T&R Team of Algorithm Design

College of Computer Science and Engineering, CQU

Algorithm Analysis & Design
Introduction to Algorithm

# 15 DYNAMIC PROGRAMMING

Overlapped sub-problems

Those who cannot remember the past are doomed to repeat it.

-----George Santayana,
The life of Reason,
Book I: Introduction and
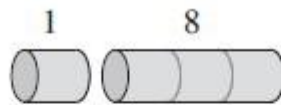Reason in Common
Sense (1905)

Rod cutting

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.

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
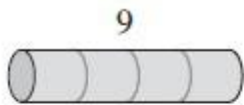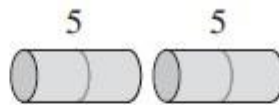
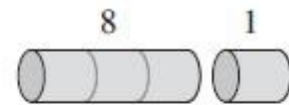

(a) (b) (c) (d)

(e) (f) (g) (h)

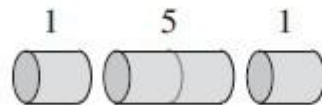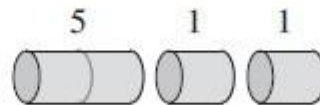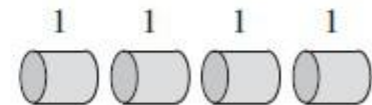For n = 4, which one is best ?
How many different ways of cutting rod with length n ?

Optimal substructure – Rod cutting

In the best solution, we cut i inch from rod at the first time,

$$r(n) = p_i + r(n-i)$$

r (n-i) is totally the same problem with just smaller size.

&

The n-i part of best solution must be the best solution for problem n-i;

Why r(n-i) must be the best solution for problem n-i ?

Recursive top-down implementation

The length of first piece have n possibilities: 1,2,....n inches long

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

CUT-ROD(p, n)

1 **if** $n == 0$
2   **return** 0
3 $q = -\infty$
4 **for** $i = 1$ **to** n
5      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
6 **return** q
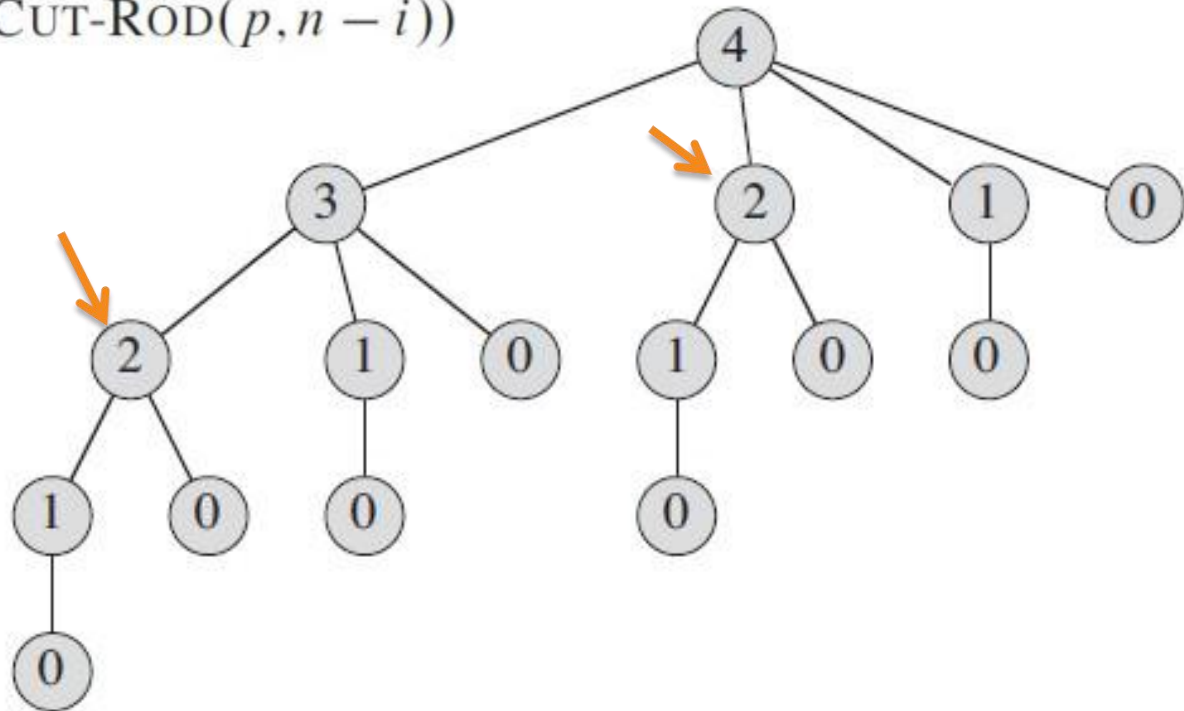
Recursive top-down implementation – good ?

```
CUT-ROD(p, n)
1   if n == 0
2       return 0
3   q = -∞
4   for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6   return q
```

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n) = 2^n$$

Dynamic Programming – top down memoization

MEMOIZED-CUT-ROD(p, n)

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

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

1   **if** $r[n] \geq 0$
2   **return** $r[n]$
3   **if** $n == 0$
4       $q = 0$
5   **else** $q = -\infty$
6   **for** $i = 1$ **to** n
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$
8   $r[n] = q$
9   **return** q

Avoid resolving same sub-problem

Dynamic Programming – Bottom up

BOTTOM-UP-CUT-ROD(p, n)

1   let $r[0 \ldots n]$ be a new array
2    $r[0] = 0$
3   **for** $j = 1$ **to** n
4        $q = -\infty$
5   **for** $i = 1$ **to** j
6            $q = \max(q, p[i] + r[j - i])$
7        $r[j] = q$
8   **return** $r[n]$

Avoid resolving same sub-problem

Dynamic Programming – Bottom up

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

In class exercise:

compute r[1] – r[10]with bottom up method.

Dynamic Programming – Efficiency

Avoid resolving same sub-problem

$$\Theta(n^2)$$
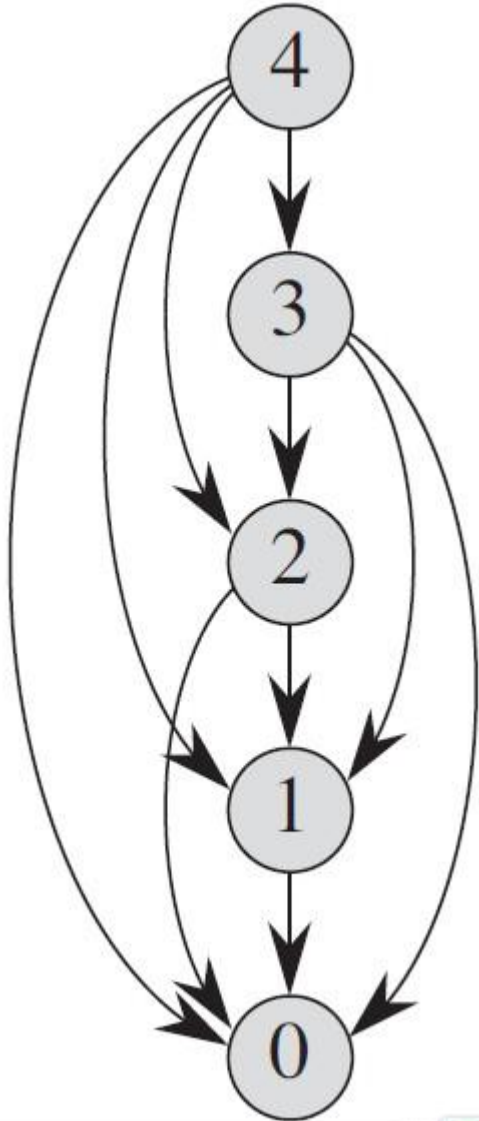
Reconstructing the solution

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1   let r[0..n] and s[0..n] be new arrays
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           if q < p[i] + r[j - i]
7               q = p[i] + r[j - i]
8               s[j] = i
9       r[j] = q
10  return r and s
```

PRINT-CUT-ROD-SOLUTION(p, n)

