

start[] and finish[] ]**Example 2 :** Consider the following 6 activities sorted by by finish time. start[] = {1, 3, 0, 5, 8, 5}; finish[] = {2, 4, 6, 7, 9, 9};A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [ These are indexes in start[] and finish[] ]

# 8. Counting change using Greedy

Imagine you're a vending machine. Someone gives you £1 and buys a drink for £0.70p. There's no 30p coin in pound sterling, how do you calculate how much change to return?

For reference, this is the denomination of each coin in the UK: 1p 2p 5p 10p 20p 50p 1pound

The greedy algorithm starts from the highest denomination and works backwards. Our algorithm starts at £1. £1 is more than 30p, so it can't use it. It does this for 50p. It reaches 20p. 20p < 30p, so it takes 1 20p.

The algorithm needs to return change of 10p. It tries 20p again, but 20p > 10p. It next goes to 10p. It chooses 1 10p, and now our return is 0 we stop the algorithm.

We return 1x20p and 1x10p.

# 9. Job Sequencing Problem

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes the single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

```
Input: Four Jobs with following deadlines and profitsJobID Deadline Profit a 4 20 b 1 10c 1 40 d 1 30Output: Following is maximum profit sequence of jobs c, a
```

```
Input: Five Jobs with followingdeadlines and profitsJobID Deadline Profit a 2 100 b 1 19c 2 27d 1 25e 3 15Output: Following is maximum profit sequence of jobs c, a, e
```

# 10. Minimum product subset of an array

Given an array a, we have to find minimum product possible with the subset of elements present in the array. The minimum product can be single element also.

```
Input : a[] = { -1, -1, -2, 4, 3 }Output : -24Explanation : Minimum product will be ( -2 * -1 * -1 * 4 * 3 ) = -24
```

```
Input : a[] = { -1, 0 }Output : -1Explanation : -1(single element) is minimum product possible Input : a[] = { 0, 0, 0 }Output : 0
```

# 11. Minimum increment/decrement to make array non-Increasing

Given an array a, your task is to convert it into a non-increasing form such that we can either increment or decrement the array value by 1 in minimum changes possible.

```
Input : a[] = {3, 1, 2, 1}Output : 1Explanation :We can convert the array into 3 1 1 1 bychanging 3rd element of array i.e. 2 into its previous integer 1 in one stephence only one step is required.
```

```
Input : a[] = {3, 1, 5, 1}Output : 4We need to decrease 5 to 1 to make array sortedin non-increasing order.
```

```
Input : a[] = {1, 5, 5, 5}Output : 4We need to increase 1 to 5.
```

# 12. Fractional Knapsack Problem

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

***Input:***
***Items as (value, weight) pairs***

***arr[] = {{60, 10}, {100, 20}, {120, 30}} Knapsack Capacity, W = 50;***

***Output:***

*Maximum possible value = 240*
*by taking items of weight 10 and 20 kg and 2/3 fraction*
*of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240*