Greedy algorithms – part 2

Activity selection problem:

Last class we defined the problem and developed a (somewhat tedious) Dynamic Programming approach. Here we will show the greedy algorithm solution.

Greedy solution to activity selection problem:

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] ≥ f[k]
6            A = A ∪ {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)

Greedy algorithms: Two main properties:

1. Greedy choice property: At each decision point, make the choice that is best at the moment. We typically show that if we make a greedy choice, only one property remains (unlike dynamic programming, where we need to solve multiple subproblems to make a choice)

2. Optimal substructure: This was also a hallmark of dynamic programming. In greedy algorithms, we can show that having made the greedy choice, then a combination of the optimal solution to the remaining subproblem and the greedy choice, gives an optimal solution to the original problem. (note: this is assuming that the greedy choice indeed leads to an optimal solution; not every greedy choice does so).

Greedy vs dynamic:

- both dynamic programming and greedy algorithms use optimal substructure

- but when we have a dynamic programming solution to a problem, greedy sometimes does or does not guarantee the optimal solution (when not optimal, can often prove by contradiction; find an example in which the greedy choice does not lead to an optimal solution)

- could be subtle differences between problems that fit into the two approaches

Example: two knapsack problems.

a. 0-1 knapsack problem

- ***n*** items worth ***vi*** dollars each, and weighing ***wi*** pounds each.

- a thief robbing a store (or someone packing for a picnic…) can carry at most ***w*** pounds in the knapsack and wants to take the most valuable items

- It's called 0-1, because each item can either be taken in whole, or not taken at all.

b. Fractional knapsack problem:

- same as above, except that thief (or picnic packer) can now take fractions of items.

Here the fractional knapsack problem (b) has a greedy strategy that is optimal but the 0-1 problem (a) does not!

We show the figure in the book, and then give a brief explanation.

Figure:

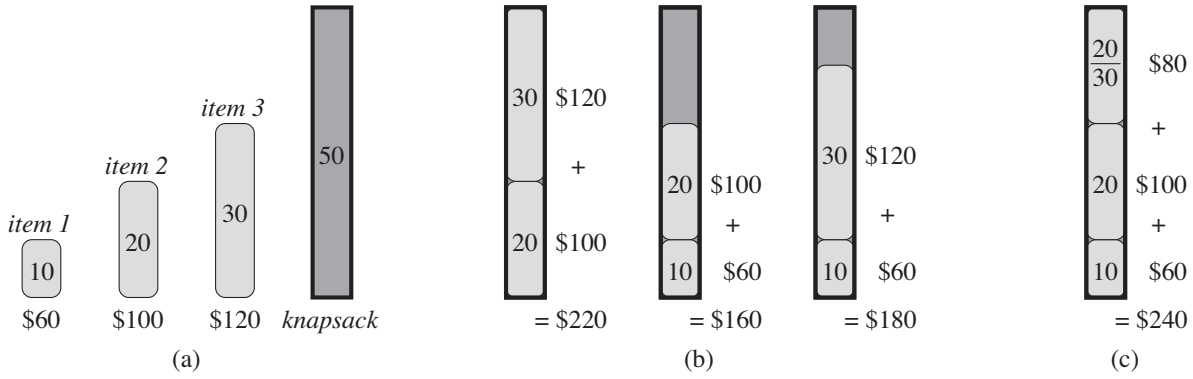

Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Explanation:

a. 0-1: Consider taking items in a greedy manner based on the highest value per pound. In the 0-1 knapsack problem, this would lead to a contradiction, since then would take 10 the pound item first with value per pound equal to 6 (non optimal).

To get the optimal solution for the 0-1 problem, we must compare the solution or subproblem that includes the 10 pound item, with the solution or subproblem that excludes it. There are many overlapping subproblems we must compare.

b. Fractional: The fractional does have an optimal greedy solution of filling the highest value per pound first until the knapsack is full. This works indeed because we can fill fractions of items, and so proceed until knapsack is entirely full. See panel (c) in the figure.

Note that: both problems have optimal substructure:

a. 0-1: Consider the most valuable load that weighs at most w pounds. If we remove item ***j***, the remaining load must be the most valuable load weighing at most ***W-wj*** pounds, that the thief can take from the n-1 original items (excluding ***j***).

b. Fractional: If we remove weight ***w*** from item ***j*** (a fraction of the weight of ***j***), then the remaining load must be the most valuable weighing at most ***W – w*** that the thief can take from the ***n-1*** original items, plus ***wj – w*** pounds from item ***j***

Also note: These are well known problems you still hear about in conferences today; could be for many optimization problems…