*arg$min$ * arg$max$

$\lceil \rceil \lfloor \rfloor$

# 1 Greedy algorithms

## 1.1 Activity selection problem

We would like to book as many pairwise disjoint activities as possible for a seminar/lecture room. Each activity is characterized by two features : its starting time and its finishing time. Let us write the $i$-th activity as $\mathrm{a}_i = [s_i\ f_i[$ with $s_i$ and $f_i$ its starting and finishing time, respectively. Let us also assume that activities are sorted such that $f_1 \leq f_2 \leq \ldots \leq f_n$.

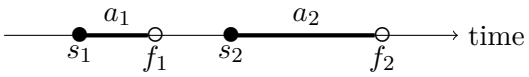



Figure 1: Example of two disjoint activities

### 1.1.1 Dynamic programming solution

**Can we divide this into subproblems ?**

Let us define $\mathrm{S}_{ij} = \{a_k \,|\, a_k \subseteq [f_i\ s_j[\}$ and $c_{ij}$ as the maximum number of disjoint activities we can book in $[f_i\ \ s_j[$ from activities in $S_{ij}$.

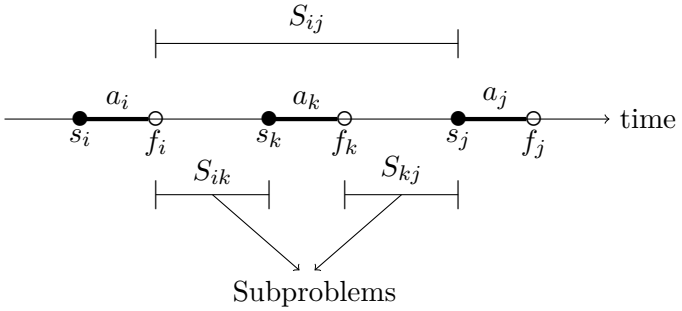



Figure 2: Decomposition in subproblems

From Figure 2, it is pretty easy to see that $\mathrm{c}_{ij} = \begin{cases} \max\limits_{a_k \in S_{ij}} (c_{ik} + c_{kj} + 1) & \text{if } S_{ij} \neq \emptyset \\ 0 & \text{if } S_{ij} = \emptyset\ . \end{cases}$

This decomposition in subproblems tells us that there exists a dynamic programming algorithm to solve the activity selection problem.

### 1.1.2 Greedy solution

However, we can be smarter than that ! Indeed, we can know which activity to choose in advance : the one that finishes first among $S_{ij}$, i.e. the one with smalllest index in $S_{ij}$

(since $f_1 \leq f_2 \leq \ldots \leq f_n$).

### Why ?

If we have an optimal solution (for the subproblem $c_{ij}$) that doesn't use $a_{k_{\min}}$ (the activity in $S_{ij}$ with earliest finish) but starts with $a_k \in S_{ij}$ instead, then we can get a solution just as good (same cardinality $c_{ij}$) by replacing $a_k$ with $a_{k_{\min}}$. This is illustrated on Figure 3.

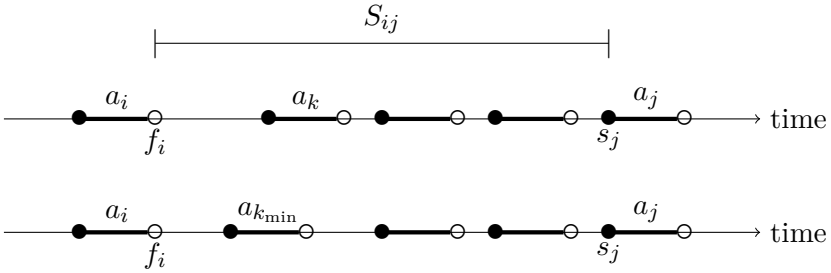



Figure 3: Illustration of the greedy algorithm argument

Consequently, a greedy algorithm can be designed to take advantage of that fact. That greedy algorithm is the following :
`ActivitySelect`($\{ a_1, \ldots, a_n\}$) := $\{a_1\} \bigcup$ `ActivitySelect`({activities disjoint from $a_1$}).

The time complexity of this algorithm is $\overbrace{\Theta(n \log n)}^{\text{Sort}} + \Theta(n) = \Theta(n \log n)$ . On the other hand, dynamic programming would cost $\mathcal{O}(n^3)$. We can see a big difference in terms of time complexity between the greedy approach and the dynamic programming approach for the activity selection problem.

