EEECE7205: Fundamentals of Computer Engineering

Dynamic Programming

Introduction

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub-problems.
- "Programming" in this context refers to a tabular method, not to writing computer code.
- Dynamic programming applies when the sub-problems overlap—that is, when sub-problems share sub-sub-problems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-sub-problems.
- A dynamic-programming algorithm solves each sub-sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem.

Optimization Problems

- We typically apply dynamic programming to ***optimization problems***.

- Such problems can have many possible solutions. Each solution has a **value**, and we wish to find a solution with the optimal (minimum or maximum) value.

- We call such a solution ***an* optimal** solution to the problem, as opposed to ***the* optimal** solution, since there may be several solutions that achieve the optimal value.

"*n* choose *k*" Example (1 of 3)

- **Problem**: Write an algorithm to calculate the number of combinations "*n* choose *k*", `C(n, k)`
- *<u>Solution 1</u>* - using factorials:

$$C(n, k) = \frac{n!}{k!\,(n-k)!}$$

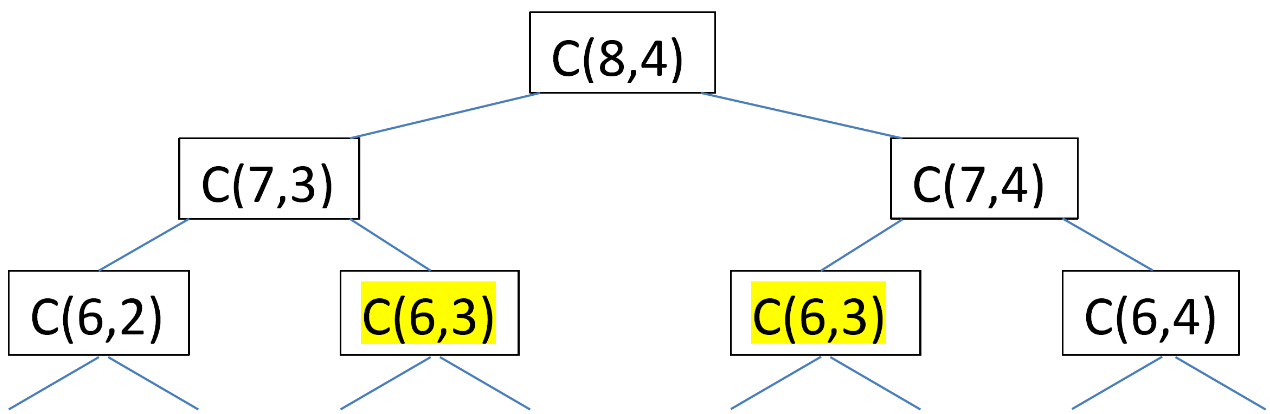
- The problem with this approach is that the factorial of a number grows very rapidly and will exceed the range of even long integer variables.

"*n* choose *k*" Example (2 of 3)

- *Solution 2* - using Pascal's recursive formula for combinations:

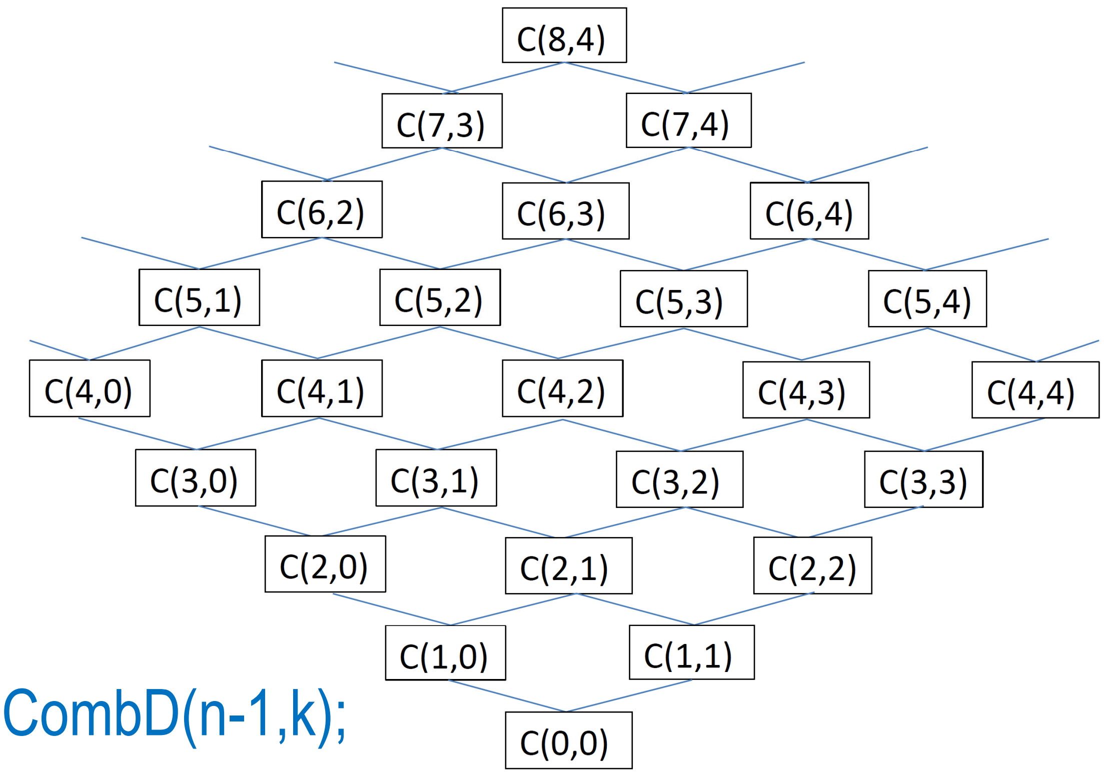
```
C(n, 0) = 1
C(n, n) = 1
C(n, k) = C(n–1, k–1) + C(n–1, k), when 0 < k < n
```

