Greedy Algorithms: Activity Selection

Chapter 16.1

Greedy Algorithms

- For many optimization problems, using dynamic programming to make choices is overkill.
- Sometimes, the correct choice is the one that appears "best" at the moment.
- Greedy algorithms make these locally best choices in the hope (or knowledge) that this will lead to a globally optimum solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do. (The same can be said of dynamic programming.)

Activity Selection

- We have a collection $S = \{a_1, a_2, \ldots, a_n\}$ of activities that all want to use a common resource which can only be used by one activity at a time (e.g. a TV camera).
- Each activity a_i has a given start time s_i and finish time f_i.
- Our problem is to select a maximum set of activities that can use the resource. These activities must not overlap in time.

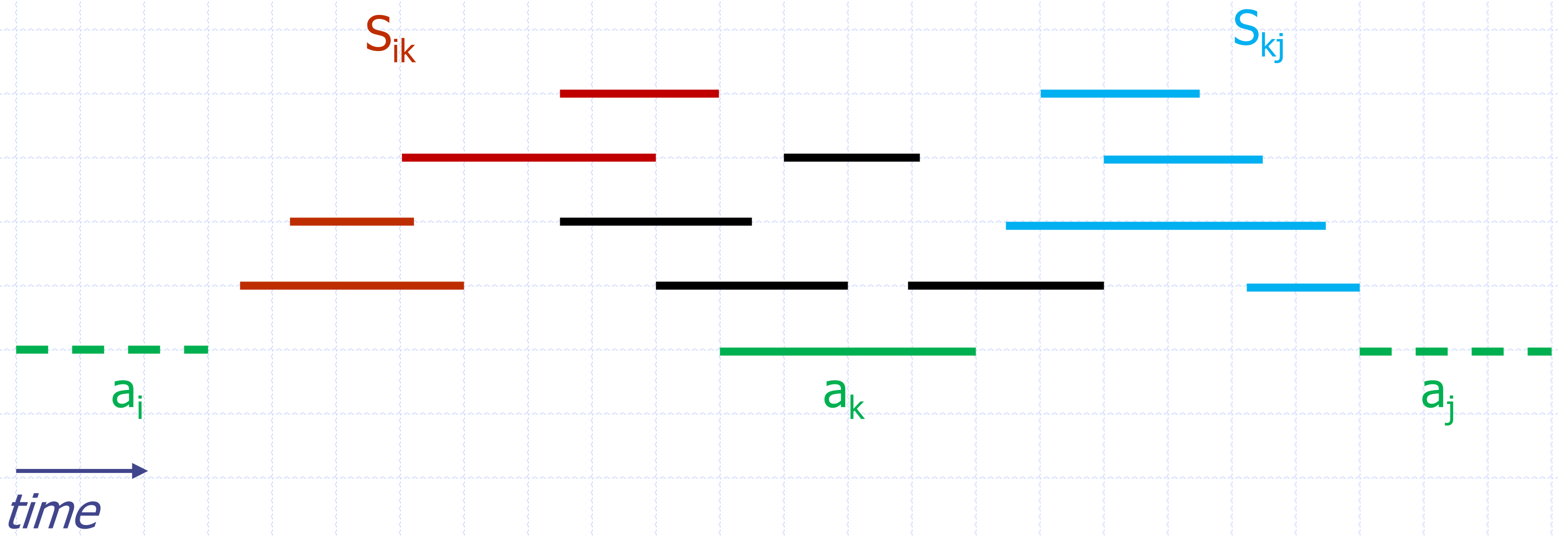
i	1	2	3	4	5	6	7	8	9
s_i	3	0	2	3	5	8	9	8	12
f_i	4	6	7	8	9	10	12	13	15

Activity Selection

- We start with a DP solution for the problem.
- Let $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$ be the set of activities which can use the resource between activity i and activity j.
- Add sentinel activities a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$ to S.
- First assume activities are sorted by increasing order of finish time. (This requires a sort – O(n log n) time – if activities are not given in this order.)
- Then $S_{ij} = \varnothing$ when $i \ge j$.