```
1   (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2   while n > 0
3       print s[n]
4       n = n - s[n]
```

Homework

CLRS 15.1-3
CLRS 15.1-5

Knapsack Problem

The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

0-1 Knapsack problem
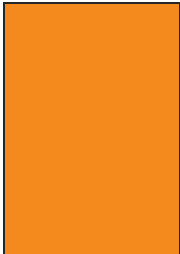
- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items

- Each item *i* has some weight w_i and benefit value b_i (all w_i, b_i and *W* are integer values)

- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

This is a knapsack
Max weight: W = 20

W = 20

The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" each item must be taken or left in entirety
- Generalization, the *unbounded knapsack problem*:
  - No bound on the number of each item
  - Think about the thief ran into a bank…

0-1 Knapsack – Formulation

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0,1\}, 1 \leq i \leq n \end{cases}$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.

Unbounded Knapsack – Formulation

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \displaystyle\sum_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0,1,2,....\}, \ \ 1 \leq i \leq n \end{cases}$$

- The problem is called a *unbounded* problem, because each item has no bound on its number that can be accepted.

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.

- We go through all combinations and find the one with the most total value and with total weight less or equal to W

- Running time will be $O(2^n)$

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

$OPT(k) = \{items\ labeled\ 1,\ 2,\ ..\ k\}$

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for *OPT(k)= {items labeled 1, 2, .. k}*

- This is a valid subproblem definition.

- The question is: can we describe the final solution OPT(n) in terms of subproblems OPT(1),OPT(2),…,OPT(n-1)?

- Unfortunately, we <u>can't</u> do that. Explanation follows….

Defining a Subproblem

| $w_1 = 2$ $b_1 = 3$ | $w_2 = 4$ $b_2 = 5$ | $w_3 = 5$ $b_3 = 8$ | $w_4 = 3$ $b_4 = 4$ | |

?

Max weight: W = 20

For S$_4$:

Total weight: 14; total benefit: 20

OPT(4)

OPT(5)

Item #	Weight W_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

| $w_1 = 2$ $b_1 = 3$ | $w_3 = 4$ $b_3 = 5$ | $w_4 = 5$ $b_4 = 8$ | $w_5 = 9$ $b_5 = 10$ |

For S$_5$:
Total weight: 20 total benefit: 26

OPT(4) is not part of the solution for OPT(5)!!!

- As we have seen, the solution for *OPT(4)* is not part of the solution for OPT(5)

- So our definition of a subproblem is flawed and we need another one!

- Let's add another parameter: *w*, which will represent the <u>exact</u> weight for each subset of items

- The subproblem then will be to compute *OPT[i,w]*

Dynamic Programming – Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  –OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  –new weight limit = w – wi
  –OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w), \ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack – Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w₁,…,wₙ, v₁,…,vₙ

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wᵢ > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vᵢ + M[i-1, w-wᵢ ]}

return M[n, W]
```

Knapsack Algorithm

$W + 1$

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

$n + 1$

OPT: { 4, 3 }
value = 22 + 18 = 40

$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Running time. $\theta(n\,W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Exercise in Class: 0-1 Knapsack Problem

Compute OPT(5,12)

W + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

n + 1

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Q: Unbounded Knapsack – Formulation

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0,1,2,....\}, \ 1 \leq i \leq n \end{cases}$$

- The problem is called a *unbounded* problem, because each item has no bound on its number that can be accepted.

Q: Unbounded Knapsack – Formulation

0-1 Knapsack Problem

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w), \quad v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- Case 1: OPT does not select item i.
  –OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  –new weight limit = w – wi
  –OPT selects best of { 1, 2, …, ? } using this new weight limit

Exercise in Class: Unbounded Knapsack Problem

Compute OPT(5,11)

W + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	34	40

n + 1

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Matrix chain multiplication

- In particular for *1 ≤ i ≤ p* and *1 ≤ j ≤ r*,

$$C[i, j] = \sum_{k = 1 \text{ to } q} A[i, k]\, B[k, j]$$

- Observe that there are *pr* total entries in *C* and each takes *O(q)* time to compute, thus the total time to multiply 2 matrices is *pqr*.
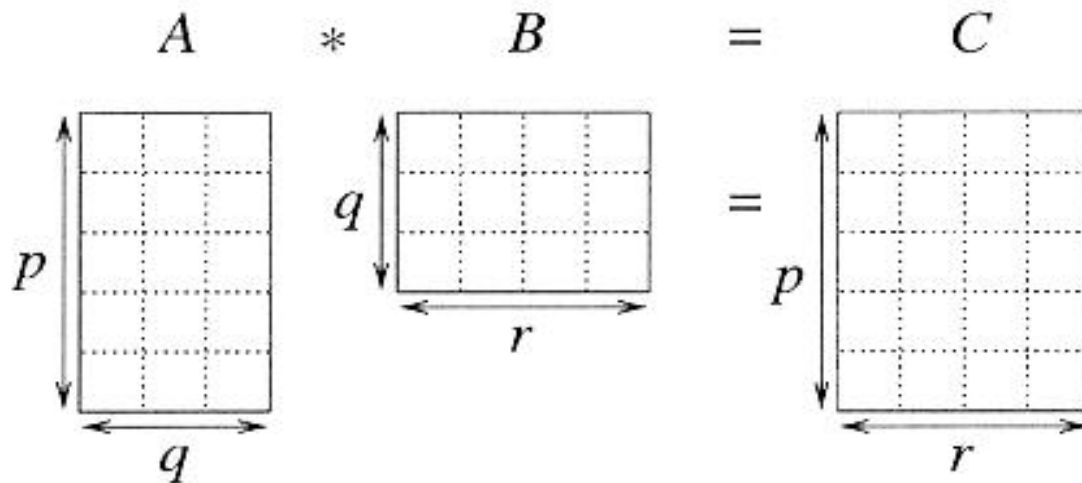
Multiplication time = *pqr*

Matrix chain multiplication

If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix;

MATRIX-MULTIPLY (A, B)

```
1   if A.columns ≠ B.rows
2       error "incompatible dimensions"
3   else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5           for j = 1 to B.columns
6               c_{ij} = 0
7               for k = 1 to A.columns
8                   c_{ij} = c_{ij} + a_{ik} · b_{kj}
9   return C
```

Total times of multiplication is p x q x r;

Matrix chain multiplication

- Given a sequence of matrices $A_1 A_2 \ldots A_n$, and dimensions $p_0 p_1 \ldots p_n$ where A_i is of dimension p_{i-1} x p_i, determine multiplication sequence that minimizes the number of operations.

- This algorithm does not perform the multiplication, it just figures out the best order in which to perform the multiplication.

Matrix chain multiplication

$A = 50 \times 10$ $B = 10 \times 40$ $C = 40 \times 30$ $D = 30 \times 5$

Matrix multiplication is associative, and so all parenthesizations yield the same product.

$(A((BC)D))$ $(A(B(CD)))$ $((AB)(CD))$

$(((AB)C)D)$ $((A(BC))D)$

16000, 10500, 36000, 87500, 34500

Counting the number of parenthesizations

- If we have just 1 item, then there is only one way to parenthesize. If we have *n* items, then there are *n-1* places where you could break the list with the outermost pair of parentheses, namely just after the first item, just after the 2nd item, etc. and just after the (*n-1*)th item.

- When we split just after the *k*th item, we create two sub-lists to be parenthesized, one with *k* items and the other with *n-k* items. Then we consider all ways of parenthesizing these. If there are *L* ways to parenthesize the left sub-list, *R* ways to parenthesize the right sub-list, then the total possibilities is *L∗R*.

Counting the number of parenthesizations

$$P(n) = \begin{cases} 1 & n = 1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

Catalan number

$$P(n) = \begin{cases} 1 & n = 1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases} \Rightarrow P(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

$$P(n) = C(n-1)$$

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n / n^{3/2})$$

DP Solution(I)

- Let $A_{i...j}$ be the product of matrices i through j. $A_{i...j}$ is a p_{i-1} x p_j matrix. At the highest level, we are multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1...n} = (A_{1...k})(A_{k+1...n})$$

- The problem of determining the optimal sequence of multiplication is broken up into 2 parts:
    Q : How do we decide where to split the chain (what k)?
    A : Consider all possible values of k.
    Q : How do we parenthesize the subchains $A_{1...k}$ & $A_{k+1...n}$?
    A : Solve by recursively applying the same scheme.
    NOTE: this problem satisfies the "*principle of optimality*".

- Next, we store the solutions to the sub-problems in a table and build the table in a bottom-up manner.

DP Solution(II)

- For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i \ldots j}$.

- Example: Minimum number of multiplies for $A_{3 \ldots 7}$

$$A_1 A_2 \underbrace{A_3 A_4 A_5 A_6 A_7}_{m[3,7]} A_8 A_9$$

- **In terms of p_i , the product $A_{3 \ldots 7}$ has dimensions ____.**

DP Solution(III)

- The optimal cost can be described be as follows:
  - $i = j \Rightarrow$ the sequence contains only 1 matrix, so $m[i, j] = 0$.
  - $i < j \Rightarrow$ This can be split by considering each k, $i \leq k < j$,

    as $A_{i...k}$ (p_{i-1} x p_k) times $A_{k+1...j}$ (p_k x p_j).

