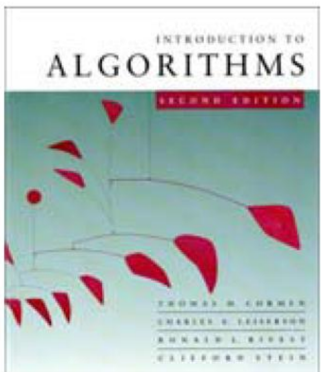
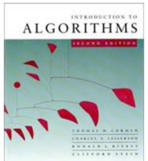
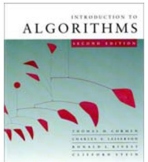


Greedy Algorithm



Introduction

- **Similar to dynamic programming.**
- **Used for optimization problems.**
- **Not always yield an optimal solution.**
- **Make choice for the one looks best right now.**
- **Make a locally optimal choice in hope of getting a globally optimal solution.**



Activity selection

n activities require exclusive use of a common resource.

For example, scheduling the use of a classroom.

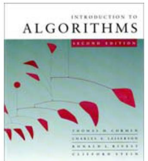
Set of activities $S = \{a_1, \ldots, a_n\}$.

a_1 needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i = start time and f_i = finish time.

Goal: Select the largest possible set of nonoverlapping (mutually compatible) activities.

Note: Could have many other objectives:

- Schedule room for longest time.
- Maximize income rental fees.



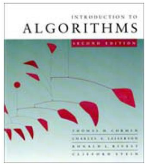
Activity selection

Example: S sorted by finish time

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Maximum-size mutually compatible set: $\{a_1, a_4, a_8, a_{11}\}$.

Not unique: also $\{a_2, a_4, a_9, a_{11}\}$.

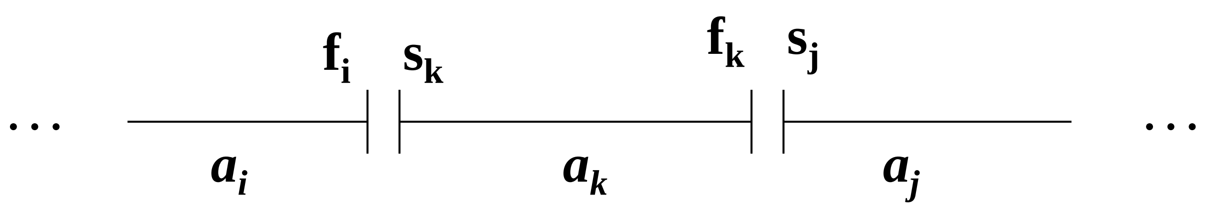


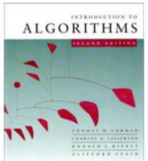
Activity selection

Optimal substructure of activity selection

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

= activities that start after a_i finishes and finish before a_j starts.





Activity selection

Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

To represent the entire problem, add fictitious activities:

$a_0 = [-\infty, 0)$

$a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$

we don’t care about $-\infty$ and “$\infty + 1$”.

Then $S = S_{0,n+1}$.

Range for S_{ij} is $0 \leq i, j \leq n+1$.

Activity selection

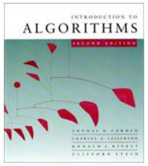
Assume that activities are sorted by monotonically increasing finish time:

$f_0 \le f_1 \le f_2 \le \cdots \le f_n \le f_{n+1}$.

Then $i \ge j \Rightarrow S_{ij} = \emptyset$.

- If there exists $a_k \in S_{ij}$:
  $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$.
- But $i \ge j \Rightarrow f_i \ge f_j$. Contradiction.

So only need to worry about S_{ij} with $0 \le i < j \le n + 1$.
All other S_{ij} are $\emptyset$.



