CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues, BC 215, 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

Textbook: Cormen, Leiserson, Rivest, Stein.
Introduction to Algorithms (3nd Edition)

So far....

Finished looking at lower bounds and linear sorts.

Next: Memoization

-- Optimization problems - Dynamic programming

– A scheduling problem

– Matrix multiplication optimization

– Longest Common Subsequence

– Principles of dynamic programming

Divide and Conquer

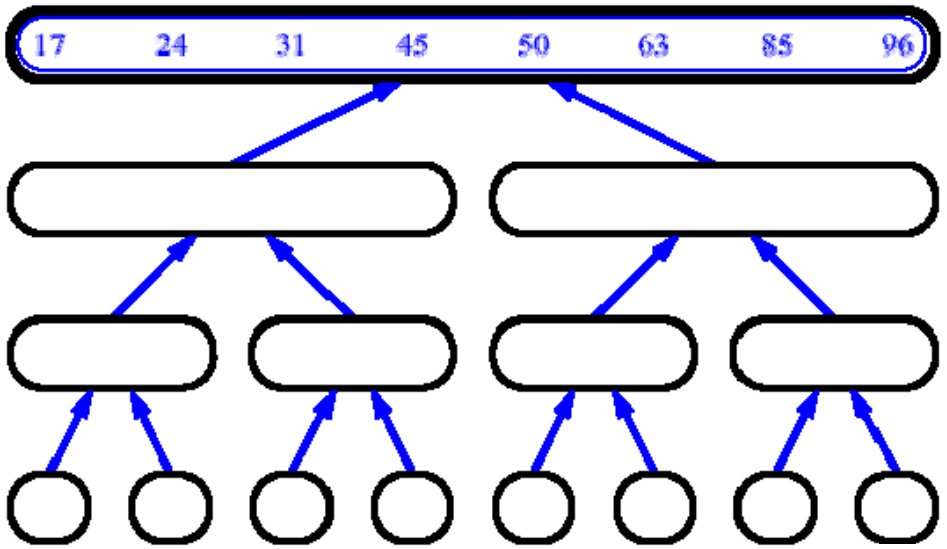
- Divide and conquer method:
  - Divide: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - Conquer: Use divide and conquer recursively to solve the subproblems
  - Combine: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem

Divide and Conquer(2)

- E.g., MergeSort

```
Merge-Sort(A, p, r)
   if p < r then
      q←(p+r)/2
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

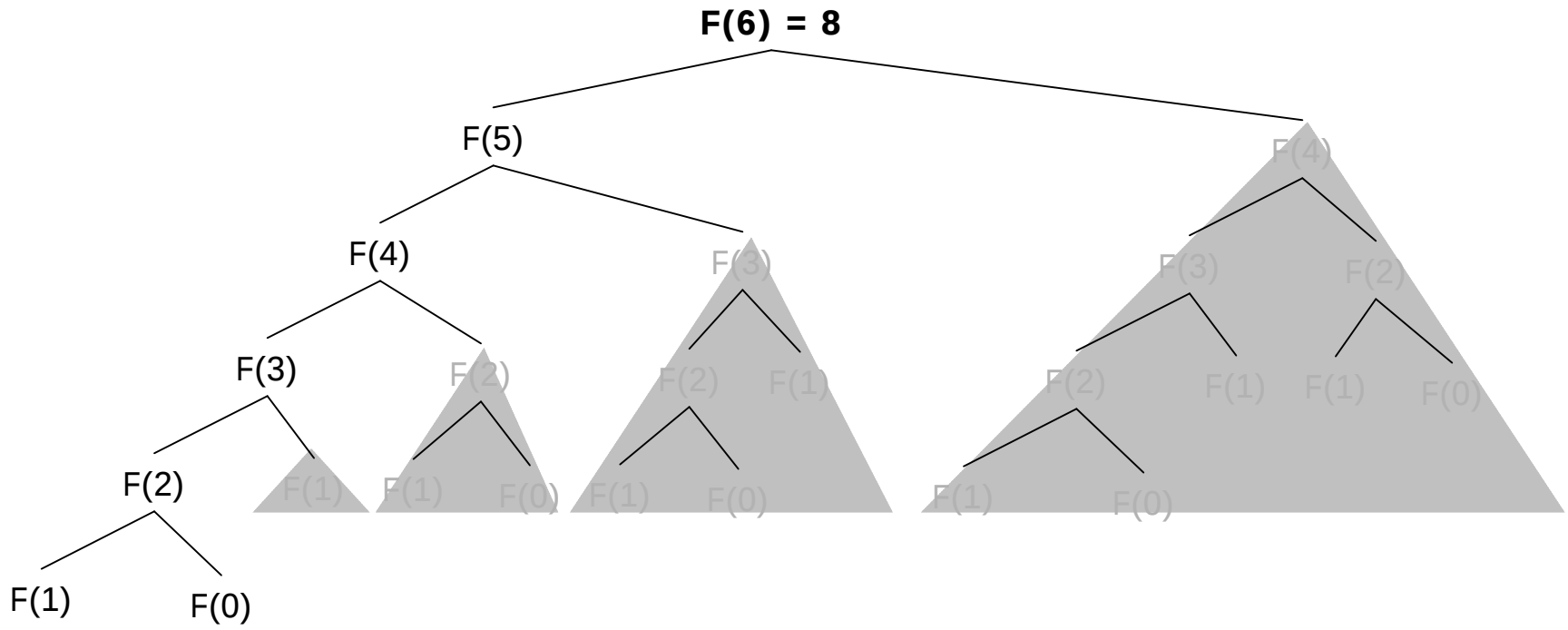
- The subproblems are independent.



Fibonacci Numbers

- $F_n = F_{n-1} + F_{n-2}$
- $F_0 = 0$, $F_1 = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …
- Straightforward recursive procedure is slow!
- Why? How slow?
- Let's draw the recursion tree

Fibonacci Numbers (2)



- We keep calculating the same value over and over!

Fibonacci Numbers (3)

- How many summations are there?
- Golden ratio $\frac{F_{n+1}}{F_n} \approx \phi = \frac{1+\sqrt{5}}{2} \approx 1.61803...$
- Thus $F_n \approx 1.6^n$
- Our recursion tree has only 0s and 1s as leaves, thus we have $\approx 1.6^n$ summations
- Running time is exponential!

Fibonacci Numbers (4)

- We can calculate F_n in linear time by remembering solutions to the solved subproblems – memoization
- Compute solution in a bottom-up fashion
- Trade space for time!
  - In this case, only two values need to be remembered at any time (less than the depth of recursion stack!)