"Greedy" : we build up a solution from $\emptyset$ by adding at each step the "best" piece : in a short-sighted way : we don't care for the future.

A greedy approach can sometimes be adopted in dynamic programming situations where the **principle of suboptimality** applies. However, a greedy choice does not always exist.

**Other examples of greedy algorithms**

- Minimum spanning tree : Kruskal and Prim algorithms
- Shortest path : Dijkstra algorithm

## 1.2 Scheduling problem

Let us now consider another famous problem:

**Scheduling problem**

- $n$ jobs, each having a one unit of time duration
- $t_i$ deadline for the job $i$
- $c_i$ : profit for job $i$

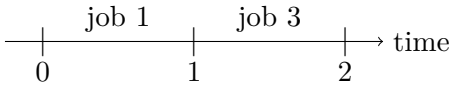
It is important to say that we earn $c_i$ from job $i$ if and only if it is scheduled to finish before its deadline $t_i$. The aim is to maximize the total profit $P_{Total}$, being the sum of the individual profits of the jobs finishing before their respective deadline.

### 1.2.1 Example

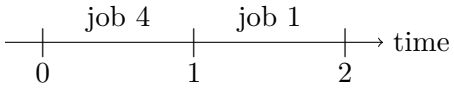
Let's take the following instance

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $c_i$ | 50 | 10 | 15 | 30 |
| $t_i$ | 2 | 1 | 2 | 1 |

One possible scheduling of the jobs is the following




In that particular case, the total profit will simply be $P_{Total} = c_1 + c_3 = 50 + 15 = 65$ . Another possible scheduling of the jobs is




We have now improved the total profit: $P_{Total} = c_1 + c_4 = 50 + 30 = 80$ . This is in fact the optimal solution. Indeed, if we schedule first job 4, and then job 1, we obtain the best solution since they are the jobs with highest individual profits, and furthermore it is impossible to schedule more than 2 jobs, then it must be the best combination.

### 1.2.2 Generalization: Greedy Algorithm

The greedy algorithm is very simple:

```
L = Empty list (will hold selected jobs)
S = stack of jobs ordered by profit with highest on top.
While S not Empty
    a = S.pop()
    Try add a in L at the farthest time index within deadlines
```

The list L has length $T$ where $T$ is the time horizon considered. The $i$-th element of L is the job being scheduled from $t = i - 1$ to $t = i$, or is empty if no job is scheduled at that time slot. Note that this algorithm works if $t_i \in \mathbb{N}\ \forall i$ but doesn't work if there exist a $t_i$ non-integral. Let's see how it works on our example (in this case $T = 2$):

```
Select job with highest profit : job1
    Try add job1 in L -> possible
    $ L ={/, job1} : Profit of 50
Select job with 2nd highest profit : job4
    Try add job4 in L -> possible
    $ L ={job4, job1} : Profit of 80
Select job with 3rd highest profit : job3
    Try add job3 in L -> impossible
    $ Do not add it
Select with 4th highest profit : job2
    Try add job2 in L -> impossible
    $ Do not add it
STOP
```

**Does the greedy algorithm work ? How to check that a set of jobs is feasible ?**

**Theorem 1.** *A set of jobs with deadlines $t_1 \leq t_2 \leq\leq t_n$ is feasible, if and only if, the order $1, 2, \ldots, n$ is feasible.*

*Proof.*

- ($\Rightarrow$) If the schedule


  is feasible, with $t_i \leq t_j$, then we can swap $i$ and $j$, and it's still a feasible schedule. Indeed, since this schedule is feasible, we have $t_i \geq b$ and $t_j \geq a$. Furthermore, we have that $b \geq a$ and $t_j \geq t_i$. It follows that $t_i \geq b \geq a$ and $t_j \geq t_i \geq b$, which make the schedule where $i$ and $j$ are swapped feasible. We can repeat swaps as many times as necessary $\Rightarrow$ order $1, 2, , n$ feasible.

- ($\Leftarrow$) Trivial.

