CSCI203

Algorithms and Data Structures

Dynamic Programming (I)

Lecturer: Dr. Xueqiao Liu
Room 3.117
Email: xueqiao@uow.edu.au

Dynamic Programming (DP)

- Typically applied to optimization problems

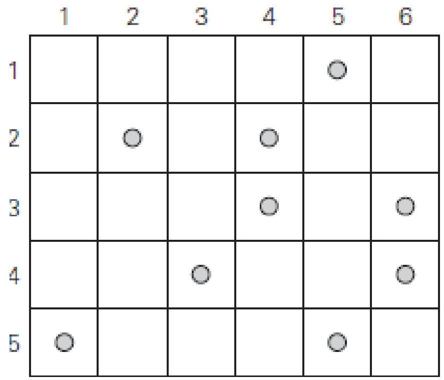
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Coin-Collection Problem

- Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell.

- On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin.

- Find the maximum number of coins the robot can collect and a path it needs to follow to do this.

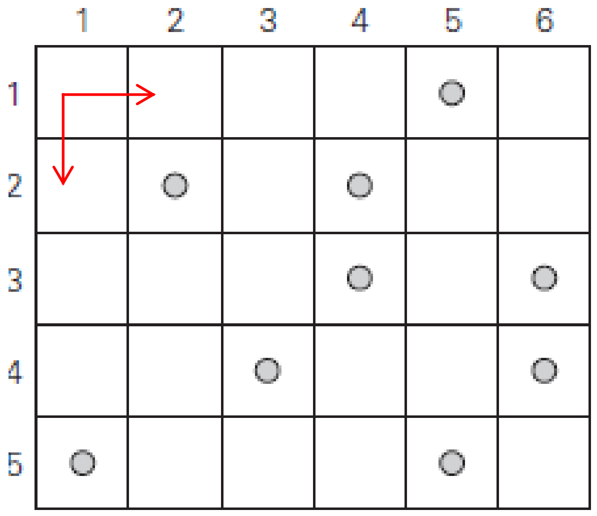


Coin-Collection Problem

- $F(i, j)$ - the largest number of coins the robot can collect and bring to the cell (i, j).

- It can reach this cell either from the adjacent cell $(i-1, j)$ above it or from the adjacent cell $(i, j-1)$ to the left of it. The largest numbers of coins that can be brought to these cells are $F(i-1, j)$ and $F(i, j-1)$, respectively.

- $F(i-1, j)$ and $F(i, j-1)$ are equal to 0 for their nonexistent neighbors



Coin-Collection Problem

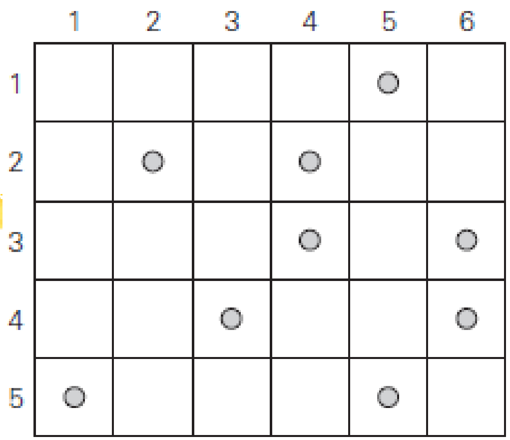
$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \text{ for } 1 \le i \le n, 1 \le j \le m$$

$F(0, j) = 0 \text{ for } 1 \le j \le m$ and $F(i, 0) = 0 \; for \; 1 \le i \le n$