- The problem with this approach is repeating the calculations of the same sub-problems resulting in an inefficient *exponential* running time as shown:



"*n* choose *k*" Example (3 of 3)

- *Solution 3* – using dynamic programming implementation of Pascal's triangle for combinations. This approach is called Top-Down with Memoization.




```
allocate array C[0…n][0…k]
and initialize its contents to -1

int CombD(int n, int k) {
   if (C[n][k] != -1) return C[n][k];
   if (k == 0 || k == n) C[n][k] = 1;
      else C[n][k] = CombD(n-1,k-1) + CombD(n-1,k);
  return C[n][k];
}
```

- Pros: Running time: $\theta(nk)$ and $O(n^2)$ as $k \leq n$
- Cons: Extra space needed also of $\theta(nk)$

The Rod Cutting Problem

- Using dynamic programming to solve a simple problem in deciding where to cut steel rods.

- The Problem:

  Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots , n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

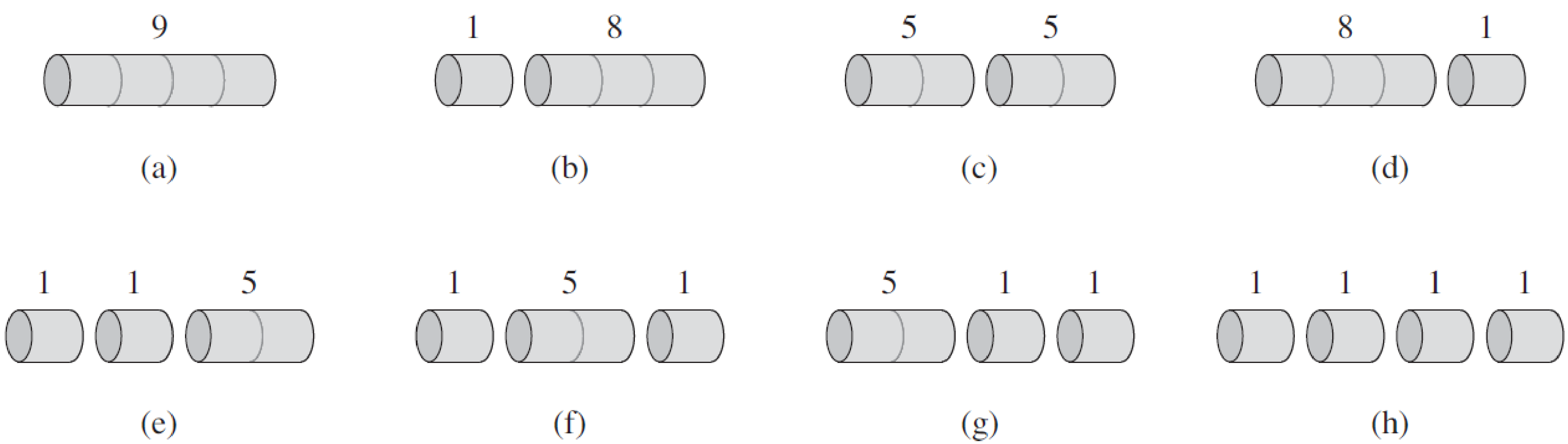
- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end, for $i = 1, 2, \ldots , n - 1$.

Rod Cutting Example

- Assume a rod of size 4 inches. The following is the given table of prices p_i

length i	1	2	3	4
price p_i	1	5	8	9

- The following figure shows all the ways to cut up the rod and the revenue of each cut.
- We see that cutting the rod into two 2-inch pieces produces the optimal revenue of 10.



