ch15: Dynamic Programming

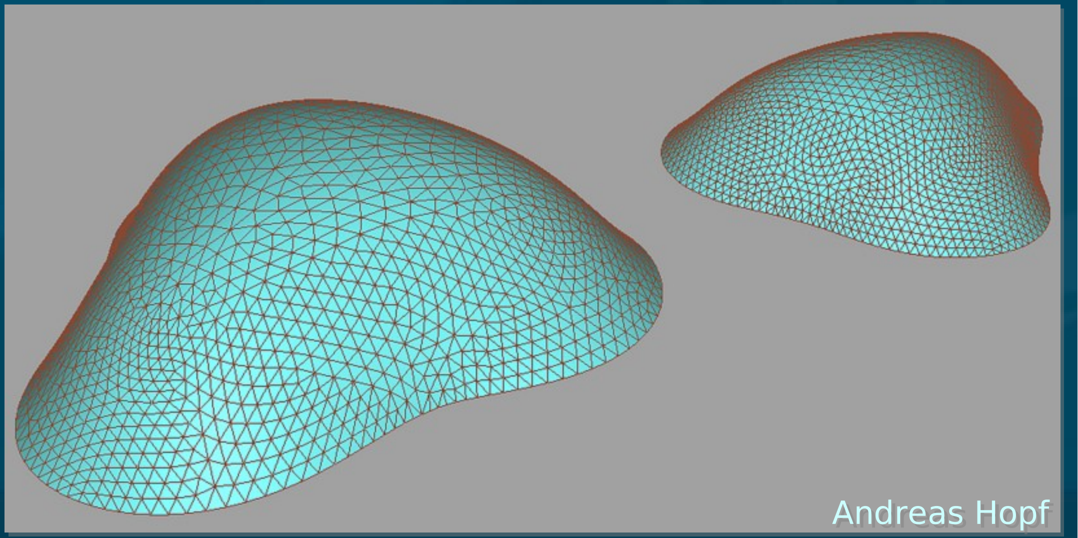
6 Nov 2012
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Dynamic programming for optimisation
  - Optimal substructure
    - Naive top-down
    - Top-down with memoisation
    - Bottom-up
- Examples:
  - Rod-cutting problem
  - Fibonacci
  - Matrix-chain multiplication
  - Shortest unweighted path
  - Optimal binary search trees

Optimisation

- A large class of real-world problems consist of:
  - Find the maximum (or minimum) value of some goal/cost function, over some search space
- Search space may be discrete or continuous, low-dimensioned or very high (10^6 or more) dim
- Goal function may be analytic or some black-box
  - May or may not have accessible derivatives
- Exhaustive search is usually way too slow



Dynamic programming

- "Programming" as in tables, e.g., linear prog.
- Divide-and-conquer approach, but
  - Store and re-use solutions to sub-problems
- 3 implementation schemes:
  - Recursive top-down (inefficient)
  - Top-down with memoisation (save sub-results)
  - Bottom-up (solve smaller sub-problems first)
- Efficiency depends on:
  - Optimal substructure
  - Overlapping subproblems

Outline for today

- Dynamic programming for optimisation
  - Optimal substructure
    - Naive top-down
    - Top-down with memoisation
    - Bottom-up
- Examples:
  - Rod-cutting problem
  - Fibonacci
  - Matrix-chain multiplication
  - Shortest unweighted path
  - Optimal binary search trees

Rod-cutting problem

- Steel rods of length i can be sold for \$$p_i$ each
- How to cut a single rod of length n into pieces so as to maximise revenue?
  - Assume cuts are free
- e.g., price table p=[1, 5, 8, 9]. Rod length n=4
  - Exhaustive search:
    \$9, \$8+1, \$1+8, \$5+5,
    \$5+1+1, \$1+5+1, \$1+1+5, \$1+1+1+1
    - Optimal: 2 pieces of length 2 ⇒
    - CutRod(p, 4) = r_4 = \$5+5
- Can we solve by reusing results of sub-problems?