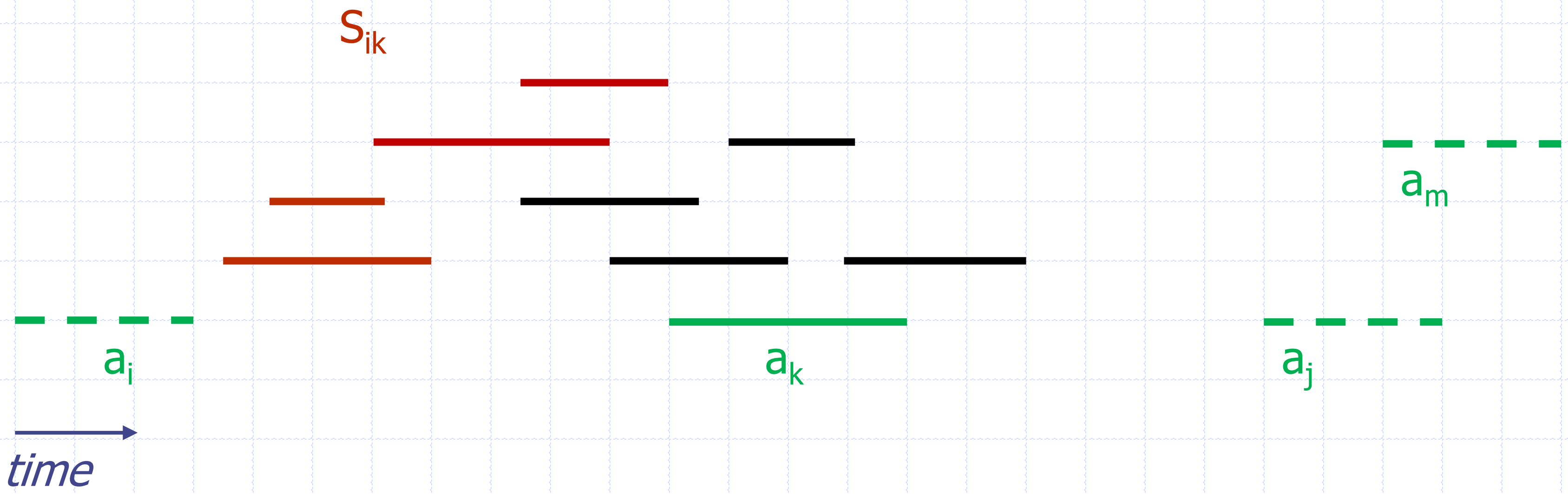
Optimal substructure

- We consider S_{ij} as a subproblem: find a maximal set of nonoverlapping activities in the set S_{ij}.
- Suppose an optimal solution to S_{ij} includes activity a_k. Then it must also include an optimal solution for S_{ik} and S_{kj}.