- This suggests the following recursive rule for computing $m[i, j]$:

  $m[i, i] = 0$

  $m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$ for $i < j$

Computing m[i,j]

- For a specific k,

$$(A_i ... A_k)(A_{k+1} ... A_j)$$

$$=$$

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j)$$

Computing m[i,j]

- For a specific k,

$$(A_i ... A_k)(A_{k+1} ... A_j)$$

$$= A_{i...k}(A_{k+1} ... A_j) \qquad (m[i, k] \text{ mults})$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

Computing m[i,j]

- For a specific k,

  $(A_i ... A_k)(A_{k+1} ... A_j)$

  $= A_{i...k}(A_{k+1} ... A_j)$ ($m[i, k]$ mults)

  $= A_{i...k} A_{k+1...j}$ ($m[k+1, j]$ mults)

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Computing m[i,j]

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$$= A_{i\ldots k}(A_{k+1} \ldots A_j) \qquad\qquad (m[i, k] \text{ mults})$$

$$= A_{i\ldots k} A_{k+1\ldots j} \qquad\qquad (m[k+1, j] \text{ mults})$$

$$= A_{i\ldots j} \qquad\qquad (p_{i-1} \, p_k \, p_j \text{ mults})$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

Computing m[i,j]

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$$= A_{i \ldots k}(A_{k+1} \ldots A_j) \qquad (m[i, k] \text{ mults})$$

$$= A_{i \ldots k} A_{k+1 \ldots j} \qquad (m[k+1, j] \text{ mults})$$

$$= A_{i \ldots j} \qquad (p_{i-1} p_k p_j \text{ mults})$$

- For solution, evaluate for all k and take minimum.

$$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

Applying dynamic programming

$$A_i A_{i+1} ... A_j = (A_i A_{i+1} ... A_k)(A_{k+1} A_{k+2} ... A_j)$$

$$A_i \quad : \qquad p_{i-1} \times p_i$$

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & i < j \end{cases}$$

Implementation

MATRIX-CHAIN-ORDER(p)

