Greedy Algorithms

- Given a set of activities (s_i, f_i), we want to schedule the maximum number of non-overlapping activities.
- GREEDY-ACTIVITY-SELECTOR (s, f)

```
n = length[s]
S = {a_1}
i = 1
for m = 2 to n do
        if s_m is not before f_i then
                S = S U {a_m}
                i = m
return S
```

Dynamic Programming

- Activity Problem Revisited: Given a set of n activities $a_i = (s_i, f_i)$, we want to schedule the maximum number of non-overlapping activities.
- New Approach:
  - Observation: To solve the problem on activities $A_n = \{a_1,\ldots,a_n\}$, we notice that either
    - optimal solution does not include a_n (Problem on A_{n-1})
    - optimal solution includes a_n (Problem on A_k, which is equal to A_n without activities that overlap a_n, I.e., a_k is the last activity that finishes before a_n starts.)

An efficient implementation

- Why not solve the problem on $A_1,\ldots,A_{n-1},A_n$?
- In what order to solve them?
- Is the problem on A_1 easy?
  - YES, trivial
- Can the optimal solutions to the problems on $A_1,\ldots,A_i$ help to solve the problem on A_{i+1}?
  - YES! Either:
    - optimal solution does not include a_{i+1} (Problem on A_i)
    - optimal solution includes a_{i+1} (you are left with a problem on A_k, which is equal to A_i without activities that overlap a_{i+1}, i.e., a_k is the last activity that finishes before a_{i+1} starts.)

Dynamic Programmming: Activity Selection

- Select the maximum number of non-overlapping activities from a set of n activities $A = \{a_1, \ldots, a_n\}$ (sorted by finish times).
- Identify "easier" subproblems to solve.

  $A_1 = \{a_1\}$

  $A_2 = \{a_1, a_2\}$

  $A_3 = \{a_1, a_2, a_3\}, \ldots,$

  $A_n = A$
- Subproblems: Select the max number of non-overlapping activities from A_i

Dynamic Programmming: Activity Selection

- Solving for A_n solves the original problem.
- Solving for A_1 is easy.
- If you have optimal solutions $S_1, \ldots, S_{i-1}$ for subproblems on $A_1, \ldots, A_{i-1}$, how to compute S_i?
- The optimal solution for A_i either
  - Case1: does not include a_i or
  - Case 2: includes a_i
- Case 1:
  - $S_i = S_{i-1}$
- Case 2:
  - $S_i = S_k \cup \{a_i\}$, for some $k < i$.
  - How to find such a k? We know that a_k cannot overlap a_i.

Dynamic Programmming: Activity Selection

- DP-ACTIVITY-SELECTOR (s, f)

```
n = length[s]
N[1] = 1          // number of activities in S_1
F[1] = 1          // last activity in S_1
for i = 2 to n do
    let k be the last activity finished before s_i
    if (N[i-1] > N[k]) then   // Case 1
        N[i] = N[i-1]
        F[i] = F[i-1]
    else   // Case 2
        N[i] = N[k] + 1
        F[i] = i
```

How to output S_n?

Backtrack!

Time Complexity?

O(n lg n)

Dynamic Programming Features

- Identification of subproblems
- Recurrence relation for solution of subproblems
- Overlapping subproblems (sometimes)
- Identification of a hierarchy/ordering of subproblems
- Use of table to store solutions of subproblems (MEMOIZATION)
- Optimal Substructure

Longest Common Subsequence

S_1 = CORIANDER　　　　CORIANDER

S_2 = CREDITORS　　　　CREDITORS

Longest Common Subsequence($S_1[1..9]$, $S_2[1..9]$) = **CRIR**

Subproblems:

- LCS[$S_1[a..b]$, $S_2[c..d]$], for all a, b, c, and d
- LCS[$S_1[1..i]$, $S_2[1..j]$], for all i and j [BETTER]

- Recurrence Relation:
  - LCS[i,j] = LCS[i-1, j-1] + 1, if $S_1[i] = S_2[j]$)

    LCS[i,j] = max { LCS[i-1, j], LCS[i, j-1] }, otherwise
- Table (m X n table)
- Hierarchy of Solutions?

LCS Problem

```
LCS_Length (X, Y )
m ← length[X]
n ← Length[Y]
for i = 1 to m
do c[i, 0] ← 0
for j =1 to n
do c[0,j] ←0
for i = 1 to m
      do for j = 1 to n
            do if ( xi = yj )
                  then c[i, j] ← c[i-1, j-1] + 1
                        b[i, j] ← "↖"
                  else if c[i-1, j] c[i, j-1]
                        then c[i, j] ← c[i-1, j]
                        b[i, j] ← "↑"
                     else
                        c[i, j] ← c[i, j-1]
                        b[i, j] ← "←"
return
```

LCS Example

		H	A	B	I	T	A	T
	0	0	0	0	0	0	0	0
A	0	0↑	1↖	1←	1←	1←	1↖	1←
L	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
P	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
H	0	1↖	1↑	1↑	1↑	1↑	1↑	1↑
A	0	1↑	2↖	2←	2←	2←	2↖	2←
B	0	1↑	2↑	3↖	3←	3←	3←	3←
E	0	1↑	2↑	3↑	3↑	3↑	3↑	3↑
T	0	1↑	2↑	3↑	3↑	4↖	4←	4↖

Dynamic Programming vs. Divide-&-conquer

- Divide-&-conquer works best when all subproblems are independent. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem.
- Dynamic programming is needed when subproblems are dependent; we don't know where to partition the problem.

  For example, let S_1= {ALPHABET}, and S_2 = {HABITAT}.

  Consider the subproblem with S_1' = {ALPH}, S_2' = {HABI}.

  Then, $LCS(S_1', S_2') + LCS(S_1 - S_1', S_2 - S_2') \neq LCS(S_1, S_2)$
- Divide-&-conquer is best suited for the case when no "overlapping subproblems" are encountered.
- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

Dynamic programming vs Greedy

1. Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. The solution comes up when the whole problem appears.

   Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem, then reduce the problem to a smaller one. The solution is obtained when the whole problem disappears.

2. Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy. However, there are some problems that greedy can not solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.