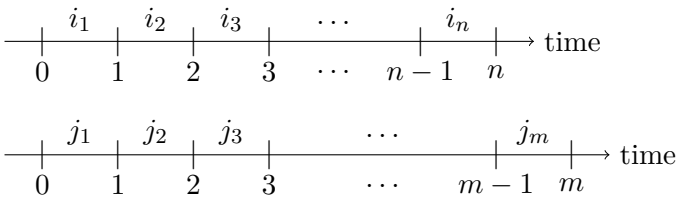
□

Note that if the jobs are already ordered by increasing deadlines, then checking feasibility of the schedule is done in $\Theta(n)$.

**Theorem 2.** *The greedy algorithm provides the optimal scheduling.*

*Proof.* Suppose that $I = \{i_1, i_2, , i_n\}$ is the greedy solution and $J = \{j_1, j_2, \ldots, j_m\}$ is the optimal solution. $\Rightarrow$ Both feasible $\Rightarrow$ can be ordered at unit time slots from the left

We want to "align" $I$ and $J$ as much as possible to compare them: schedule same jobs at the same time.



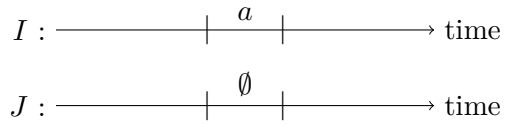

**Example**

Suppose that job $k \in I \bigcap J$, e.g. $k = i_4 = j_8$, then job $k$ is scheduled at the 4-th time slot in $I$ and at the 8-th time slot in $J$. We want to swap jobs either in $I$ or in $J$ in order to have job $k$ scheduled at the same time slot in both schedules.

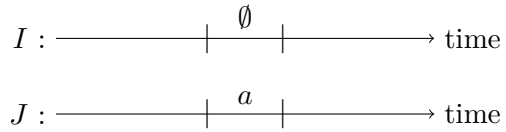
To that end, we will swap $i_4$ and $i_8$ in $I$. If $i_8$ is empty, then after the swap is performed, $i_4$ will be empty and $i_8$ will contain job $k$. If $i_8 = \tilde{k}$ was not empty, then after the swap has been performed, job $\tilde{k}$ is now scheduled on the 4-th time slot and job $k$ on the 8-th. Moving job $\tilde{k}$ from the 8-th time slot to the 4-th does not put the feasibility of the new schedule in jeopardy, since it is now scheduled earlier. Furthermore, scheduling the job $k$ at the 8-th time slot instead of the 4-th is also feasible since job $k$ is scheduled at that 8-th time slot in $J$, and $J$ is assumed to be feasible.

Could we have swapped $j_8$ and $j_4$ in $J$ instead of swapping $i_4$ and $i_8$ in $I$? The answer is : it depends, but in general **NO** ! Indeed, although scheduling job $k$ at the 4-th time slot instead of at the 8-th in $J$ would always be feasible, there is no guarantee that the job that was initially in $j_4$ would still be feasible at the 8-th time slot, therefore preventing the "swapped" schedule to be feasible !

Repeat all swaps until no more is possible, then all common jobs are aligned. Now, let us take a look at all the situations one could think of where the jobs are different in $I$ and in $J$. The first situation looks like

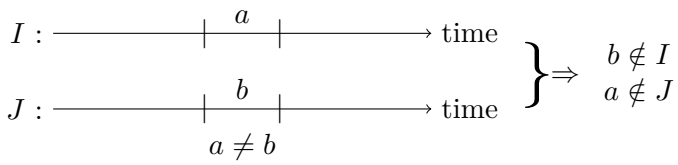



This first situation is impossible because since $a \notin J$, we can schedule it in the empty slot and improve the total profit of the sequence $J$ strictly. A second situtation is




Once again, this situation is impossible because since $a \notin I$, then adding $a$ to $I$ is feasible $\Rightarrow$ the greedy algorithm would have chosen it. Finally, the last situation that could potentially occur is

If $c_b > c_a$ : then the greedy algorithm would have chosen $b$ before $a \Rightarrow b$ would be in $I$.
If $c_a > c_b$ : then $J$ is strictly improved by replacing $b$ with $a$, then impossible case.




It follows that $c_a = c_b$, and consequently we have that $C_I = C_J$. □

Time complexity $\underbrace{\Theta(n \log n)}_{\text{Sort by profit}} + \underbrace{\Theta(n^2 \log n)}_{\text{Sort by deadline } n \text{ times}}$

But we can do better : $\Theta(n \log n)$