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4          m[i, i] = 0
5   for l = 2 to n                       // l is the chain length
6          for i = 1 to n − l + 1
7                  j = i + l − 1
8                  m[i, j] = ∞
9                  for k = i to j − 1
10                         q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11                         if q < m[i, j]
12                                 m[i, j] = q
13                                 s[i, j] = k
14  return m and s
```
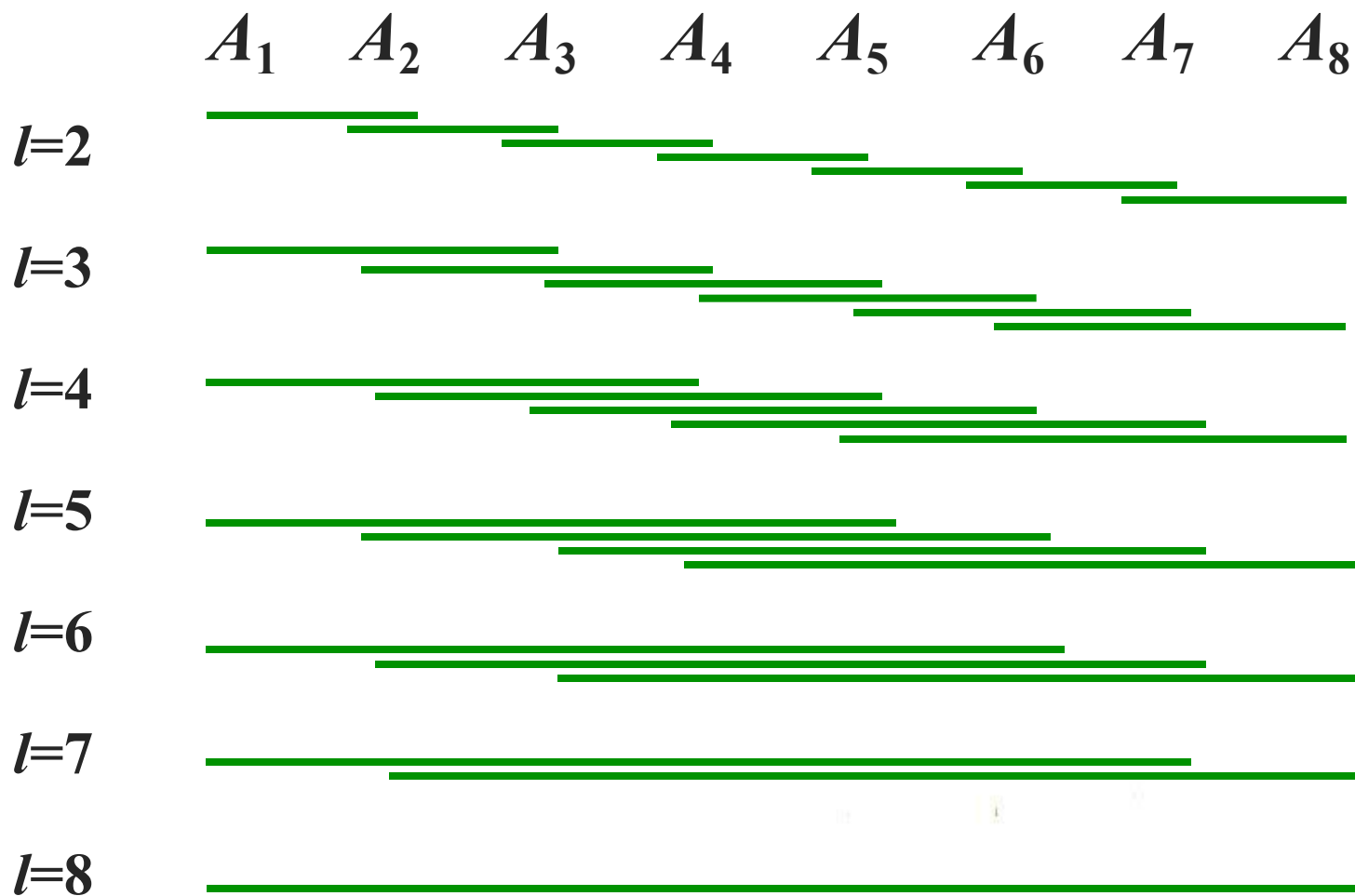
Applying dynamic programming

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25
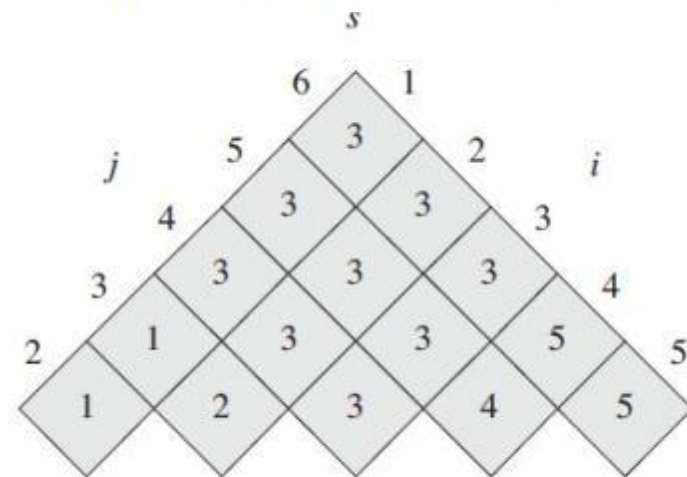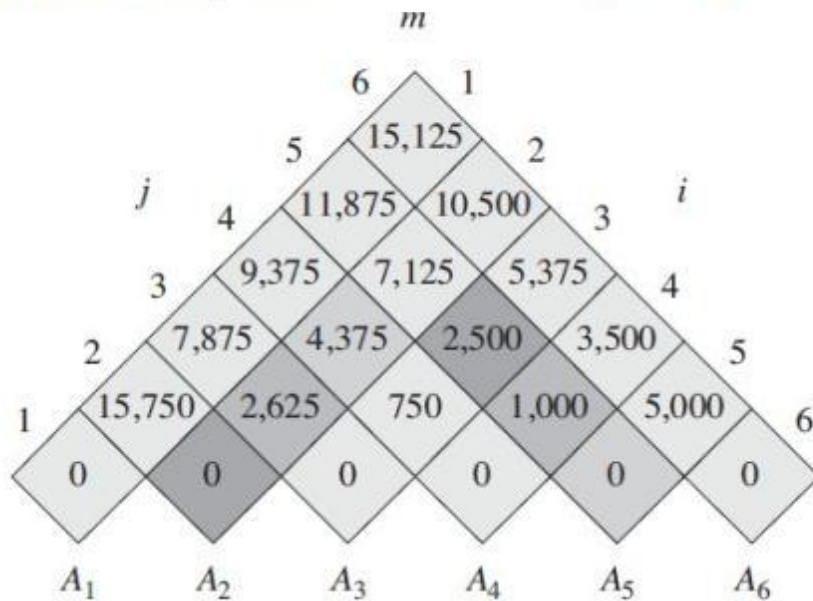
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases}$$

$$= 7125.$$

s[2,5] = 3

Analysis

- The array $s[i, j]$ is used to extract the actual sequence (see next).

- There are 3 nested loops and each can iterate at most n times, so the total running time is $\Theta(n^3)$.

For $A_1*A_2...*A_{n-1}*A_n$, How many different sub-problems ?

Extracting Optimum Sequence

- Leave a split marker indicating where the best split is (i.e. the value of k leading to minimum values of $m[i, j]$). We maintain a parallel array $s[i, j]$ in which we store the value of k providing the optimal split.

- If $s[i, j] = k$, the best way to multiply the sub-chain $A_{i...j}$ is to first multiply the sub-chain $A_{i...k}$ and then the sub-chain $A_{k+1...j}$, and finally multiply them together. Intuitively $s[i, j]$ tells us what multiplication to perform *last*. We only need to store $s[i, j]$ if we have at least 2 matrices & $j > i$.

Constructing optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1   if i == j
2       print "A"ᵢ
3   else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Mult (*A, i, j*)

1. if (*j* > *i*)

2.   then *k = s*[*i, j*]

3.   *X* = Mult(*A, i, k*)   *// X = A*[*i*]*…A*[*k*]

4.   *Y* = Mult(*A, k+1, j*)   *// Y = A*[*k+1*]*…A*[*j*]

5.   return *X*Y*   *//* Multiply *X*Y*

6.   else return *A*[*i*]   *//* Return *i*th matrix

Exercise in Class

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	20 X 30	30 x 35	35 x 10	10 x 20	20 x 15	15 x 25
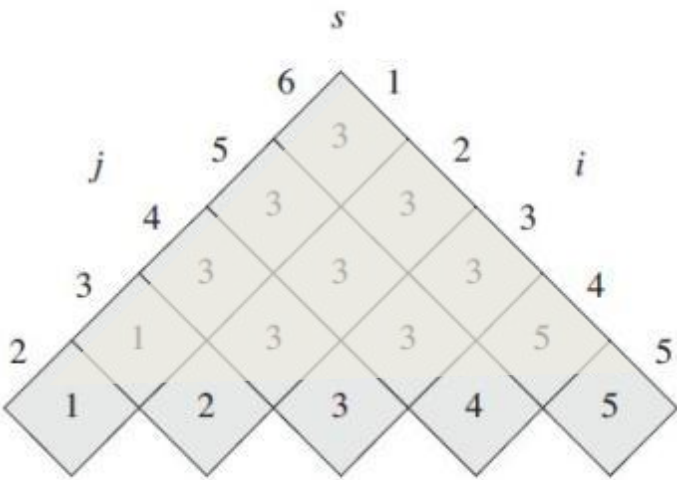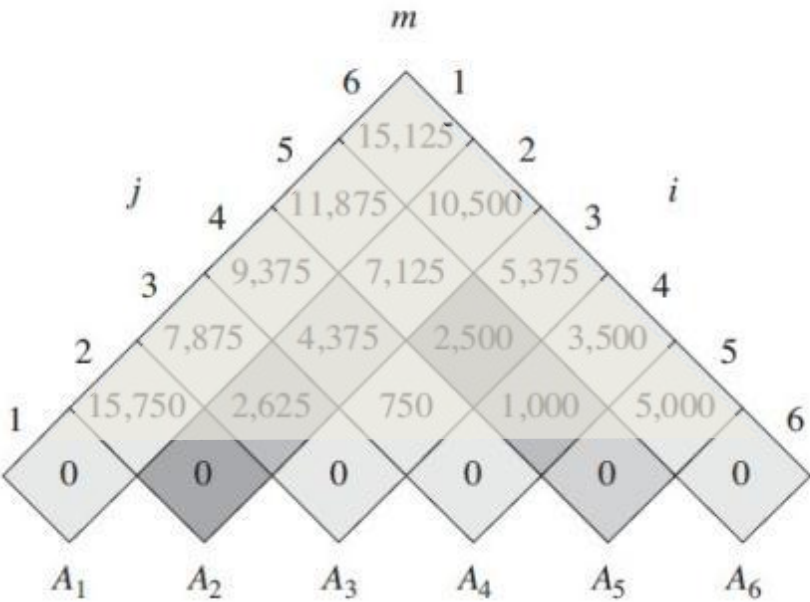
- Figure out the "top-level" choice you have to make (e.g., where to split the list of matrices)

- List the options for that decision

- Each option should require smaller sub-problems to be solved

- Recursive function is the minimum (or max) over all the options

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

- Think what decision is the "last piece in the puzzle"
  - Where to place the outermost parentheses in a matrix chain multiplication

$$(A_1) \, (A_2 \, A_3 \, A_4)$$

$$(A_1 \, A_2) \, (A_3 \, A_4)$$

$$(A_1 \, A_2 \, A_3) \, (A_4)$$

DP Step 2

- Ask what subproblem(s) would have to be solved to figure out how good your choice is
  - How to multiply the two groups of matrices, e.g., this one (A_1) (trivial) and this one ($A_2 \ A_3 \ A_4$)

DP Step 3

- Write down a formula for the "goodness" of the best choice

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_k p_j)$$

DP Step 4

- Arrange subproblems in order from small to large and solve each one, keeping track of the solutions for use when needed

- Need 2 tables
  - One tells you value of the solution to each subproblem
  - Other tells you last option you chose for the solution to each subproblem

Matrix-Chain-Order(*p*)

1. $n \leftarrow length[p]$ - 1
2. for $i \leftarrow 1$ to n // initialization: $O(n)$ time
3.   do $m[i, i] \leftarrow 0$
4.   for $L \leftarrow 2$ to n // L = length of sub-chain
5.   do for $i \leftarrow 1$ to n - $L+1$
6.   do $j \leftarrow i + L - 1$
7. $m[i, j] \leftarrow \infty$
8.   for $k \leftarrow i$ to $j - 1$
9.   do $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
10.   if $q < m[i, j]$
11.   then $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. return m and s

Homework

CLRS 15.2-1

Elements of Dynamic Programming

- Optimal substructure
- Overlapping subproblems

Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the **solutions to the subproblems used within the optimal solution must themselves be optimal.** Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
  - 1. *How many subproblems* are used in an optimal solution.
  - 2. *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).
- How many subproblems and choices do the examples considered contain?
- Dynamic programming uses optimal substructure **bottom up**.
  - *First* find optimal solutions to subproblems.
  - *Then* choose which to use in optimal solution to the problem.

Optimal Substucture

- Does optimal substructure apply to all optimization problems? No.

- Applies to determining the shortest path but NOT the longest simple path of an unweighted directed graph.

- Why?

  - Shortest path has independent subproblems.

  - Solution to one subproblem does not affect solution to another subproblem of the same problem.

  - Subproblems are not independent in longest simple path.

    - Solution to one subproblem affects the solutions to other subproblems.
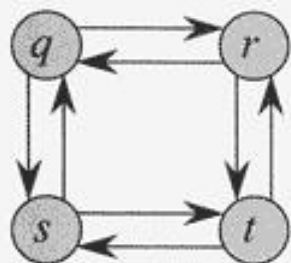
Figure 15.4 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

Overlapping Subproblems

- The space of subproblems must be "small".
- The total number of distinct subproblems is a polynomial in the input size.
  - A recursive algorithm is exponential because it solves the same problems repeatedly.
  - If divide-and-conquer is applicable, then each problem solved will be brand new.