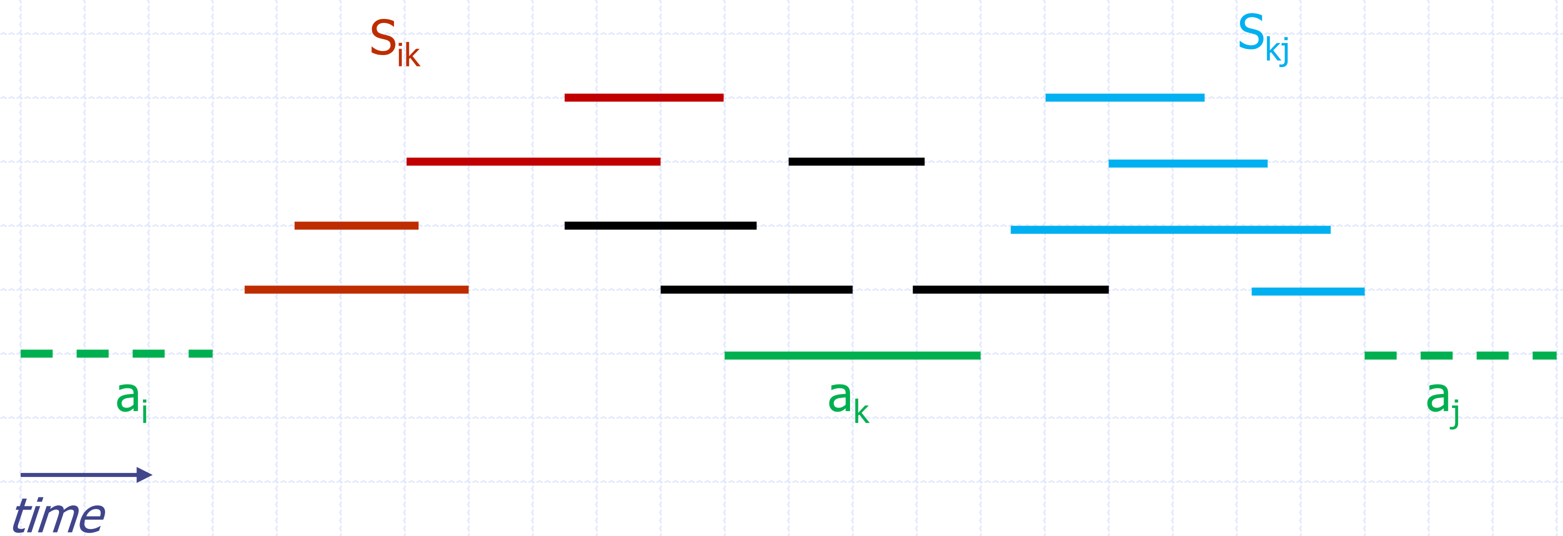
Overlapping subproblems

- Consider subproblem S_{ij} and S_{im} where the start time of m is greater than the start time of j.
- Let a_k be in S_{ij} – then it is also in S_{im}.
- If a_k is chosen in S_{ij}, it generates subproblem S_{ik}.
- If a_k is chosen in S_{im}, it also generates subproblem S_{ik}.



Recursive Solution

- Let c(i, j) be the maximum number of activities in a solution to subproblem S_{ij}.
- c(i, j)=0 when $S_{ij} = \emptyset$.   In particular, c(i, j) = 0 for $i \geq j$.
- If a_k is used in optimal solution to S_{ij}, then
  $c(i, j) = c(i, k) + 1 + c(k, j)$



Recursive Solution

- So we try this over all possible a_k:

$$c(i,j) = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c(i,k) + c(k,j) + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

A memoization or dynamic programming solution based on this will run in $O(n^3)$ time (there are $O(n^2)$ table entries to compute, and each one takes linear time).

Write out the solution and analysis if you didn't follow that.

On closer inspection…

Theorem:

Consider any nonempty subproblem S_{ij} and let a_m be the activity with the earliest finish time:

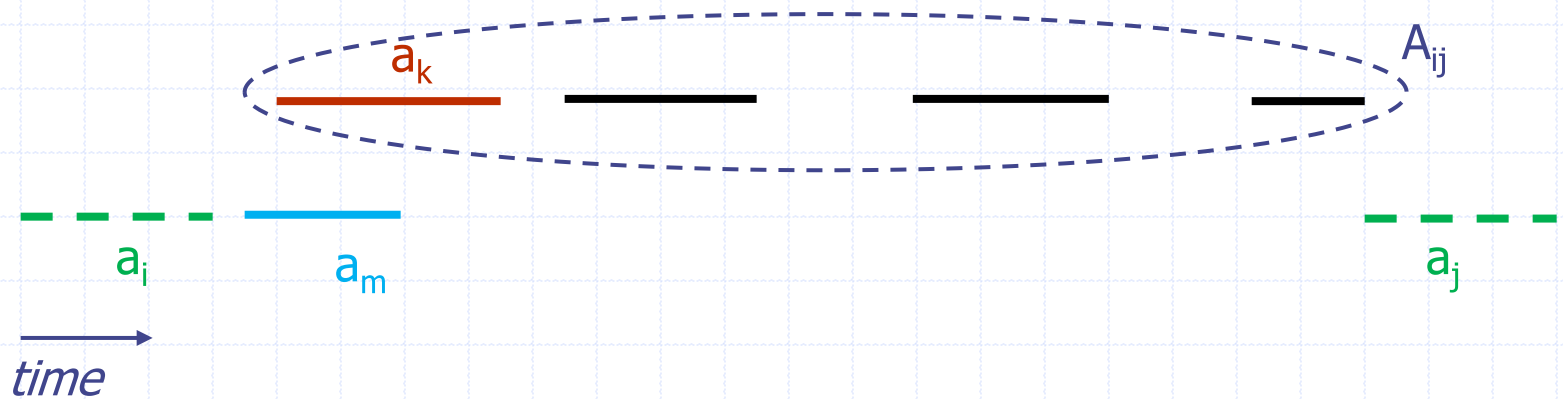
$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

