Lecture 20: Introduction to dynamic programming (15.1, 15.3)

CS303: Algorithms

Last update: February 26, 2014

1 Where we are

1. theory and divide and conquer
2. sorting
3. data structure (hashing, BST, RB tree)
4. design techniques (DP, greedy)

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping sub-problems. It was invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS.

The programming in the definition really means planning or constructing.

Note: DP has nothing to do with general programming language (programming here means table)

2 Idea of dynamic programming – big idea, hard, yet simple

- There is a pretty large class of seemingly exponential problems have a polynomial time solution only if you use DP.
- It is particularly true for optimization problems (min/max)

There are some problems that local optimum and global optimum are related. Thus dynamic programming tries to build up global optimum from local optimum.

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

Q:/ What kind of problem can be solved with dynamic programming?

A:/ If it satisfies the following tow two attributes

- Optimal substructure (i.e. global optimum is the generalized local optimum)
- Overlapping subproblems (i.e. we can build from local optimum and get global optimum)

Usually dynamic program uses memoization (i.e. table) in its process to get global optimum by reusing the previous results.

3 Example: rod cutting

A steel company buys long steel rods and cuts them into shorter rods, which it sells. Cut is free since they have the machine. The question is what is the best way to cut since the price of rods isn't always linear with respect to their lengths.

e.g.

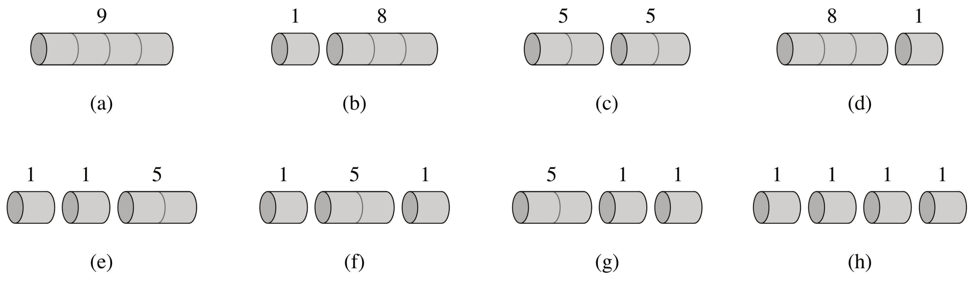
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

if length n is 4, then the best way to sell is cut the rod into 2 pieces of 2 inch each

3.1 Analysis of this problem

Input: a length n and table of prices $p[1, 2, ..., n]$
Output: the max revenue obtainable from rods whose lengths sum to n, computed as the sum of prices for the individual rods.

Q:/ How many different ways can we cut up a rod of size n?
A:/ there are 2^{n-1} ways since at each inch point, we have the choice of cut or not cut.
e.g. There are 8 ways if n is 4



Can determine optimal revenue r_n for a rod of length n by taking the maximum of

- p_n : the price we get by not making a cut,

- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches
- ...
- $r_{n-1} + r_1$

That is, $r_n = max\{p_n; r_1 + r_{n-1}; r_2 + r_{n-2}; ...; r_{n-1} + r_1\}$

3.2 Optimal substructure

To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, we have two subproblems. The optimal solution to the original problem incorporates optimal solutions to the subproblems. We may solve the subproblems independently.

e.g.: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. We need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

This problem can be simplified because every optimal solution has a leftmost cut. In other words, we don't have to worry about the first cut (i.e. the first cut no longer needs to be r, it can be p). This will leave only one sub-problem to be solved.
Assuming that the solution with no cuts has the first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$

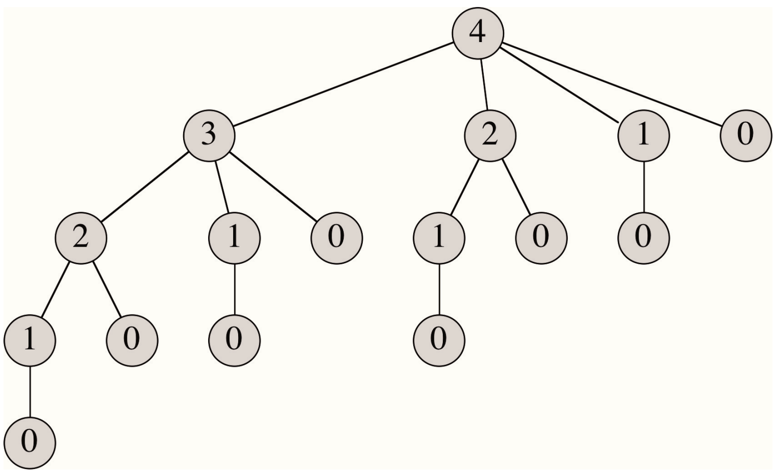
$r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})$.

Solution 1: top down calculation

```
Cut-rod(p,n)
if(n==0)
   return 0
q=$-\infty$
for i=1 to n
    $q = max(q, p[i]+$Cut-rod$(p, n-i))$
return q
```

Q:/ Is it good?
A:/ In fact it is very bad because we are doing the same thing repeated – similar to Fibonacci sequence problem.

e.g. n=4



Q:/ What's the efficiency?
A:/ exponential. (proof in homework. Hint: use substitution method)

Q:/ How to avoid repeated caclulation?
A:/ Memoization! Remember what we did.

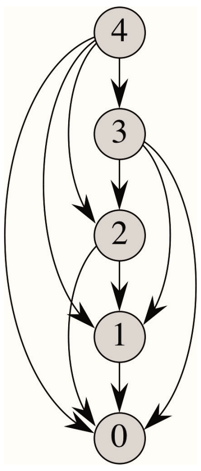
This will save lots of time → it will be $O(n^2)$ if we remember.

3.3 Subproblem graphs

How to understand the subproblems involved and how they depend on each other.

- Directed graph:
- One vertex for each distinct subproblem.
- Has a directed edge (x, y) if computing an optimal solution to subproblem x directly requires knowing an optimal solution to subproblem y.

Example: For rod-cutting problem with n =4:



Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child. Subproblem graph can help determine running time. Because we solve each sub- problem just once, running time is sum of times needed to solve each subproblem.

Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.

Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.