- $c_{ij} = 1$ if there is a coin in cell (i, j) and $c_{ij} = 0$ otherwise

Algorithm

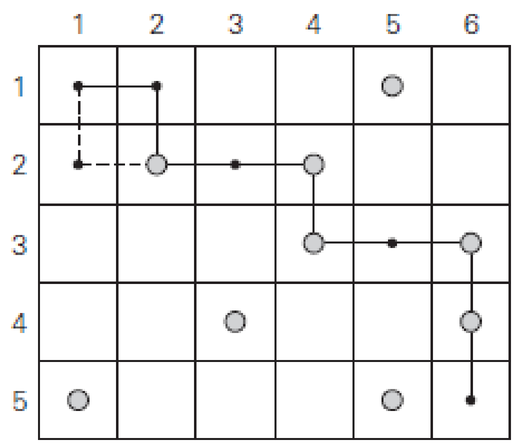
```
RobotCoinCollection(C[1..n, 1..m])
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an n × m board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix C[1..n, 1..m] whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)

F[1, 1]←C[1, 1];
for j ←2 to m do
    F[1, j]←F[1, j − 1]+ C[1, j]
for i ←2 to n do
    F[i, 1]←F[i − 1, 1]+ C[i, 1]
    for j ←2 to m do
        F[i, j ]←max(F [i − 1, j], F[i, j − 1]) + C[i, j ]
return F[n, m]
```



(a)

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

(b)



(c)

- (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible

Shortest Paths

- Let us apply the insights we have gained on dynamic programming to find
  - Single source, single destination shortest path.
- We will proceed as follows:
  - Create a top down, recursive, naïve algorithm;
  - Memorize it;

Step 1: the Naïve, Recursive Algorithm

- In deriving the naïve algorithm we need to introduce another key component of dynamic programming…
  - …guessing!
- Don't know the answer?
  - Guess!
- Don't just try any guess…
  - …try them all!
- So, DP = recursion + memoization + guessing.
- The best guess is the answer we are looking for

Some Notation for Shortest Paths

- Remember from last week:
  - Given a graph, $G = (V, E, W)$, find the shortest path from a starting vertex, $s \in V$, to all other vertices, $v \in V$;
  - $w(u, v)$ is the weight of the edge (u, v);
  - $D(s, v)$ is the length of the shortest path between s and v.
- If some vertex, u, is on the shortest path from s to v then:
  - D(s, v) = D(s, u) + D(u, v).
- Specifically, if vertex u immediately precedes vertex v in the shortest path from s to v, then:
  - $D(s, v) = D(s, u) + w(u, v)$.
- Our problem is that we don't know which vertex, u, to try…
  - …so we guess—try them all and pick the best.

The Naïve Algorithm

```
Procedure short(V{}: vertex, E{}: edge, W(): weight,
                s: vertex, v: vertex)
      if v==s then
            d=0
      else
            d = ∞
            for each u where (u,v) ∈ E
                  d = min(d, short(V, E, W, s, u) + w(u,v))
            rof
      fi
      return d
End procedure short
```

- This is a really bad algorithm:
  - We compute the shortest path between s and every other vertex repeatedly.
- It is really easy to improve, however;
- Memoize the computation.

Step 2: The Memoized Algorithm

```
D: dictionary {}
Procedure shortDP(V{}: vertex, E{}: edge, W(): weight, s: vertex,
        v: vertex)
        if v==s then
                d=0
        else
                d = ∞
                for each u where (u,v) ∈ E
                  if (u in D) then
                        d=min(d, D[u]+w(u,v))
                  else
                        d = min(d, shortDP(V, E, W), s, u) + w(u,v))
                  fi
                rof
        fi
        D[v]=d
        return d
End procedure shortDP
```

Analysis

- This algorithm only works for acyclic group
  - It takes infinite time if G has one or more cycles.
- If g is acyclic the algorithm is $O(|V| + |E|)$

