WIA2005: Algorithm Design and Analysis
Semester 2, Session 2016/17

Lecture 10: Dynamic Programming

Learning objectives

- Know what is dynamic programming
  - memoization
  - bottom up

Introduction

- Dynamic programming(DP), like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Steps in DP

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

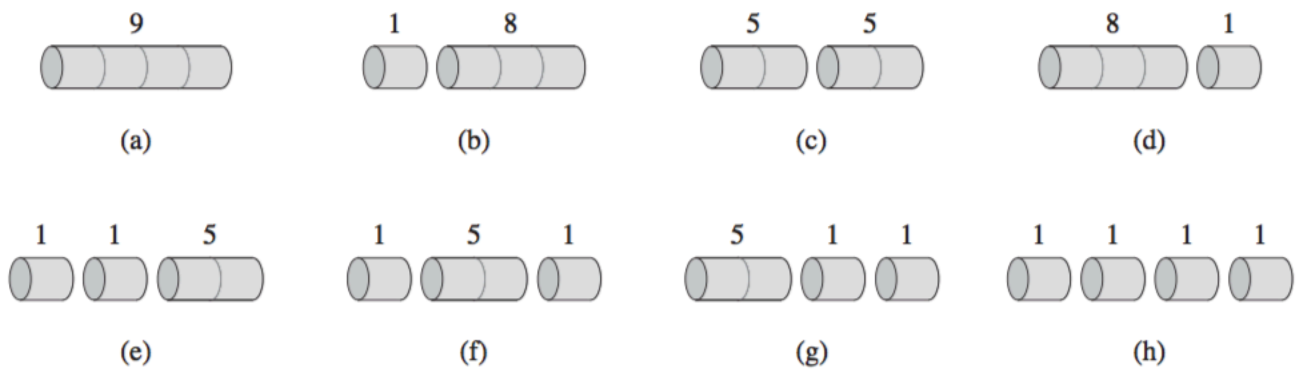
Rod Cutting Problem

- Examines the problem of cutting a rod into rods of smaller length in way that maximizes their total value.
- Given a rod of length n inches and a table of prices p_i for i = 1, 2, 3, …, n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
- Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

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

Example

- Consider the case when n = 4.
- We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.



How to determine best cut

- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for i =1, 2, 3, …, n-1.
- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

- of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Optimal revenue figures

- For i = 1, 2, …, 10, by inspection, with the corresponding optimal decompositions

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

- More generally, we can frame the values r_n for n≤1 in terms of optimal revenues from shorter rods:

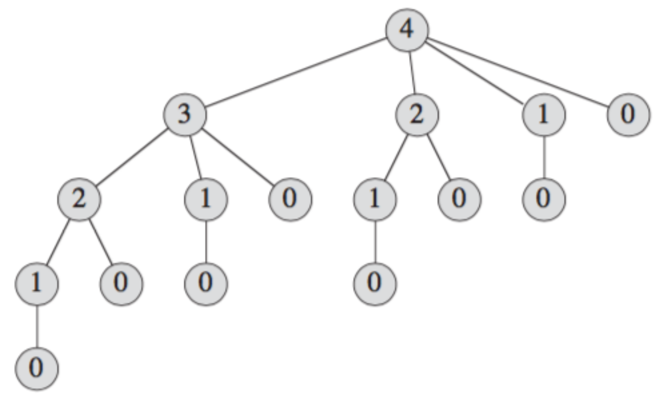
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

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

Recursive top-down implementation

- The following procedure implements the computation implicit in equation in a straightforward, top-down, recursive manner.

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```



Question

- Why is CUT-ROD so inefficient?
- The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly.

Using dynamic programming for optimal rod cutting

- Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution.
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it.
- Dynamic programming thus uses additional memory to save computation time; it serves an example of a ***time-memory trade-off***.
- The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.
- A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

Ways to implement DP

- *Top-down (memoization*)
- *Bottom-up (tabular)*

Memoization

- In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- The procedure now first checks to see whether it has previously solved this subproblem.
- If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.
- We say that the recursive procedure has been ***memoized***; it "remembers" what results it has computed previously.

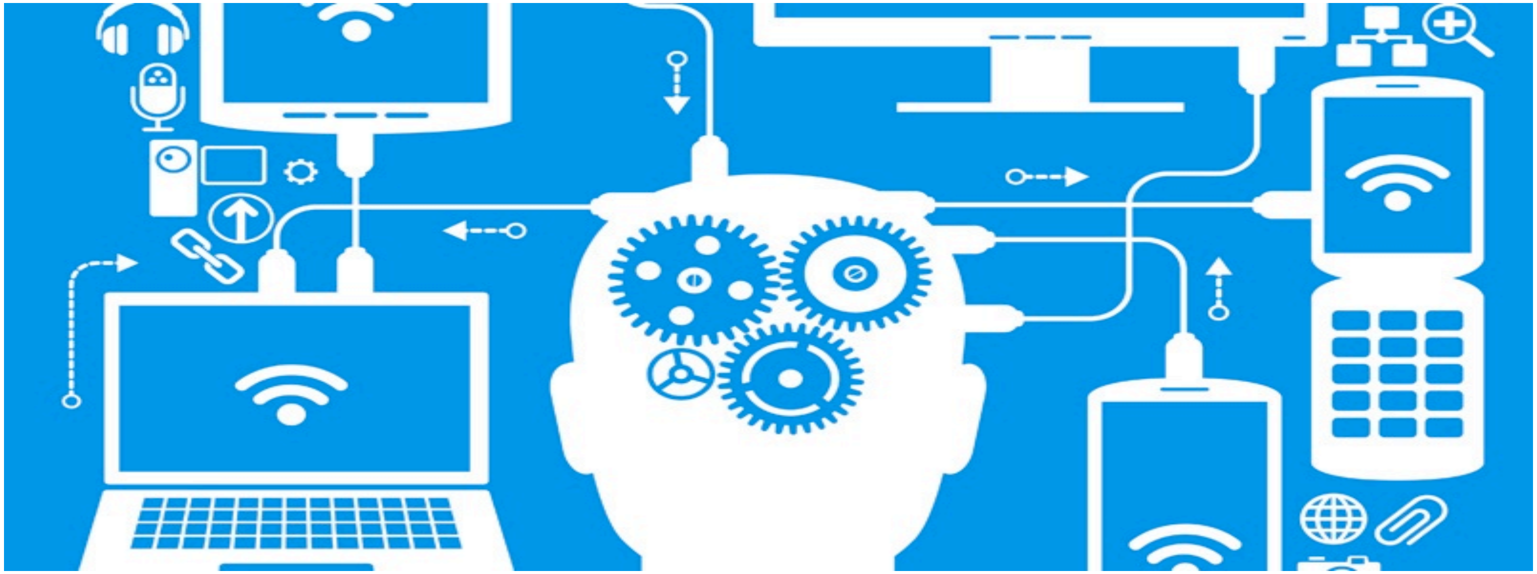
Bottom up (Tabular)

- This approach typically depends on some natural notion of the “size” of a subproblem, such that solving any particular subproblem depends only on solving “smaller” subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.
- We solve each sub- problem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Comparing memoization and bottom up

- These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems.
- The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

In the next lecture..



Lecture 11: Graph Algorithm