```
Fibonacci(n)
    F0←0
    F1←1
    for i ← 1 to n do
        Fi ← Fi-1 + Fi-2
```

Lessons

We were able to reduce redundant computation by evaluating the recurrence in a certain order, and remembering previous values.

This is called memoization (no typo). This is used very often in dynamic programming.

Test your understanding

The following encoding is used to encode text:

a:1, b:2, …, y:25, z:26.

Unfortunately this is not a prefix code!

So parsing is ambiguous:

Given 1125: possible decodings are

aabe, aay, ale, kbe, ky

Problem: Given a string of digits, find the number of valid decodings.

Optimization Problems

- We have to choose one solution out of many – a one with the optimal (minimum or maximum) value.
- A solution exhibits a structure
  - It consists of a string of choices that were made – what choices have to be made to arrive at an optimal solution?
- The algorithms computes the optimal value plus, if needed, the optimal solution

A simple problem: optimizing an itinerary

We want to go from city 0 to city n using buses.
The road goes through cities 1,2,…, n-1.
The cost of going from city i to city j is c_{ij}.
Assume monotonic paths only (all edges go forward).
What is the minimum cost of going from 0 to n?

Exponential number of paths possible

(2 choices at each station – may or may not change buses there, n stations)

Important property: The optimal cost of going from (say) 2 to 9 has no relation with the same from 11 to 16.

Optimization Problems

- We have to choose one solution out of many – a one with the optimal (minimum or maximum) value.
- A solution exhibits a structure
  - It consists of a string of choices that were made – what choices have to be made to arrive at an optimal solution?
- The algorithms computes the optimal value plus, if needed, the optimal solution

A simple problem: optimizing an itinerary

We want to go from city 0 to city n using buses.
The road goes through cities 1,2,…, n-1.
The cost of going from city i to city j is c_{ij}.
Assume monotonic paths only (all edges go forward).
What is the minimum cost of going from 0 to n?

Exponential number of paths possible

(2 choices at each station – may or may not change buses there, n stations)

Important property: The optimal cost of going from (say) 2 to 9 has no relation with the same from 11 to 16.

Optimizing an itinerary