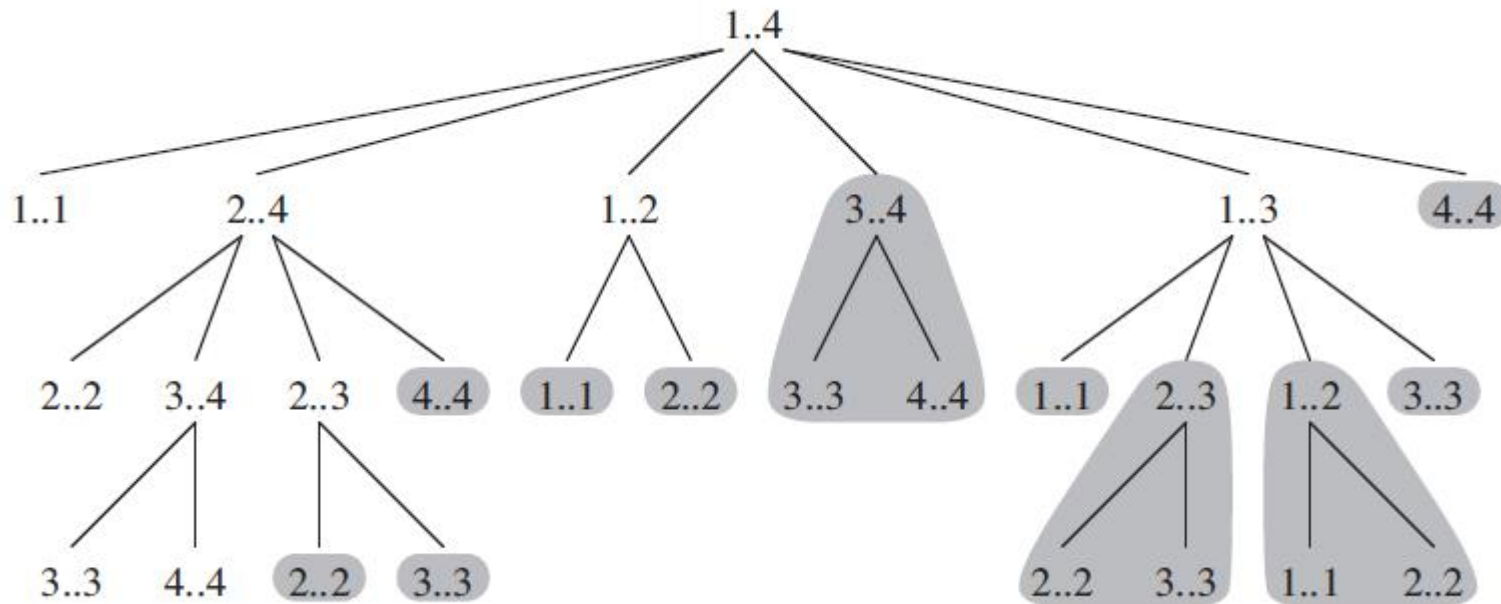
Dynamic Programming

- Similar to divide-and-conquer, it breaks problems down into smaller problems that are solved recursively.

- In contrast, DP is applicable when the sub-problems are not independent, i.e. when sub-problems share sub-sub-problems. It solves every sub-sub-problem just once and save the results in a table to avoid duplicated computation.

Elements of DP Algorithms

- **Sub-structure:** decompose problem into smaller sub-problems. Express the solution of the original problem in terms of solutions for smaller problems.

- **Table-structure:** Store the answers to the sub-problem in a table, because sub-problem solutions may be used many times.

- **Bottom-up computation:** combine solutions on smaller sub-problems to solve larger sub-problems, and eventually arrive at a solution to the complete problem.

Applicability to Optimization Problems

- **Optimal sub-structure (principle of optimality):** for the global problem to be solved optimally, each sub-problem should be solved optimally. This is often violated due to sub-problem overlaps. Often by being "less optimal" on one problem, we may make a big savings on another sub-problem.

- **Small number of sub-problems:** Many NP-hard problems can be formulated as DP problems, but these formulations are not efficient, because the number of sub-problems is exponentially large. Ideally, the number of sub-problems should be at most a polynomial number.

Homework

CLRS 15.3-2

CLRS 15.3-3

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,...,x_m \rangle$ and $Y = \langle y_1,...,y_n \rangle$, find a common subsequence whose length is maximum.

springtime   ncaa tournament   basketball

printing   north carolina   snoeyink

Subsequence need not be consecutive, but must be in order.

Naïve Algorithm

- For every subsequence of *X*, check whether it's a subsequence of *Y* .
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of *X* to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan *Y* for first letter, for second, and so on.

Optimal Substructure

> **Theorem**
>
> Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.
> 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
> 2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
> 3.   or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Notation:

prefix $X_i = \langle x_1, \ldots, x_i \rangle$ is the first i letters of X.

This says what any longest common subsequence must look like; do you believe it?

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y .

3.   or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

(1)   longest common subsequence (LCS) Z must end in $x_m = y_n$.

(2) Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}, and

(3)   there is no longer CS of X_{m-1} and Y_{n-1}, or Z would not be an LCS.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.

3.   or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of X_{m-1} and Y, and

(2) there is no longer CS of X_{m-1} and Y, or Z would not be an LCS.

Recursive Solution

- Define *c[i, j]* = length of LCS of *X$_i$* and *Y$_j$* .

- We want *c[m,n]*.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem. But does it solve it well?

Recursive Solution

$$c[\alpha, \beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\,\alpha, prefix\,\beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\,\alpha, \beta], c[\alpha, prefix\,\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$

$c[\text{springtime, printing}]$

$c[\text{springtim, printing}]$ $c[\text{springtime, printin}]$

$[\text{springti, printing}]$ $[\text{springtim, printin}]$ $[\text{springtim, printin}]$ $[\text{springtime, printi}]$

$[\text{springt, printing}]$ $[\text{springti, printin}]$ $[\text{springtim, printi}]$ $[\text{springtime, print}]$

Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta]+1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\alpha, \beta], c[\alpha, prefix\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$

• Keep track of $c[\alpha,\beta]$ in a table of nm entries:

  • top/down

  • bottom/up

		p	r	i	n	t	i	n	g
s									
p									
r									
i									
n									
g									
t									
i									
m									
e									

Computing the length of an LCS

LCS-LENGTH (*X, Y*)

1. *m ← length*[*X*]
2. *n ← length*[*Y*]
3. **for** *i ←* 1 **to** *m*
4.   **do** *c*[*i*, 0] ← 0
5. **for** *j ←* 0 **to** *n*
6.   **do** *c*[0, *j*] ← 0
7. **for** *i ←* 1 **to** *m*
8.   **do for** *j ←* 1 **to** *n*
9.   **do if** $x_i = y_j$
10.   **then** *c*[*i, j*] ← *c*[*i*−1, *j*−1] + 1
11.   *b*[*i, j*] ← "↖"
12.   **else if** *c*[*i*−1, *j*] ≥ *c*[*i, j*−1]
13.   **then** *c*[*i, j*] ← *c*[*i*− 1, *j*]
14.   *b*[*i, j*] ← "↑"
15.   **else** *c*[*i, j*] ← *c*[*i, j*−1]
16.   *b*[*i, j*] ← "←"
17. **return** *c* and *b*

b[*i, j*] points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

c[*m,n*] contains the length of an LCS of *X* and *Y*.

Time: $O(mn)$

Constructing an LCS

PRINT-LCS (b, X, i, j)
1. **if** $i = 0$ or $j = 0$
2.   **then return**
3. **if** $b[i, j] = $ " ↖ "
4.   **then** PRINT-LCS($b, X, i-1, j-1$)
5.   print x_i
6.   **elseif** $b[i, j] = $ "↑"
7.   **then** PRINT-LCS($b, X, i-1, j$)
8. **else** PRINT-LCS($b, X, i, j-1$)

- Initial call is PRINT-LCS (b, X, m, n).
- When $b[i, j] = $ ↖, we have extended LCS by one character. So LCS = entries with ↖ in them.
- Time: $O(m+n)$

Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B,$ $D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

CLRS 15.4-1
CLRS 15.4-3

- ## Problem:

  - The Palmia country is divided by a river into the north and south bank. There are N towns on both the north and south bank. Each town on the north bank has its unique friend town on the south bank. No two towns have the same friend. Each pair of friend towns would like to have a ship line connecting them. They applied for permission to the government. Because it is often foggy on the river the government decided to prohibit intersection of ship lines (if two lines intersect there is a high probability of ship crash).

- Problem (in mathematical terms)
  - There exists a sequences $a_1, a_2, \ldots a_n$ and
    $b_1, b_2, \ldots b_n$ such that
    $a_s < a_j$ if s<j
    $b_s < b_j$ if s<j
    for each s, there exists a s' such that a_s and $b_{s'}$
    connected.
  - Find the **maximum** value m such that there exist
    $x_1 < x_2 < x_3 < \ldots x_m.$ m<=n such that
  - $b_{x0}, < b_{x1}, < b_{x2}, < \ldots b_{xm},$

Dynamic Programming

- Example Problem

a: 2   4   9   10   15   17   22

b: 2   3   4   6   8   12   17

- Example Problem

a: 2   4   9   10   15   17   22

b: 2   3   4   6   8   12   17

- **Simple Brute Force Algorithm**

- Pick all possible sets of routes.

- Check if selection of routes is valid
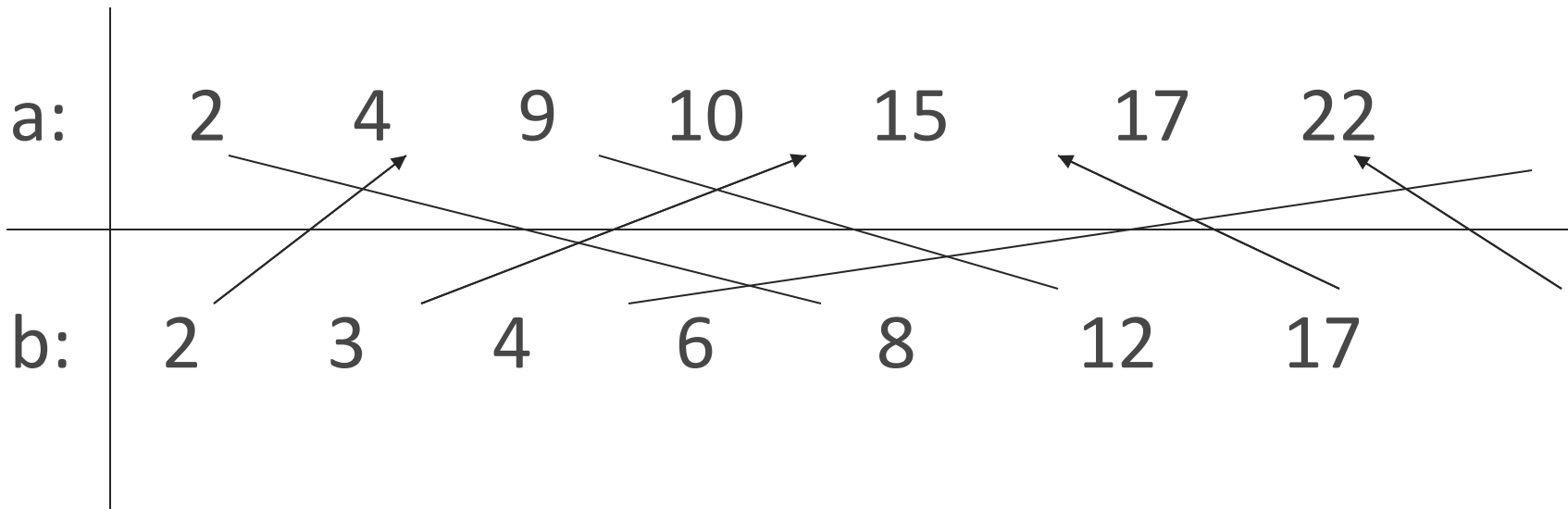
Dynamic Programming

- **Simple Brute Force Algorithm (Analysis)**

- Pick all possible sets of routes. …. $O(2^n)$ possible routes

- Check if selection of routes is valid… $O(n)$ time to check

- In total will take $O(2^n * n)$ time.

- EXPONENTIAL TIME = SLOW!!!!!

- Elegant Solution

- Let c(i,j) be the maximum possible routes using the first *i* sequences in a, and j sequences in b.

- We notice that c(I,j) is connected to c(i-1,j), c(i,j-1) and c(i-1,j-1).

RULE 1 of Dynamic Programing

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic Programming

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } a[i] = b[j], \\ \max\{c[i-1,j], c[i,j-1]\} \end{cases}$$

Proof. Case $a[i] = b[j]$:
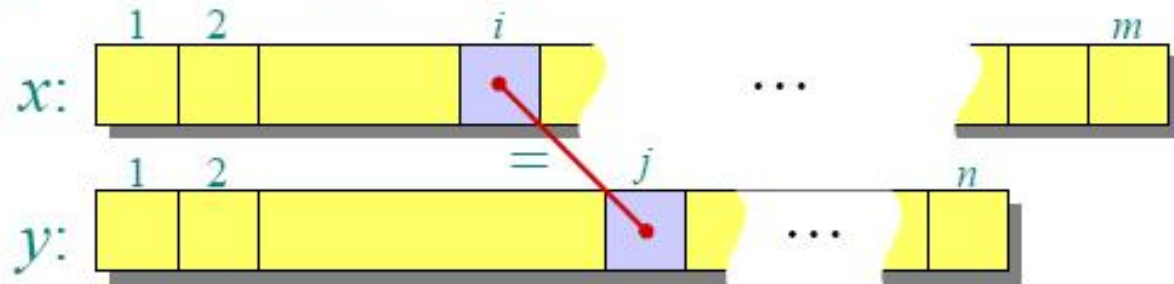
We notice that if a[i] = b[j] (connected), then we can use them in our solution and optimal solution becomes c[i-1,j-1]+1(because we just used a new route).
In all other cases, we just consider the case where we do not use b[j] (i.e c[i,j-1])
or we do not use a[i] (i.e c[i-1,j]

Another example: Sequence partitioning

- Sequence of n tasks to do <u>in order</u>
- Let amount of work per task be $s_1, s_2, \ldots s_n$
- Divide into k shifts so that no shift gets too much work
  - i.e., minimize the max amount of work on any shift

- Note: solution at http://snipurl.com/23c2xrn
- What is the runtime? Can we improve it?

- Variant: Could use more than k shifts, but an extra cost for adding each extra shift

Another example: Sequence partitioning

Divide sequence of n=9 tasks into k=4 shifts – need to place 3 boundaries

Branch and bound: place 3rd boundary, then 2nd, then 1st

5 2 3 7 6 8 1 9

5 2 3 7 6 8 1 9 | 5 2 3 7 6 8 1 | 9   5 2 3 7 6 8 | 1 9   5 2 3 7 6 | 8 1 9 ...

can prune: already
know longest shift ≥ 18

... 5 2 3 7 6 | 8 1 | 9   5 2 3 7 | 6 8 1 | 9 ...   ... 5 2 3 7 6 | 8 | 1 9   5 2 3 7 | 6 8 | 1 9 ...

can prune: already
know longest shift ≥ 14

... 5 2 3 | 7 6 | 8 1 | 9   ... 5 2 | 3 7 | 6 8 1 | 9   ... 5 2 3 | 7 6 | 8 | 1 9 ...

Longest shift
has length 13

Longest shift
has length 15

Longest shift
has length 13
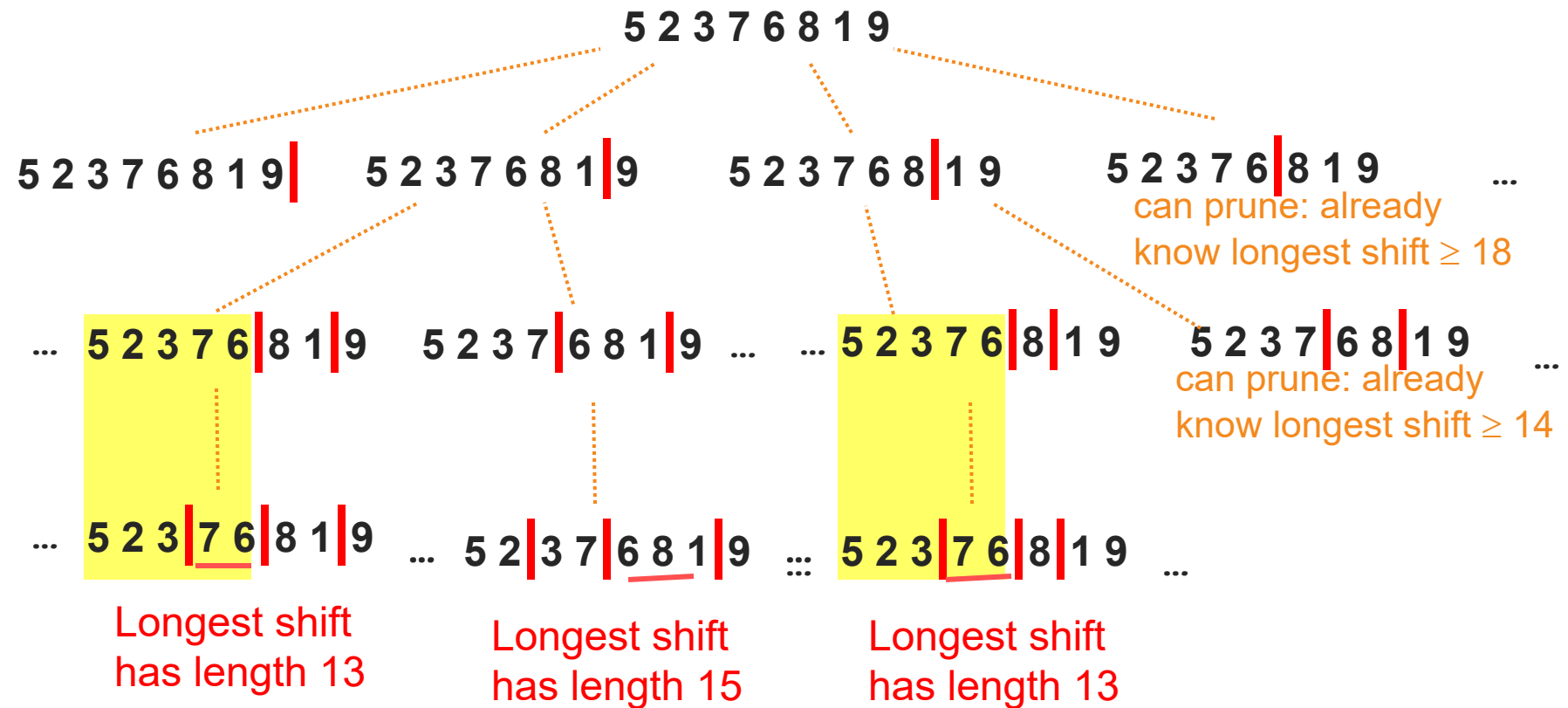
These are really solving the same subproblem (n=5, k=2)
Longest shift in this subproblem = 13
So longest shift in full problem = max(13,9) or max(13,10)

96

Another example: Sequence partitioning

Divide sequence of N tasks into K shifts

- int best(N,K):   *// memoize this!*
    - if K=0   *// have to divide N tasks into 0 shifts*
        - if N=0 then return 0 else return ∞   *// impossible for N > 0*
    - else   *// consider # of tasks in last shift*
        - bestanswer = ∞   *// keep a running minimum here*
        - lastshift = 0   *// total work currently in last shift*
        - while N ≥ 0   *// number of tasks not currently in last shift*
            - if (lastshift > bestglobalsolution) then break   *// prune node*
            - bestanswer min=   max(best(N,K-1),lastshift)
            - lastshift += s[N]   *// move another task into last shift*
            - N = N-1
        - return bestanswer

Another example: Sequence partitioning

Divide sequence of N tasks into K shifts

- Dyna version?

Recursive Solution

- Define *d*[*n, k*] = optimal solution for dividing n tasks into k shifts .

- We want d[*N,K*].

- s[i,j]= s[i]+s[i+1]+…+s[j], (s[i,j]=0, if i>j)

$$d[n,k] = \begin{cases} 0 & n = 0 \\ \min_{0 \leq i \leq n} (\max (d[n-i, k-1], s[n-i+1, n])) & n, k > 0 \\ \infty & k = 0 \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

- Define *d*[*n, k*] = optimal solution for dividing n tasks into k shifts .

- We want d[*N,K*].

- s[i,j]= s[i]+s[i+1]+…+s[j]

$$d[n,k] = \begin{cases} 0 & n = 0 \\ \min_{1 \le i \le P} \left(\max \left(d[n-i, k-1], \, s[n-i+1, n] \right) \right) & \begin{array}{l} k > 0, n > P > 0, \\ s[n-P+1, n] \ge d[n-P, k-1] \end{array} \\ \infty & k = 0 \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Optimal binary search trees

- e.g. binary search trees for 3, 7, 9, 12;
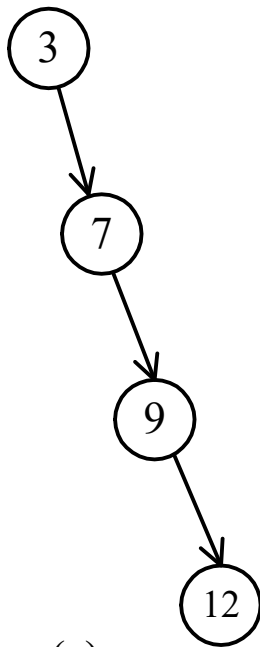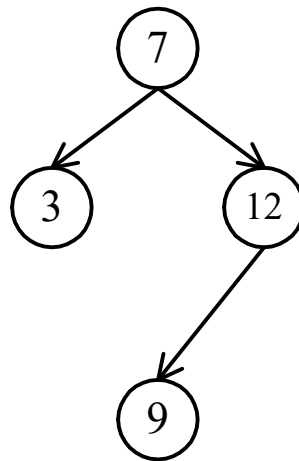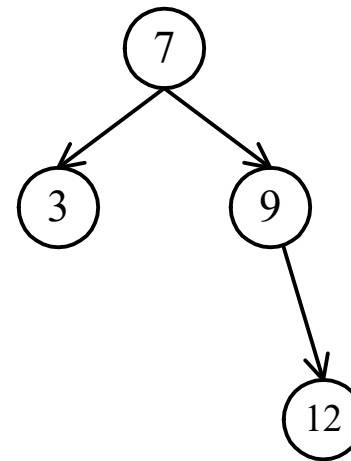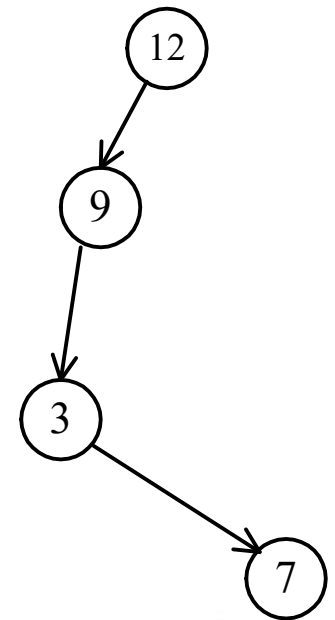
(a)   (b)   (c)   (d)

Optimal Binary Search Trees

- ## Problem

  - Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i.

  - Want to build a binary search tree (BST) with minimum expected search cost.

