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 cases in which it does not work.

We'll first develop the approach and then 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.

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:

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.

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!

Greedy solution:

Main idea: What is we could just make one choice – a greedy choice – to add to our optimal solution. That is, rather than considering all possible choices, we make a single choice.

Look back at original example and remember that it was pre-sorted by finish time.

Greedy choice: Choose the activity that ends earliest, a1, to give most time for putting in other activities.

Remaining subproblem: If we make a greedy choice, we have one remaining subproblem: find activities that start after a1 finishes (why?)

Formally: Greedy strategy after choosing a1: find best greedy solution in set S1 (set that starts after a1 finishes).

Main algorithm idea: repeatedly choose activity that finishes first, and then keep only set of compatible activities and choose from the remaining set (until set is empty).

Main structure for greedy algorithm:

1. Make a choice (single choice!)

2. Solve remaining subproblem recursively

Caveat: greedy algorithms don't always give optimal solution. It does here (theorem in book). Problem sets: will see examples in which greedy solution is not always optimal.

Pseudo code: We can write out pseudo code as recursive or as bottom up. Here we will show bottom up, which is more intuitive. But you could see recursive in the book.

Main sketch of code for recursive:

Recursive-activity-selector(s,f,k,n)   // for choice k, start times s, finish times f

1. Find first activity in remaining set Sk = {ak+1, ak+2, … an} that starts after ak finishes

2. Return that activity (which we denote am with m>=k+1) and recurse on the remaining activities for this choice m: Recursive-activity-selector(s,f,m,n)

Bottom-up:

```
GREEDY-ACTIVITY-SELECTOR($s, f$)
1  $n = s.length$
2  $A = \{a_1\}$
3  $k = 1$
4  for $m = 2$ to $n$
5      if $s[m] \geq f[k]$
6          $A = A \cup \{a_m\}$
7          $k = m$
8  return $A$
```

Run time: O(n)

[Plus if counting pre-sorting of finish times: O(n log n)]

Main sketch of proof that greedy choice is optimal:

Theorem (simplified from book!): Consider subproblem Sk that has am as the earliest finish time in Sk, and has other activities. Define Ak as the maximum size subset of compatible activities in Sk. Then the claim is that am is included in Ak.

Proof sketch: Let ***aj*** be the activity in ***Ak*** (the optimal subset) with the earliest finish time.

Then ***Ak*** = {***aj***, and other activities that are themselves compatible}

a. If ***aj*** is **equal** to am we are done (am is in the optimal solution of some max size subset of compatible activities)

b. If ***aj*** is **not equal** to am, then we construct a new set ***Ak'***, in which we remove ***aj*** and we add ***am***:

Ak' = Ak – {aj} + {am}

(note: book uses union operation instead of +; here we just use minus and plus for removing and adding an activity from the set)

Then: ***Ak'*** = {***am***, and other activities that are themselves compatible}

Since am has earliest finish time in original set ***Sk***, then ***fm <= fj***

So since ***Ak*** was compatible, our new set ***Ak'*** is also compatible, and includes the same number of activities as in ***Ak***

(so again ***am*** is in the optimal solution of some max size subset of compatible activities)