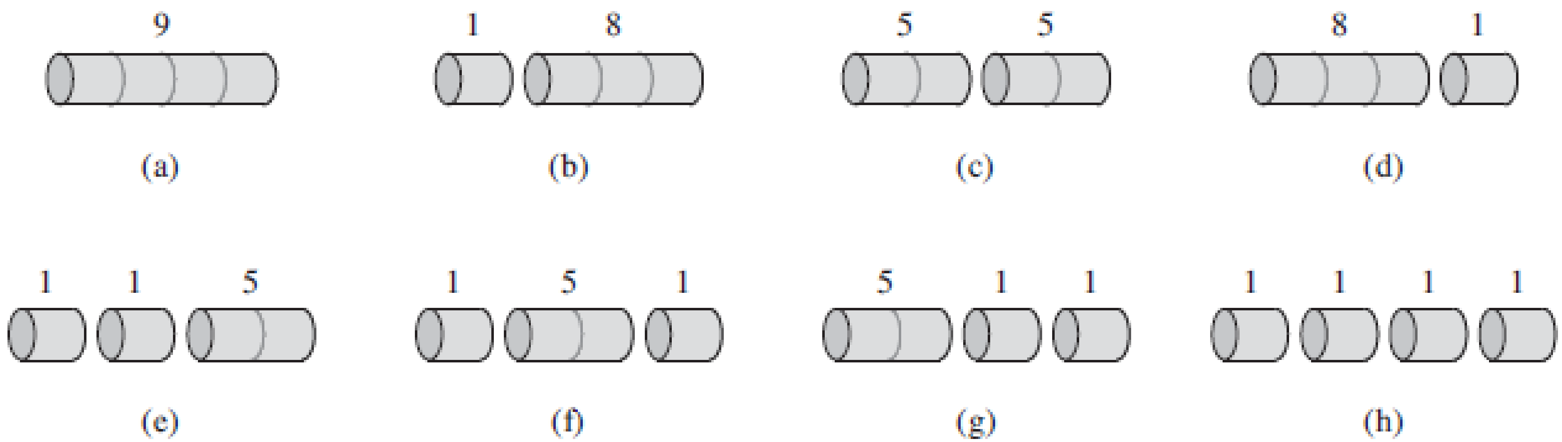
Rod-Cutting Problem

- Given a rod of length n inches and a table of prices p_i for $i = 1,2,\cdots,n$, 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.
- We will proceed as follows:
  - Create a top down, recursive algorithm;
  - Memorize it;
  - Reconstruct it as a bottom up algorithm.
- This is a useful general approach to algorithm design in dynamic programming.

A rod of length n has 2^{n-1} cutting options

- Consider a case $n = 4$

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



Optimal revenue $p_2 + p_2 = 10$

- A rod of length n in 2^{n-1} different ways
- For $i = 1,2,\cdots,n-1$, we denote a decomposition using ordinary additive notation
  - $7 = 2 + 2 + 3$ means a rod of 7 length is cut into three pieces, two of length 2 and one of length 3

- If an optimal solution cuts the rod into k pieces, $1 \leq k \leq n$, of length $i_1, i_2, \cdots, i_k$

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

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

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

- For the sample, we can determine the optimal revenue figures r_i for $i = 1,2,\cdots,10$ by inspection

$r_1 = 1$ from solution $1 = 1$ (no cuts);

$r_2 = 5$ from solution $2 = 2$ (no cuts);

$r_3 = 8$ from solution $3 = 3$ (no cuts);

$r_4 = 10$ from solution $4 = 2 + 2$;

$r_5 = 13$ from solution $5 = 2 + 3$;

$r_6 = 17$ from solution $6 = 6$ (no cuts) ;

$r_7 = 18$ from solution $7 = 1 + 6$ or $7 = 2 + 2 + 3$;

$r_8 = 22$ from solution $8 = 2 + 6$;

$r_9 = 25$ from solution $9 = 3 + 6$;

$r_{10} = 10$ from solution $10 = 10$ (no cuts) :

- Generally, r_n for $n \geq 1$ in terms of optimal revenues from shorter rods
  - $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$
  - p_n corresponds to no cut at all
  - $r_i + r_{n-i}$ corresponds to an initial cut into two pieces of size i and $n - i$ for each $i = 1,2,\cdots, n-1$, then optimally cut those pieces further to obtain revenues r_i and r_{n-i}