Then

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The suproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be empty.

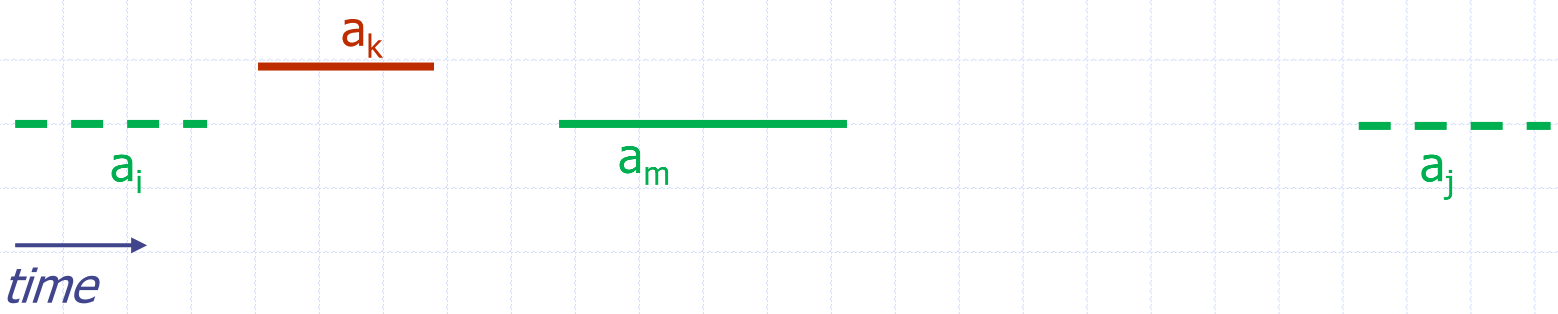
Proof:

(1) Let A_{ij} be a maximum-size subset of mutually compatible activities of S_{ij}. If a_m is in A_{ij}, we are done. If a_m is not in A_{ij}, let a_k be the activity of A_{ij} that is first (has first finish time). Since a_m finishes at or before any other activity in S_{ij}, it finishes before a_k. Therefore $A_{ij}-a_k+a_m$ is compatible, and it is a maximum-size subset of S_{ij}.



Proof:

(2) By contradiction. Suppose that S_{im} is nonempty – there is an activity a_k with $f_i \le s_k < f_k \le s_m < f_m$. Then a_k is also in S_{ij} and $f_k < f_m$, which contradicts our choice of a_m.



Reducing the substructure

The theorem reduces the choices and the recursive computation necessary in the DP solution.

The choices of activity to include is reduced to one: the activity with the earliest finish time.

The number of subproblems that we must consider in solving any subproblem is reduced from two to one: only S_{mj} is considered, as S_{im} is empty.

The form of the subproblems considered is reduced from S_{ij} to $S_{i,n+1}$. (We don't have to consider arbitrary j.)

Recursive solution

Given: arrays s[] and f[] with start and finish times, sorted to be increasing by finish time.