  - Actual cost = # of items examined.

  - For key k_i, cost = $\text{depth}_T(k_i)+1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T.

Expected Search Cost

$E[\text{search cost in } T]$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \qquad (1)$$

Sum of probabilities is 1.

Example

- Consider 5 keys with these search probabilities:
  $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.
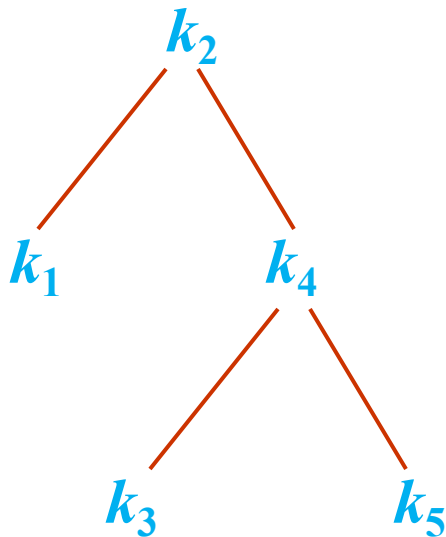
i	depth$_T(k_i)$	depth$_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, E[search cost] = 2.15.

Example

- $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.
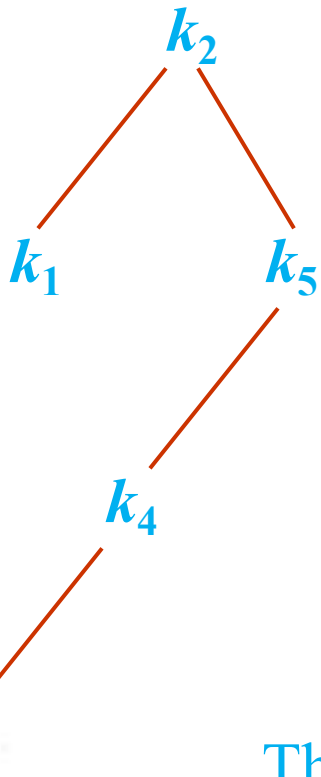
i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, E[search cost] = 2.10.

This tree turns out to be optimal for this set of keys.

Example

- **Observations:**
  - Optimal BST may not have smallest height.
  - Optimal BST may not have highest-probability key at root.
- Build by exhaustive checking?
  - Construct each n-node BST.
  - For each,
       assign keys and compute expected search cost.
  - But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Optimal Substructure

- Any subtree of a BST contains keys in a contiguous range $k_i, ..., k_j$ for some $1 \le i \le j \le n$.
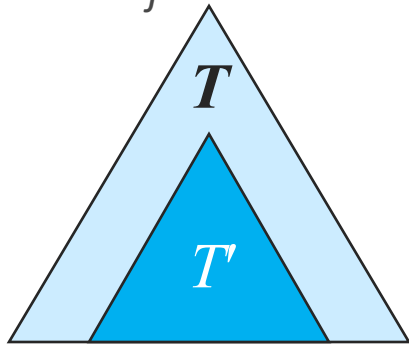
- If T is an optimal BST and
  T contains subtree T' with keys $k_i, ... , k_j$,
  then T' must be an optimal BST for keys $k_i, ..., k_j$.
- **Proof:** Cut and paste.

Optimal Substructure

- One of the keys in $k_i, ..., k_j$, say k_r, where $i \le r \le j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, ..., k_{r-1}$.
- Right subtree of k_r contains $k_r+1, ..., k_j$.
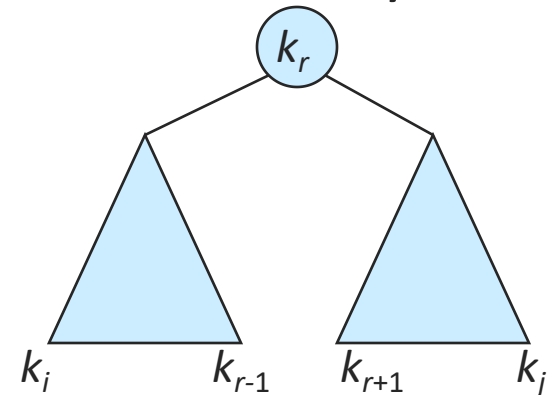
- To find an optimal BST:
  - Examine all candidate roots k_r, for $i \le r \le j$
  - Determine all optimal BSTs containing $k_i, ..., k_{r-1}$ and containing $k_{r+1}, ..., k_j$

Recursive Solution

- Find optimal BST for $k_i,...,k_j$, where $i \geq 1$, $j \leq n$, $j \geq i-1$. When $j = i-1$, the tree is empty.

- Define $e[i, j]$ = expected search cost of optimal BST for $k_i,...,k_j$.

- If $j = i-1$, then $e[i, j] = 0$.

- If $j \geq i$,
  - Select a root k_r, for some $i \leq r \leq j$.
  - Recursively make an optimal BSTs
    - for $k_i,..,k_{r-1}$ as the left subtree, and
    - for $k_{r+1},..,k_j$ as the right subtree.

Recursive Solution

- When the OPT subtree becomes a subtree of a node:
  - Depth of every node in OPT subtree goes up by 1.
  - Expected search cost increases by

  $$w(i, j) = \sum_{l=i}^{j} p_l$$ from Eq. 1

- If k_r is the root of an optimal BST for $k_i,..,k_j$:
  - $e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$

    $= e[i, r-1] + e[r+1, j] + w(i, j)$.   (because $w(i, j) = w(i, r-1) + p_r + w(r + 1, j)$)

- But, we don't know k_r. Hence,

$$e[i, j] = \begin{cases} 0 & \text{if } j = i - 1 \\ \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j \end{cases}$$

Computing an Optimal Solution

For each subproblem (*i,j*), store:

- expected search cost in a table $e[1 .. n+1 , 0 .. n]$
  - Will use only entries $e[i, j]$, where $j \geq i-1$.

- root[*i, j*] = root of subtree with keys $k_i,..,k_j$, for $1 \leq i \leq j \leq n$.

- $w[1..n+1, 0..n]$ = sum of probabilities
  - $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$.

Pseudo-code

OPTIMAL-BST(p, q, n)
1.   **for** $i \leftarrow 1$ **to** $n + 1$
2.   **do** $e[i, i-1] \leftarrow 0$
3. $w[i, i-1] \leftarrow 0$
4.   **for** $l \leftarrow 1$ **to** n
5.   **do for** $i \leftarrow 1$ **to** $n-l+1$
6.   **do** $j \leftarrow i + l - 1$
7. $e[i, j] \leftarrow \infty$
8. $w[i, j] \leftarrow w[i, j-1] + p_j$
9.   **for** $r \leftarrow i$ **to** j
10.   **do** $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$
11.   **if** $t < e[i, j]$
12.   **then** $e[i, j] \leftarrow t$
13. $root[i, j] \leftarrow r$
14. **return** e and $root$

Consider all trees with l keys.

Fix the first key.

Fix the last key

Determine the root of the optimal (sub)tree

Time: $O(n^3)$

Homework

CLRS 15.5-2

Sequence Alignment

Sequence Alignment

How similar are two strings ?

- ocurrance
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps

Edit Distance

- Basis for Unix diff.
- Speech recognition.
- Computational biology.c

Edit Distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.

C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	T	A	C	A	T

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$$2\delta + \alpha_{CA}$$

Sequence Alignment

Goal: Given two strings X = x1 x2 . . . xm and Y = y1 y2 . . . yn find alignment of minimum cost.

Def. An alignment M is a set of ordered pairs xi-yj such that each item occurs in at most one pair and no crossings.

Def. The pair xi-yj and xi'-yj' cross if i < i', but j > j'.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i:\, x_i \text{ unmatched}} \delta + \sum_{j:\, y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.

Sol: M = x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	–	G

–	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings x1 x2 . . . xi and y1 y2 . . . yj.

- Case 1: OPT matches xi-yj.
  - pay mismatch for xi-yj + min cost of aligning two strings
    x1 x2 . . . xi-1 and y1 y2 . . . yj-1
- Case 2a: OPT leaves xi unmatched.
  - pay gap for xi and min cost of aligning
    x1 x2 . . . xi-1 and y1 y2 . . . Yj
- Case 2b: Case 2b: OPT leaves yj unmatched.
  - pay gap for yj and min cost of aligning
    x1 x2 . . . xi and y1 y2 . . . yj-1

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

Sequence-Alignment. Fill up an m-by-n array.