Optimal substructure

- Optimise one cut at a time, left to right
- Cut into two pieces, assume first piece won't be cut again; recurse on second piece:
  - CutRod(p, n) = $\max_{1 \le i \le n}$(p[i] + CutRod(p, n-i))
- Re-uses smaller subproblems many times
  - CutRod() with small n is called many times
- Optimal substructure means:
  - Task can be split into subproblems which can be solved independently
  - The same subproblems show up in multiple branches of recursion tree (overlapping work)

(1) Recursive top-down

- Naive implementation of the recurrence above:
  - def CutRod(p, n):
    - if (n<1): return 0
    - q = -infinity
    - for i = 1 .. n:
      - q = max(q, p[i] + CutRod(p, n-i))
    - return q
- Each iteration of loop makes recursive call
- Complexity? Recursion tree?
  - $T(n) = 2^n$ (Exercise 15.1-1)
  - Increasing input by 1 ⇒ double the run time!
- Why so bad? e.g., CutRod(2) is run many times

(2) Top-down with memoisation

- Memoisation: cache previously-computed results
  - cache = array[0..n] of -infinity
  - cache[0] = 0
  - def CutRod(p, n):
    - if cache[n] ≠ -infinity:
      - return cache[n]
    - for i in 1 .. n:
      - cache[n] = max(cache[n], p[i] + CutRod(p, n-i))
    - return cache[n]
- CutRod(n) is computed only once for each n
  - CutRod(n) takes $\Theta(n)$ to compute if not cached
  - ⇒ Complexity is $\Sigma_i \Theta(i) = \Theta(n^2)$

(3) Bottom-up

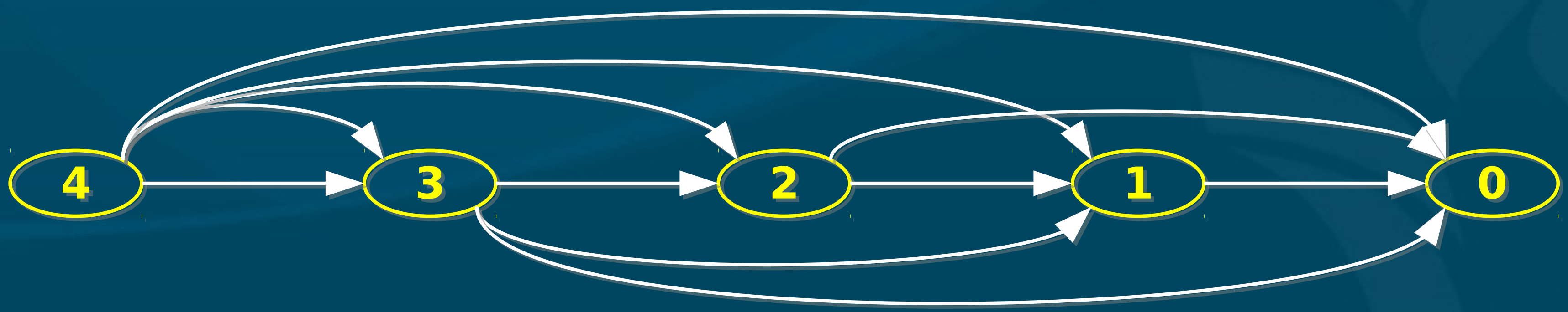
- Start from smaller subproblems, caching as we go

```
def CutRod(p, n):
    cache = array[0..n] of -infinity
    cache[0] = 0
    for j = 1 .. n:
        for i = 1 .. j:
            cache[j] = max(cache[ j ], p[ i ] + cache[ j – i ]))
    return cache[n]
```

- Non-recursive! (function calls are expensive)
- Doubly-nested for loop calculates each CutRod(j)
- Cache stores results of subproblems, which each are re-used many times
- Complexity: $\Sigma_j\ \Theta(j) = \Theta(n^2)$

Subproblem graph

- Nodes are subproblems (e.g., CutRod(n))
- Arrows indicate which other smaller subproblems are needed to compute each node
  - Like recursion tree, but collapsing same nodes
- Bottom-up: order nodes so that all dependencies are precomputed before we reach a node
- Top-down: depth-first search down to leaves
- Complexity often Θ(#nodes + #arrows)



Outline for today

- Dynamic programming for optimisation
  - Optimal substructure
    - Naive top-down
    - Top-down with memoisation
    - Bottom-up
- Examples:
  - Rod-cutting problem
  - Fibonacci
  - Matrix-chain multiplication
  - Shortest unweighted path
  - Optimal binary search trees