Start by calling RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, n)
m = i + 1
while m ≤ n and s[m] < f[i]                // find first activity
    m = m + 1

if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else
    return ∅
```

Recursive solution analysis

start: R-A-S(s, f, 0, n)

```
R-A-S(s, f, i, n)
m = i + 1
while m ≤ n and s[m] < f[i]
    m = m + 1

if m ≤ n
    return {a_m} ∪ R-A-S(s, f, m, n)
else
    return ∅
```

Over **all** calls, m starts at 1 and increases up to n+1.
Therefore the while executes O(n) times, taking O(1) time per execution.

The total time for the algorithm is the total time for the while loop plus O(1) per call. Since m (and i) increases by one each call, and is capped at n+1, there are O(n) calls. So total time is

O(n) loops * O(1) time/loop [while]

+ O(n) calls * O(1) time/call

= O(n) time overall.

Iterative solution

Given: arrays s[] and f[] with start and finish times, sorted to be increasing by finish time.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
A = {a_1}
lastSelected = 1

for m = 2 to n
    if s[m] ≥ f[lastSelected]
        A = A ∪ {a_m}
        lastSelected = m

return A
```

Iterative solution analysis

GREEDY-ACTIVITY-SELECTOR(s, f, n)	
$A = \{a_1\}$	O(1)
lastSelected = 1	O(1)
for m = 2 to n	O(n) iterations
if s[m] $\geq$ f[lastSelected]	O(1) per iteration
$A = A \cup \{a_m\}$	
lastSelected = m	
return A	O(1)
	O(n) time total

Reminder: both recursive and iterative solutions are O(n log n) if you need to sort.

Greedy can be tricky

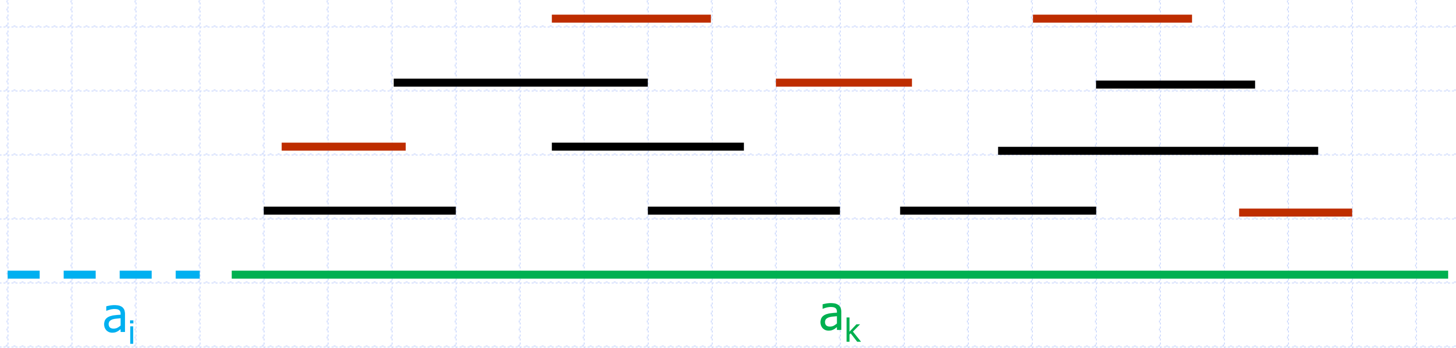
Our greedy solution used the activity with the earliest finish time from all those activities that did not conflict with the activities already chosen.

Other greedy approaches may not give optimum solutions to the problem, so we have to be clever in our choice of greedy strategy and prove that we get the optimum solution.

Here are some other greedy strategies which we could have tried: (S1) choose the activity with the earliest start time from all activities that do not conflict with the activities already chosen.

Earliest start time

Unfortunately, the activity with the earliest start time could also have the latest end time.