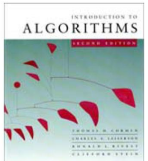
Activity selection

Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:

- S_{ik} (start after a_i finishes, finish before a_k starts)
- S_{kj} (start after a_k finishes, finish before a a_j starts)

Solution to S_{ij} is (solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (solution to S_{kj}).
Since a_k is in neither subproblem, and the subproblems are disjoint,

|solution to S| = |solution to S_{ik} | + 1 + |solution to S_{kj} | .



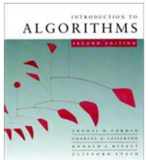
Activity selection

If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well.

Use the usual cut-and-paste argument.

Let A_{ij} = optimal solution to S_{ij} .
So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, assuming:

- S_{ij} is nonempty, and
- we know a_k.



Activity selection

Recursive solution to activity selection

$c[i, j]$=size of maximum-size subset of mutually compatible activities in S_{ij}.

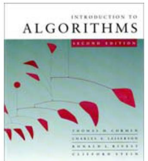
- $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0.$

If $S_{ij} \neq \emptyset$, suppose we know that a_k is in the subset. Then

$c[i, j] = c[i, k] + 1 + c[k, j].$

But of course we don't know which k to use, and so

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

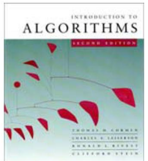


Activity selection

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij} .
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

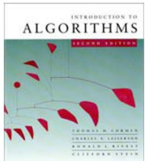


Activity selection

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$. Then $a_k \in S_{ij}$ and it has an earlier finish than f_m , which contradicts our choice of a_m. Therefore, there is no $a_k \in S_{im} \Rightarrow S_{im} = \phi$.

1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij}.
   Order activities in A_{ij} in monotonically increasing order of finish time.
   Let a_k be the first activity in A_{ij}.
   If $a_k = a_m$, done (a_m is used in a maximum-size subset).
   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m)



Activity selection

Claim

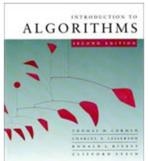
Activities in A'_{ij} are disjoint.

Proof Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \leq f_k$ (so a_m doesn't overlap with anything else in A'_{ij}).

■ (claim)

Since $|A'_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}.

■ (theorem)



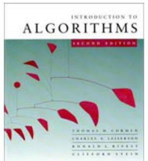
Activity selection

This is great:

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j - i - 1$	1

Now we can solve *top down*:

- To solve a problem S_{ij},
  - Choose $a_m \in S_{ij}$ with earliest finish time: the ***greedy choice***.
  - Then solve S_{mj}.



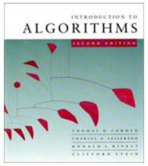
Activity selection

What are the subproblems?

- Original problem is $S_{0,n+1}$.
- Suppose our first choice is a_{m_1} .
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is a_{m_2} .
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

Each subproblem is $S_{m_i,n+1}$, i.e., the last activities to finish.
And the subproblems chosen have finish times that increase.

Therefore, we can consider each activity just once, in monotonically increasing order of finish time.

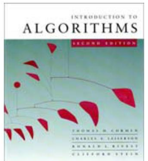


Activity selection

Easy recursive algorithm:

Assumes activities already sorted by monotonically increasing finish time. (If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:

REC-ACTIVITY-SELECTOR(s, f, i, n)
$m \leftarrow i + 1$
while $m \le n$ and $s_m < f_i$ $\qquad \triangleright$ Find first activity in $S_{i,n+1}$.
    do $m \leftarrow m + 1$
If $m \le n$
    then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
    else return $\emptyset$