Fibonacci sequence

- Recall: $F_n = F_{n-1} + F_{n-2}$
  - $F_0 = F_1 = 1$
- Closed form: $\Theta(1)$

```
def fib(n):
    return round( pow( phi, n ) )
```

- Top-down w/memo: $\Theta(n)$

```
c = array[0..n] of -1
c[0] = c[1] = 1
def fib(n):
    if (c[n]>0): return c[n]
    c[n] = fib(n-1) + fib(n-2)
    return c[n]
```

- Naive top-down: $\Theta(2^n)$

```
def fib(n):
    if (n<2): return 1
    return fib(n-1) + fib(n-2)
```

- Bottom-up: $\Theta(n)$

```
def fib(n):
    c = array[0..n] of -1
    c[0] = c[1] = 1
    for j = 2 .. n:
        c[j] = c[j-1] + c[j-2]
    return c[n]
```

- Subproblem graph?

Matrix-chain multiplication

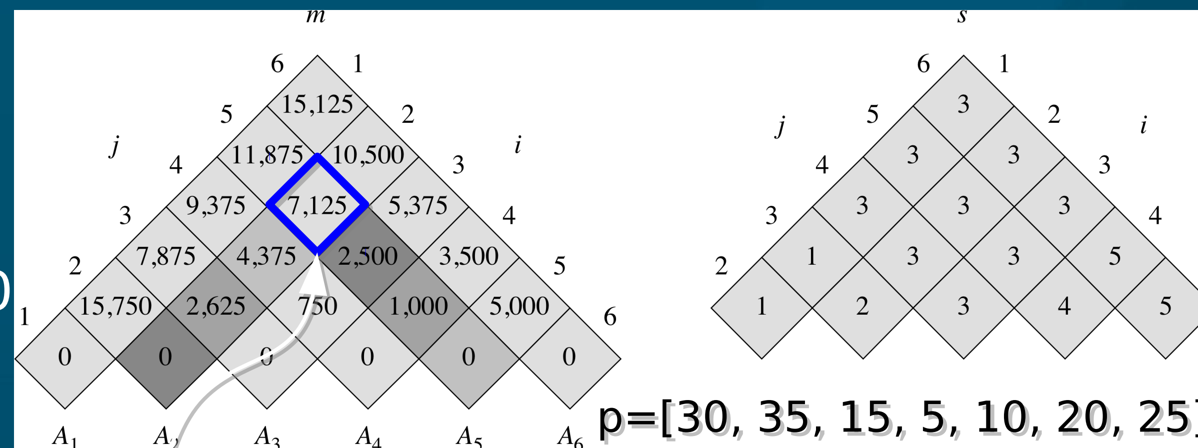
- Given a chain of n matrices (diff dims) to multiply:
  - $(A_1)\ (A_2)\ (A_3)\ \ldots\ (A_n)$
  - $(p_0 \times p_1)\ (p_1 \times p_2)\ (p_2 \times p_3)\ \ldots\ (p_{n-1} \times p_n)$
    - #cols of left matrix = #rows of right matrix
- Any parenthesisation is equivalent:
  which is best to minimise number of operations?
- e.g., (5 x 500) (500 x 2) (2 x 50):
  - Try $(A_1A_2)A_3$: 5*500*2 + 5*2*50 = 5500 ops
  - Try $A_1(A_2A_3)$: 500*2*50 + 5*500*50 = 175000
  - Exhaustive search of parentisisations: $\Theta(2^n)$

Optimal substructure

- As with rod-cutting, consider one split at a time:
  - Cost if split chain i..j at k:
    - Cost(i .. k) + Cost(k+1 .. j) + $(p_{i-1})(p_k)(p_j)$
  - Cost of the matrix mult at the split is $p_{i-1}\ p_k\ p_j$
- Naive recursive solution:
  - def MatChain(p, i, j):
    - if (i == j): return 0
    - return min(foreach(k in i .. j-1:
      MatChain(p, i, k) + MatChain(p, k+1, j)
      + p[i-1] * p[k] * p[j]))
- 2n recursive calls; very inefficient! $\Theta(2^n)$
- Smaller chains are computed repeatedly

