We first went through another dynamic programming example on the board: finding the maximum sum of **contiguous** elements of an array. We saw that dynamic programming can reduce the complexity to O(n).

Example: For array A=[-1, 2, -1, 5, -7] the maximum sum of contiguous elements is 2-1+5=6

Main approach:

Consider array A[1 .. n].

Save subproblem solutions in a table m[1.. n]. For example, m[j] is the optimal solution to the subproblem ending in position j.

The main recursion is:

m[j] = max(m[j-1] + A[j], A[j])

Here m[j-1] + A[j] is the maximum sum contiguous with the previous position j-1 , and A[j] is just the value in position j. So either we add A[j] to the solution ending at j-1 or we just take A[j].

After we compute all entries of the table, we can find the maximum of all entries in the table.

Complexity: O(n). The table is size n and the decision points for each entry of the table (subproblem) is constant (two). So the total runtime is their multiplication and therefore proportional to n.

We went through the example on the board …

Greedy algorithms

Main approach: always make the choice that looks best at the moment.

- More efficient than dynamic programming

- Always make the choice that looks "best" at the moment (just one choice; contrast this with Dynamic Programming in which we check out all possible choices!)

- Caveat: Doesn't always result in globally optimal solution, but for some problems does. We will look at such examples, as well as show cases in which it does not work by finding a counter example.

- We'll want to contrast when to use the different algorithmic approaches for different problems: divide and conquer, dynamic programming, greedy. What are the hallmarks of these different algorithmic approaches and when are they suitable for different problems?

We'll first develop the approach for a problem known as the **activity selection problem** (we'll contrast dynamic programming versus greedy algorithms for this problem!). We'll then later look at a classical example of data compression known as Huffman coding.

Example: activity selection problem:

Goal: We want to allocate activities to a lecture hall, which could only serve one activity at a time. Activities cannot overlap in time. We want to select a maximum subset of activities (as many activities as possible) that are mutually compatible (do not overlap in time). We'll just use compatible for "mutually compatible" from now on.

More formally:

Activities are a set ***S={a1, a2, … , an}***

Start times: ***s1, s2, … sn***

Finish times: ***f1, f2, .. fn***

Compatible means no time overlap: ***si >= fj or sj >= fi***

Assumption: We assume activities have been pre-sorted by finish time such that:

f1 <= f2 <= f3 … <= fn

Example:

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	9	9	10	11	12	14	16

Example compatible activities:

{a3, a9, a11}

{a3, a7, a11}

But this is not the largest subset possible.

Example of largest subset of compatible activities:

{a1, a4, a8, a11}

{a2, a4, a9, a11}

Next steps:

1. We'll first develop a dynamic programming solution. This will be a bit cumbersome (and indeed not efficient), so we'll then revisit and see that we can find a greedy solution.

2. We'll then see that we actually need just one choice – the greedy choice – for an optimal solution.

3. We'll then develop a greedy algorithm.

Dynamic programming:

a. Optimal substructure: We'll divide the full set into two subsets, such that if we know the optimal solutions to the subproblems, then this results in an optimal solution to the full problem.

Original bigger problem: Definitions:

Sij Set of activities that start after ai finishes and finish before aj starts

We'll choose index k and activity ***ak*** in ***Sij***, and divide into two subproblems: ***Sik*** and ***Skj***. (let's say we are already handed the best choice k of dividing the larger problems into subproblems).

Then to find the optimal solution to the larger problem, we need to find the max compatible subset in each of the subproblems ***Sik*** and in ***Sij***, plus include ***ak*** in the solution (since ***ak*** starts after activities in ***Sik*** and finishes before activities in ***Sjk***).

We therefore have: ***Sij = {Sik, ak, Skj}***

If we have an optimal solution to the subproblems, then these are part of the optimal solution to the bigger problem.

We'll define one extra variable, which we will use in the recursion (this will also be the table that we save in the dynamic programming):

C[i,j] = number of activities in the optimal compatible set

Main recursive equation:

C[i,j] = C[i,k] + C[k,j] + 1

The ***+1*** is the extra activity ***ak***.

b. Making a choice k: As usual for dynamic programming, we do not know the optimal k that splits the set of activities. So we try all possible k's. The recursion becomes:

C[i,j] = max ak { C[i,k] + C[k,j] + 1}

That is, we need to find index k and corresponding activity ***ak***, for which the equation (***C[i,j] = C[i,k] + C[k,j] + 1***) is maximal.

Note that unlike the rod cutting problem in which we cut off a piece of the rod and recursed on only one side, here we are reversing on activities to both sides of ***ak*** (both those that ended before ***ak*** started, and those that started after ***ak*** ended).

As an aside, when we access the subproblem solutions ***C[i,k] and C[k,j]*** saved in our table, we'll technically need to make sure they are compatible with ***ak*** (end before activity k starts; and start after activity k finishes). This turns out just an extra line of code in practice to check this condition, when computing for each choice of k.

- We could develop the recursive algorithm and "memoize" smaller subproblem solutions or use a bottom up approach. But you could already see this is getting a bit tedious… In addition the run time is proportional to n cubed (since the table we saved is proportional to n squared, and the number of choices each time is proportional to n).

- We'll instead go on to a greedy approach!

<u>Greedy solution: Next class!</u>