Activity selection

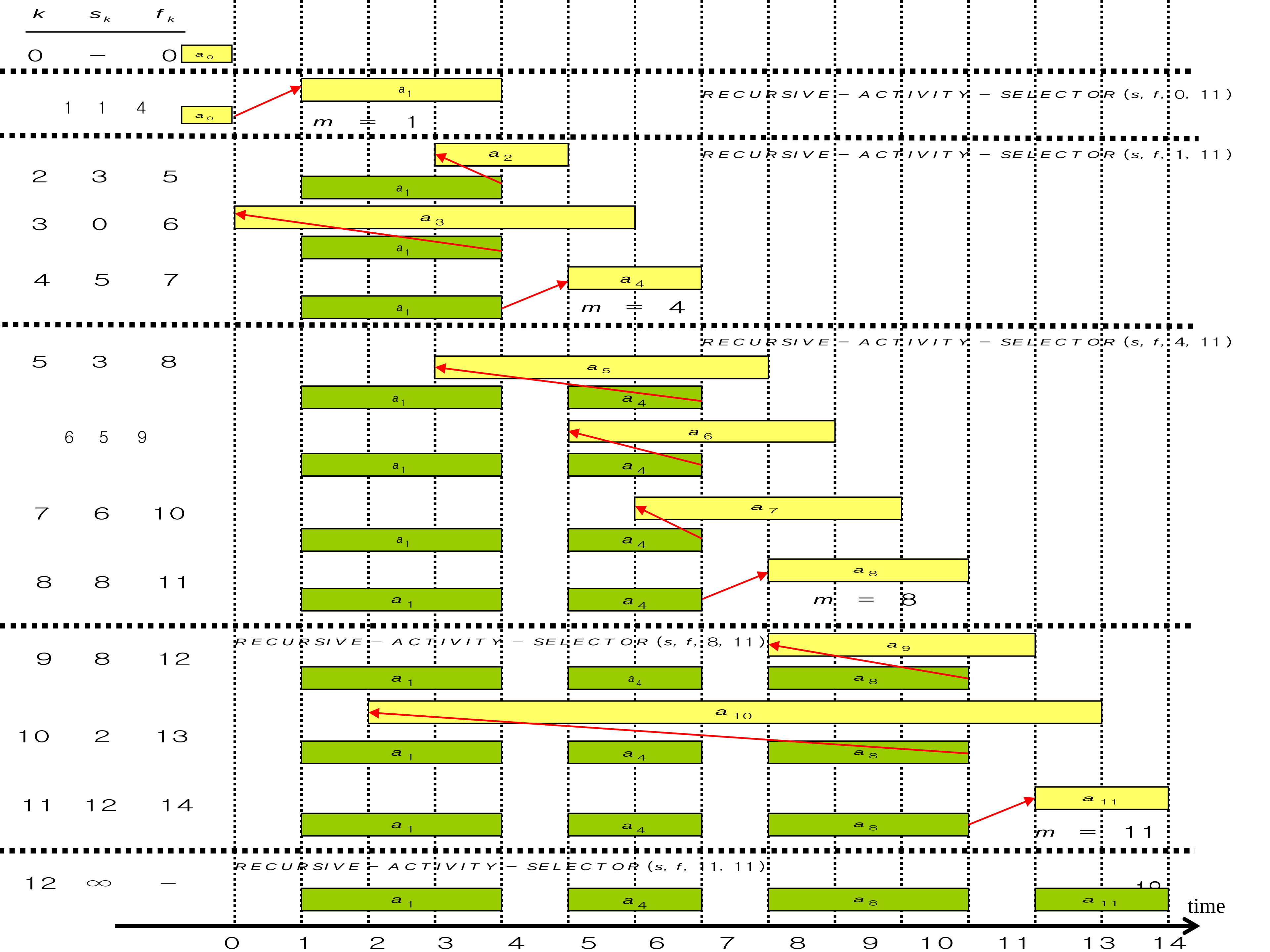
Initial call: REC-ACTIVITY-SELECTOR(s, f, 0, n)

Idea: The **while** loop checks a_{i+1}, a_{i+2}, …, a_n until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

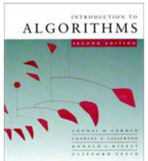
- If the loop terminates because a_m is found ($m \leq$ n), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$ – each activity examined exactly once.