Bottom-up solution

- Nodes are indexed by both start (i) and end (j)
  - ⇒ 2D grid of nodes, instead of 1D line



len=4, i=2:
min is @k=3: m[2,3]+m[4,5]+35*5*20

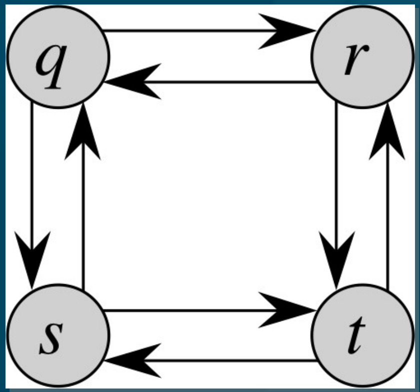
```
def MatChain(p):
    n = length(p) – 1
    m = array[1 .. n][1 .. n] of 0
    s = array[1 .. n-1][2 .. n]
    for len = 2 .. n:
        for i = 1 .. n – len + 1:
            j = i + len - 1
            m[i, j] = infinity
            for k = i .. j - 1:
                q = m[i, k] + m[k+1, j] + p[i-1] * p[k] * p[j]
                if q < m[i, j]:
                    m[i, j] = q
                    s[i, j] = k
```

Outline for today

- Dynamic programming for optimisation
  - Optimal substructure
    - Naive top-down
    - Top-down with memoisation
    - Bottom-up
- Examples:
  - Rod-cutting problem
  - Fibonacci
  - Matrix-chain multiplication
  - Shortest unweighted path
  - Optimal binary search trees

Shortest- and longest-path

- Given a set of nodes and (unweighted) edges, find the shortest path between given nodes u, v:
  - Optimal substructure: if split path at node w, then we can form the shortest path u → w → v from the shortest paths u → w and w → v
  - So we can solve with dynamic programming
- What about longest (non-cyclic) path u → v?
  - Just gluing together Longest(u → w) and Longest(w → v) won't work!
  - Might not be longest u → v
  - Might have loops