```
Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) {
    for i = 0 to m
        M[i, 0] = iδ
    for j = 0 to n
        M[0, j] = jδ

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[xᵢ, yⱼ] + M[i-1, j-1],
                          δ + M[i-1, j],
                          δ + M[i, j-1])
    return M[m, n]
}
```

- Analysis. $\theta(mn)$ time and space.
- English words or sentences: $m, n \leq 10$.
- Computational biology: $m = n = 100{,}000$. 10 billions ops OK, but 10GB array?

Project

（**问题**）有m排n列的柱桩，每一排的柱桩从左向右标号为1,2,…,n，且在每个柱桩上预先放好价值不一样的宝石。现在有位杂技演员从第一排的第1号柱桩开始跳跃，每次都必须跳到下一排的柱桩上，且每次跳跃最多只能向左或向右移动一个桩子。也就是说如果现在杂技演员站在第j号桩上，那么他可跳到下一排的第j号桩上，也可跳到下一排的第j-1 (if j>1)或者 j+1 (if j<n)号桩上，并得到桩上的宝石。计算出一条最佳的跳跃顺序，使杂技演员获得的宝石的总价值最大。

（输入）

4 4   （4排4列的柱桩，空格隔开）

1,1, 1, 1   （放在第1排各桩上的宝石价值，逗号隔开）

1, 5, 1, 1   。

2 ,1, 10, 1   。

20 ,1, 1, 1   （放在第4排各桩上的宝石价值）

（输出）

28   （最大价值）

1   (开始位置，固定)

2   (在第二排的位置)

1   (在第三排的位置)

1   (在第四排的位置)

Project

要求：
a.单人独立完成；
b.提交名为 学号_姓名_SA.rar 的压缩文件，含如下内容：1). 完整的源码 2).不依赖于IDE环境的可执行文件及测试数据 3).电子版本项目报告，报告中至少包括对算法思想、递推方程式及该问题的最优子结构性质、程序结构的描述以及计算复杂度分析,以及测试结果
c. 第15周交（每班统一U盘拷贝)

说明：
1. 不依赖于IDE环境的可执行文件指exe及其支持dll，测试数据均在同一目录中，在任意一台Win XP机器上直接双击exe即可运行。
2. 测试数据不少于20排20列，按照前述的格式放在test.txt文件里，执行结果存入output.txt文件里

参考资料：
   Algorithm Design, Jon Kleiberg. Eva Tardos, Cornell University