k
s_k
f_k
0 — 0
a_0
1 1 4
a_1
m = 1
RECURSIVE − ACTIVITY − SELECTOR (s, f, 0, 11)
2 3 5
a_2
RECURSIVE − ACTIVITY − SELECTOR (s, f, 1, 11)
3 0 6
a_3
4 5 7
a_4
m = 4
RECURSIVE − ACTIVITY − SELECTOR (s, f, 4, 11)
5 3 8
a_5
6 5 9
a_6
7 6 10
a_7
8 8 11
a_8
m = 8
RECURSIVE − ACTIVITY − SELECTOR (s, f, 8, 11)
9 8 12
a_9
10 2 13
a_10
11 12 14
a_11
m = 11
RECURSIVE − ACTIVITY − SELECTOR (s, f, 11, 11)
12 ∞ —
time
0 1 2 3 4 5 6 7 8 9 10 11 13 14



Activity selection

Can make this *iterative*.

GREEDY-ACTIVITY-SELECTOR(s, f, n)

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for $m \leftarrow 2$ **to** n

    do if $s_m \geq f_i$

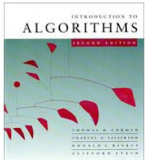
        then $A \leftarrow A \cup \{a_m\}$

            $i \leftarrow m$ $\triangleright$ a_i is most recent addition to A.

return A

Go through example given earlier. Should again get $\{a_1, a_4, a_8, a_{11}\}$.

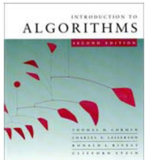
Time: $\Theta(n)$.



Greedy strategy

The choice that seems best at the moment is the one we go with. What did we do for activity selection?

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
4. Show that all but one of the subproblems resulting from the greedy choice are empty.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.



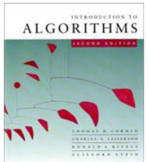
Greedy strategy

Typical streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.

No general way to tell if a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.



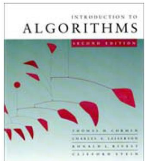
Greedy strategy

Dynamic programming:

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up*.

Greedy:

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*



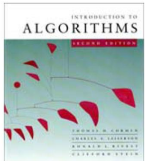
Greedy strategy

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem:

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq$ W.
- Have to either take an item or not take it – can't take part of it.

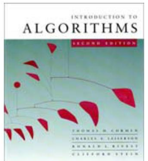


Greedy strategy

Fractional knapsack problem: Like the 0-1 kanpsack problem, but can take fraction of an item.

Both have optimal substructure.
But the fractional kanpsack problem has the greedy-choice property, and the 0-1 knapsack problem does not have greedy-choice that returns optimal solution.

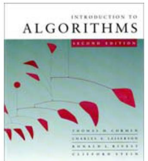
To solve the fractional problem, rank items by value/weight: v_i/w_i.
Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.



Greedy strategy

Fractional knapsack (v, w, W)
load $\leftarrow$ 0
$i \leftarrow 1$
while *load* $\leq$ W and $i \leq n$
  do if $w_i \leq$ W – *load*
    then take all of item i
    else take (W – *load*)/w_i of item i
   add what was taken to *load*
   $i \leftarrow i + 1$ $\triangleright$ *move to the next valuable item.*

Taking the items in order of greatest value per pound yields an optimal solution.



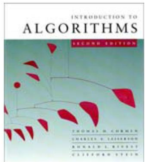
Greedy strategy

Time: O (n lg n) to sort, O (n) thereafter

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

W = 50



Greedy strategy

Greedy solution:

- Take items 1 and 2.
- value = 160, weight = 30.

Have 20 pounds of capacity left over.

- 0-1 knapsack problem can't be solved w/ greedy strategy.
  (i.e., the optimal solution can't be obtained w/ greedy strategy)
- Fraction knapsack problem can be solved w/ greedy strategy by adding 20 pounds (2/3 of item 3) $\times 4$ = \$80 value.

Optimal solution (for 0-1 knapsack w/o greedy strategy):

- Take items 2 and 3.
- value = 220, weight = 50.

No leftover capacity.