- We don't know which i optimizes revenue, so we have to consider all possible i and pick the one that optimizes the revenue
- We also have the option of picking no i , i.e. selling the rod without cut

- Assumption to make the problem simpler
  - Piece of length i is left-hand end
  - Piece of length $n-i$ is the right hand remainder
  - Only the remainder may be further divided
- We view the cut of a length n rod as follows
  - A first piece followed by some decomposition of the remainder
  - i length of cut, p_i revenue of the i length of cut
  - r_{n-i} remainder length
  - $i = n$ corresponds to no cut, revenue p_n, remainder has size 0, $r_0 = 0$

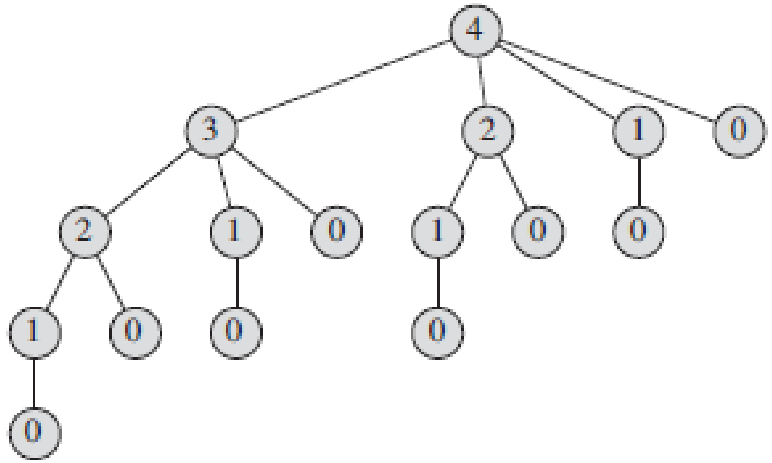
$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Recursive Top-down Implementation

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

Analysis

- An example with $n = 4$, recursive call tree



- In general, this recursion tree has 2^n nodes and 2^{n-1}leaves
- CUT-Rod explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n.

DP for Optimal Rod-Cutting

- Top-down with memorization

```
MEMOIZED-CUT-Rod(p,n)
r[0…n]: an array
for i = 0 to n
  r[i] = −∞
return MEMOIZED-CUT-ROD-AUX (p,n,r)


MEMOIZED-CUT-ROD-AUX(p,n,r)
  if r[n]>0 then
    return r[n]
  if n == 0 then
    q=0
  else
    q=−∞
    for i = 1 to n
       q = max (q,; p[i]+ MEMOIZED-CUT-ROD-AUX(p,n-i,r)
  r[n]= q
return q
```

DP for Optimal Rod-Cutting...

- Bottom-up approach

```
BOTTOM-UP-CUT-ROD (p,n)
r[0…n]: a new array
r[0]=0
for j = 1 to n
    q = −∞
    for i= 1 to j
        q = max(q, p[i]+r[j-i])
    r[j] = q;
return r[n]
```

Analysis

- Complexity
  - Top-down: $\Theta(n^2)$
  - Bottom-up: $\Theta(n^2)$

- Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution
  - a list of piece sizes.

Reconstruction a solution

```
EXTENDED-BOTTOM-UP-CUT-ROD (p,n)
  r[0…n] and s[0…n]: arrays
  r[0]=0
  for j = 1 to n
    q = −∞
    for i = 1 to j
      if q < p[i]+r[j-i] then
        q = p[i]+r[j-i]
        s[j]=i
    r[j]=q
  return r and s
```

Reconstruction a solution…

```
PRINT-CUT-ROD-SOLUTION(p,n)
(r,s)=EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
while n > 0
   print s[n]
   n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- $n = 10$, no cut
- $n = 7$, cuts 1,6

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 8.1
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 15.1