Rod Cutting Solution

- We can frame the values of maximum revenue r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, p_{n-1} + r_1, p_{n-2} + r_2, \ldots, p_1 + r_{n-1})$$

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

- p_n corresponds to making no cuts at all and selling the rod of length n as is. The other n - 1 arguments correspond to the maximum revenue obtained by adding the price of a cut of size n-i to the maximum revenue of a rode of size i. for each i = 1, 2, … , n – 1.
- In this formulation, an optimal solution embodies the solution to *one* related sub-problem—the remainder — instead of comparing 2^{n-1} solutions.

Recursive Implementation

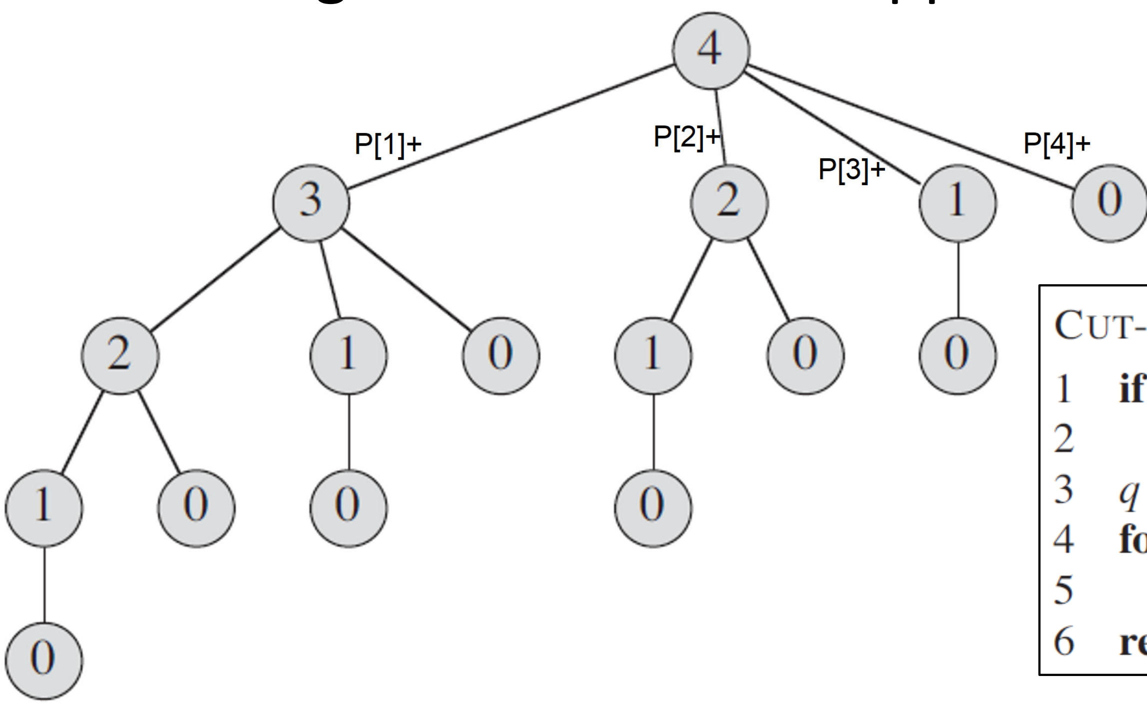
- Procedure CUT-ROD takes as input an array *p*[1 .. *n*] of prices and a rod of length *n* (an integer).

```
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
```

- If you were to code up CUT-ROD in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large (more than 40), your program would take a long time to run.

Recursive Implementation (Cont'd)

- The problem is that the presented CUT-ROD algorithm calls itself recursively repeatedly with the same parameter values; it solves the same sub-problems repeatedly.
- When this process unfolds recursively, the amount of work done, as a function of *n*, grows exponentially.
- The figure shows what happens for *n* = 4:




```
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
```

Dynamic Programming for Rod Cutting

- The dynamic-programming method arranges for each sub-problem to be solved only *once.*
- If we need to refer to this sub-problem's solution again later, we can just look it up, rather than re-computing it.
- Dynamic programming thus uses additional memory to save computation time.
- In this approach the recursive algorithm is modified to save the result of each sub-problem (usually in an array).
- The procedure first checks to see whether it has previously solved this sub-problem. If so, it returns the saved value; if not, the procedure computes the value in the usual manner.