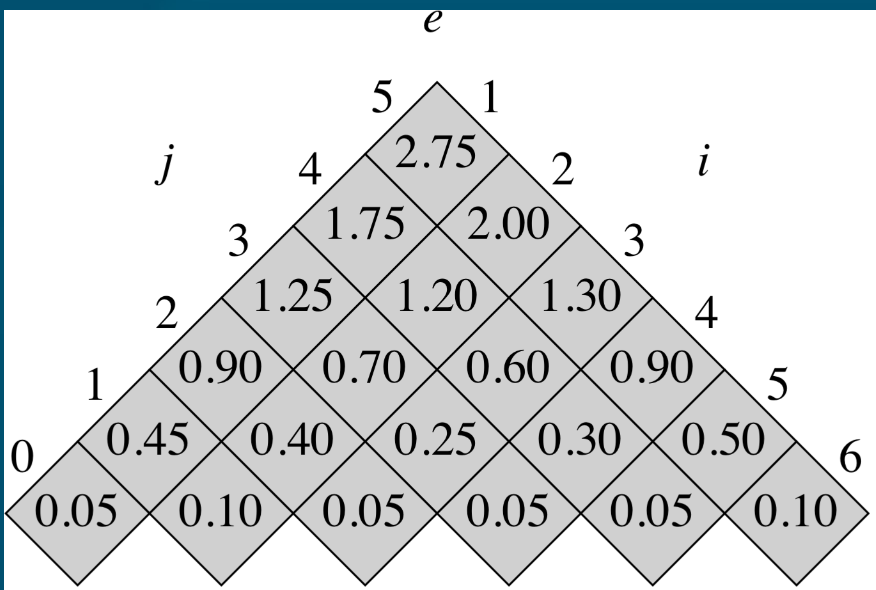
Optimal binary search trees

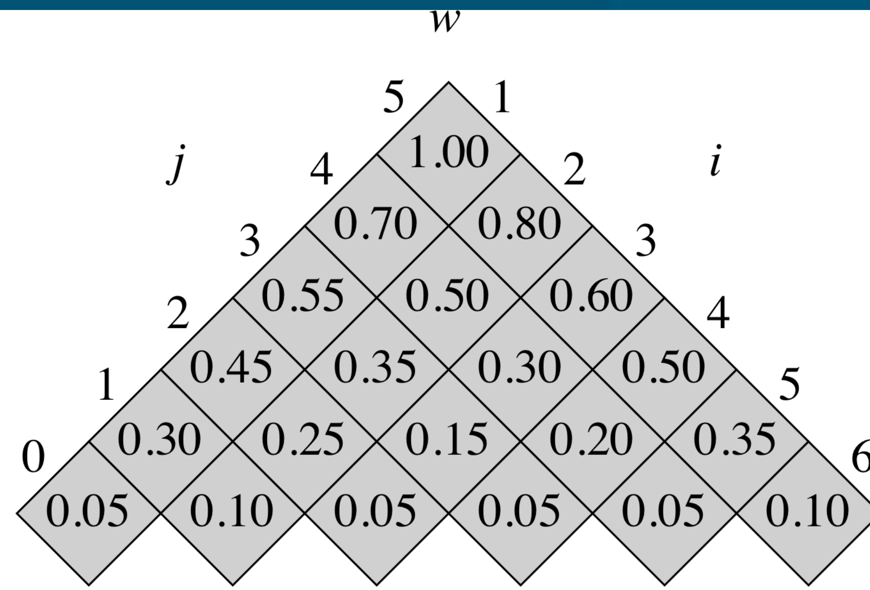
- BST operations $\Theta(h)$: depth of node in tree
- Given sorted set of keys $K = [k_1, \ldots, k_n]$ and probabilities $P = [p_1, \ldots, p_n]$:
  - Minimise expected (weighted avg) search cost
- To handle unsuccessful searches, add dummy keys $d_0, \ldots, d_n$ as leaves:
  - Dummy key d_i is for all values between (k_{i-1}, k_i)
  - Let q_i = probability of d_i: then $\Sigma p + \Sigma q = 1$
- Expected search cost = $\Sigma\,(h(k_i) + 1)p_i + \Sigma\,(h(d_i) + 1)q_i$

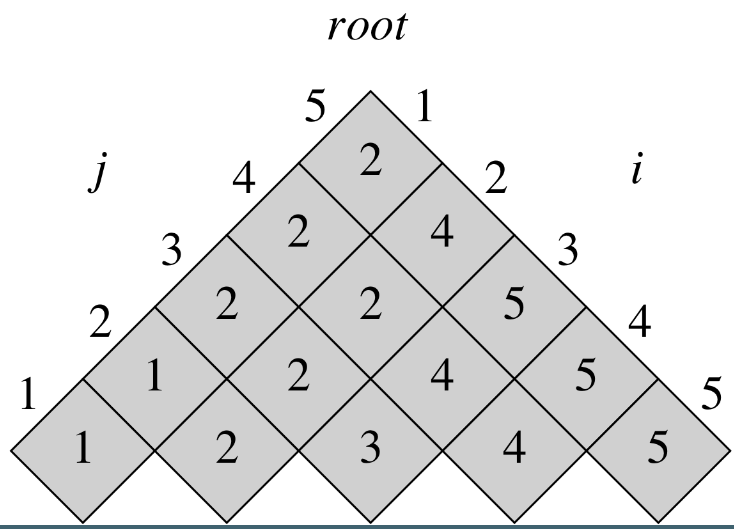
Optimal substructure

- As before, consider one split at a time:
  - "Split" = choice of root
  - To find optimal BST for keys $k_i, \ldots, k_j$,
    - Consider making k_r the root ($i \le r \le j$)
    - Find optimal BST for left subtree $k_i, \ldots, k_{r-1}$
    - Find optimal BST for right subtree $k_{r+1}, \ldots, k_j$
- Demoting a subtree increases depth to each of its nodes by 1: ⇒ increases expected search cost by $w(i,j) = \Sigma_{m=i}^{j} p_m + \Sigma_{m=i-1}^{j} q_m$
- Cost $e(i,j) = \min_{r=i}^{j} [\, e(i, r-1) + e(r+1, j) + w(i, j) \,]$

Optimal BST: example







i	p	q
0		0.05
1	0.15	0.10
2	0.10	0.05
3	0.05	0.05
4	0.10	0.05
5	0.20	0.10