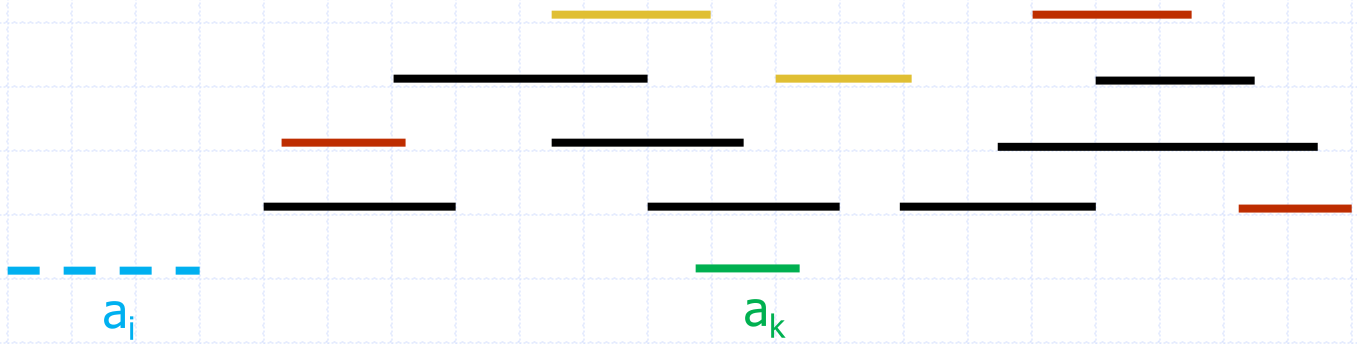


Here the activities shown in red would have been a better choice.

(S2) choose the activity with the least duration from all activities that do not conflict with already chosen activities.

Least duration

Unfortunately, the activity with the least duration could conflict with two activities from a maximal set.

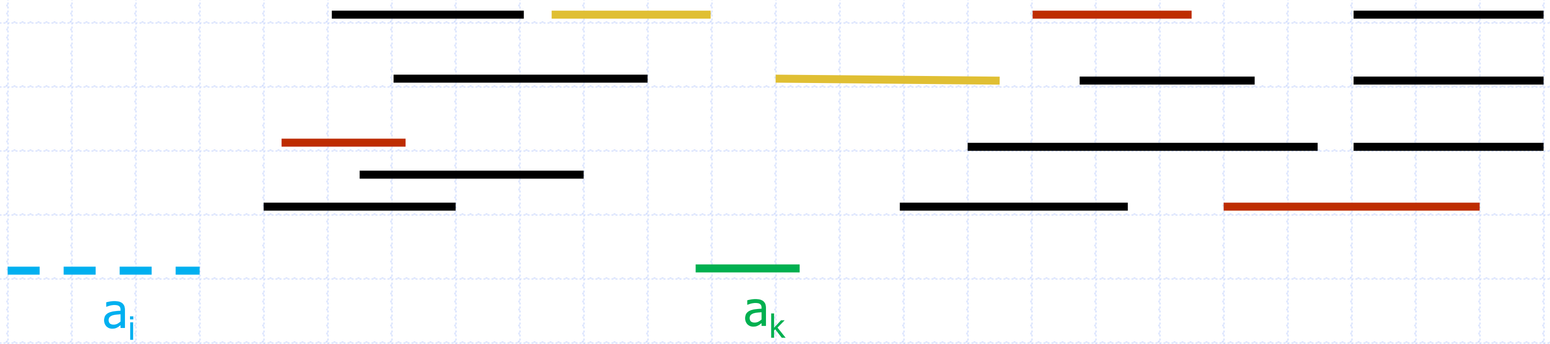


Here the activities shown in red and yellow would have been a better choice than red and green.

(S3) choose the activity with the fewest overlaps from all activities that do not conflict with already chosen activities.

Fewest overlaps

The activity with the fewest overlaps could also conflict with two activities from a maximal set.



Here the activities shown in red and yellow would have been a better choice than anything including a_k.

(S4) choose the activity with the latest start time from all activities that do not conflict with already chosen activities.

Latest start time

Choosing the activity with the latest start time is a greedy strategy that will lead to a maximum set of nonoverlapping activities.

It's actually the time-reversal of the earliest finish time strategy.

(S5) choose the activity with the latest finish time from all activities that do not conflict with already chosen activities.

Latest finish time

Choosing the activity with the latest finish time doesn't work—it's a time-reversal of the earliest start time approach.

The point is that there are often many different greedy strategies to try. Sometimes when one doesn't work, another one will, so don't necessarily give up.   And be sure to prove that the strategy you choose works!

When it does not obtain an optimal solution, the greedy approach is known as a heuristic. Sometimes, a heuristic solution is an approximation to an optimal one. Sometimes, not.