Rod Cutting Algorithm using Dynamic Programming

MEMOIZED-CUT-ROD(p, n)

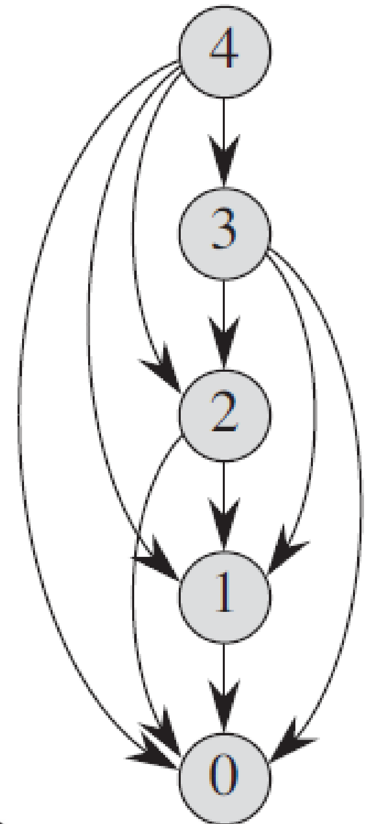
let $r[0\,..\,n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

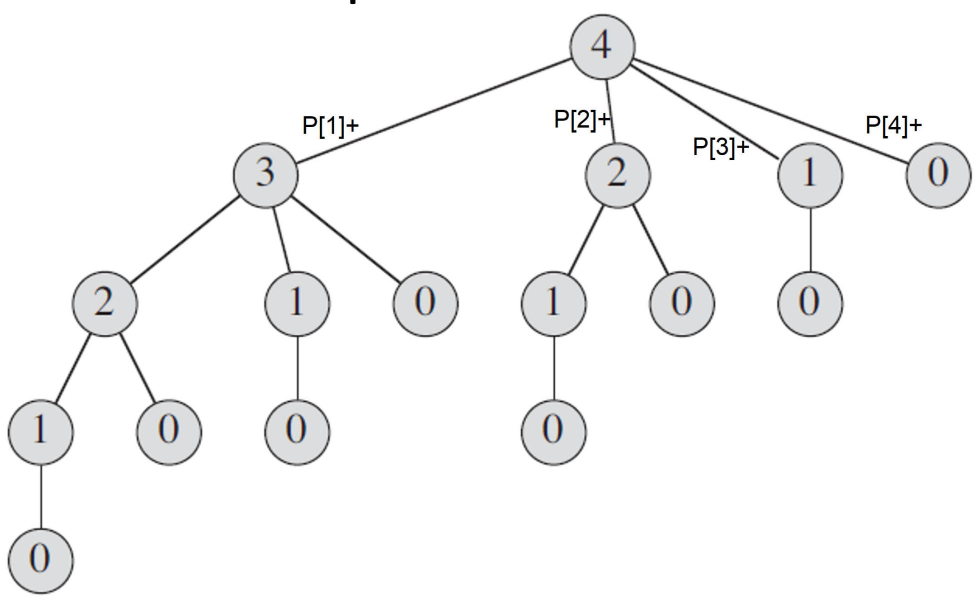
MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
  for $i = 1$ **to** n
    $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return q

Sub-problem Graphs

- The figure shows the sub-problem graph for the rod-cutting problem with *n* = 4.
- It is a directed graph, containing one vertex for each distinct sub-problem.
- A directed edge (*x*, *y*) indicates that we need a solution to sub-problem *y* when solving sub-problem *x*.
- This graph is a reduced version of the following tree that represents the recursive implementation.





Reconstructing the Solution (1 of 2)

- The dynamic-programming solution to the rod-cutting problem return the value of an optimal solution, but it does not return an actual solution (i.e., a list of piece sizes).
- Here is an extended version of the algorithm to return not only the optimal value, *val*, but the actual cut solution, *s*, too.

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  for i = 0 to n
      r[i] = -∞
  (val, s) = MEMOIZED-CUT-ROD-AUX(p, n, r, s)
  print "The optimal value is " val " and the cuts are at "
  j = n
  while j > 0
      print s[j]
      j = j - s[j]
```

Reconstructing the Solution (2 of 2)

```
MEMOIZED-CUT-ROD-AUX(p, n, r, s)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          (val, s) = MEMOIZED-CUT-ROD-AUX(p, n − i, r, s)
          if q < p[i] + val
              q = p[i] + val
              s[n] = i
  r[n] = q
  return (q, s)
```

- Array entry *s*[*n*] contains the value *i*, which is an optimal cut for a rod of length *n*. The next cut is given by *s*[*n*-*i*], and so on.

Elements of Dynamic Programming

- When should we look for a dynamic-programming solution to an optimization problem?
- Two key ingredients that an optimization problem must have for dynamic programming to apply:
  1. Optimal sub-structure
  2. Overlapping sub-problems.

Optimal Substructure

- A problem exhibits ***optimal substructure*** where we build an optimal solution to the problem from optimal solutions to sub-problems.

- We observed that the optimal way of cutting up a rod of length *n* (if we make any cuts at all) involves optimally cutting up the two pieces resulting from the first cut.

Overlapping Sub-problems

- For dynamic programming to apply, the space of sub-problems must be "<u>small</u>" to be able to store their solutions in a table.
- When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping sub-problems***.
- Dynamic-programming algorithms typically take advantage of overlapping sub-problems by solving each sub-problem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.