We want to make local choices and remember them systematically. Let T(j) be the minimum cost of going from city 0 to city j. So T(n) is the answer.

What can we say about T(j)?

Suppose someone tells you the best last choice (go from i to n). Does it help?

Suppose you also know the best way to go from 0 to i.

Then we can **glue** the solutions together and get the optimal solution!

Maybe we should not expect so much ☺

Optimizing an itinerary

Note that T(i) is a smaller subproblem than T(n).

When did T(i) go from a cost to a subproblem?

Perhaps we can solve T(i) recursively?

Then we know $T(n) = c_{in} + T(i)$

Paraphrased dialogue from *Die Hard III*:

(Samuel Jackson): And who will help you?

Great life lesson?

(SJ's kids, in chorus): no one!

How can you prevent assuming that you know the best last choice?:

Take the minimum over all last choice possibilities!

Optimizing an itinerary: putting it all together

$T(j) = \min_k [c_{kj} + T(k)]$, $k<j$.

Why does this help at all?

Can systematically compute T(j);

Hopefully results in a polynomial-time algorithm

Q: What's the right way to compute T(j)?

A: What's easy? Well T(1), since it equals c_{01}

Start from T(1). Then do T(2). Keep going until you reach T(n). Each entry uses the recursion above.

Optimizing an itinerary: getting solutions

T(n) = minimum cost of going from 0 to n.
What is the sequence of steps?

Need to remember more information;
Specifically the sequence of choices made.

$T(j) = \min_k [c_{kj} + T(k)]$, $k<j$.
C(j) = arg min k

What's the last choice? C(n).

What's the next one? C(C(n)) !

The next one is C(C(C(n))). The next one is C(C(C(C(n)))). Keep going until you hit 0.

Optimizing an itinerary: running time

T(n) = minimum cost of going from 0 to n.
What is the time required to compute T(n), assuming T(1) through T(n-1) are known?

Computing T(j) takes $\Theta(j)$ time.
Computing C(j) takes O(1) time.
So the algorithm takes $\Theta(n^2)$ time.

Next: an activity selection problem

- Two assembly lines, A_i, B_i, each with n stations.
- Each job must complete go through A_i or B_i for each i.
- Different costs for going from A_i to B_{i+1}, A_i to A_{i+1}, B_i to B_{i+1} , B_i to A_{i+1}, start to A_1, start to B_1, A_n to exit, B_n to exit.

Exponential number of paths possible (2 choices, n stations)

Again, suppose you know the first choice. Does that help?

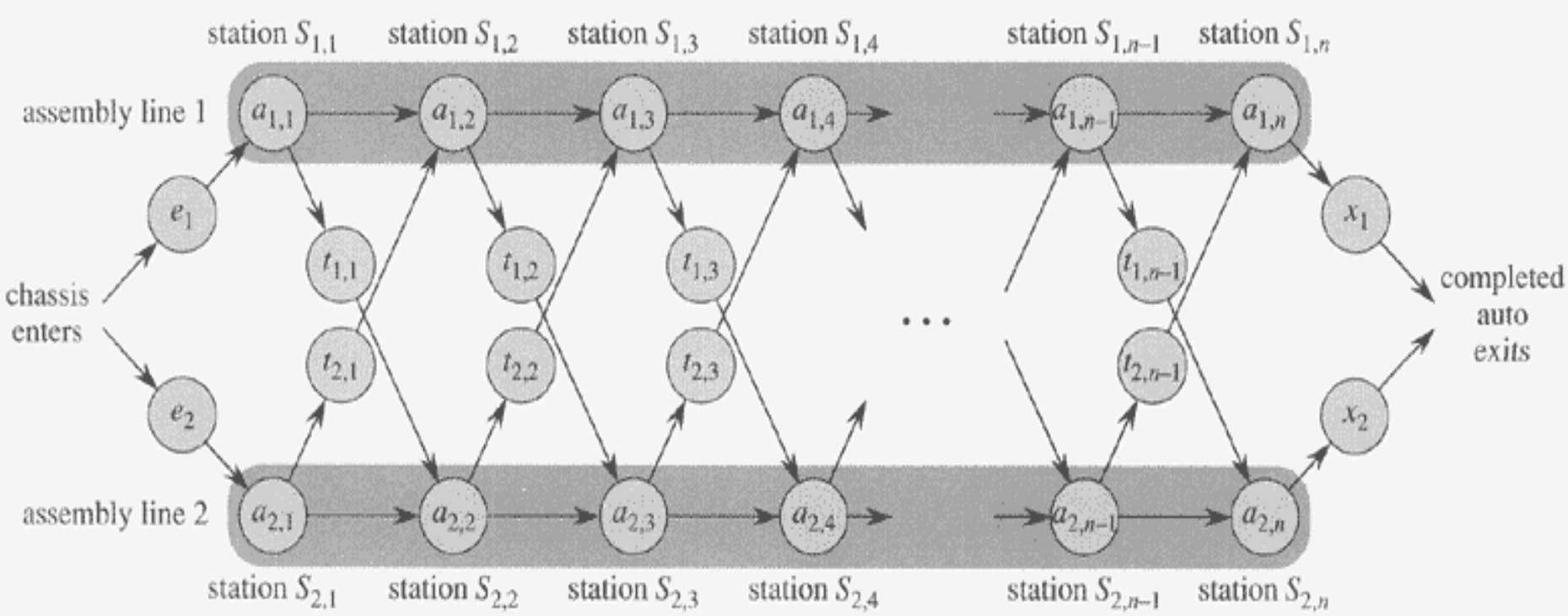
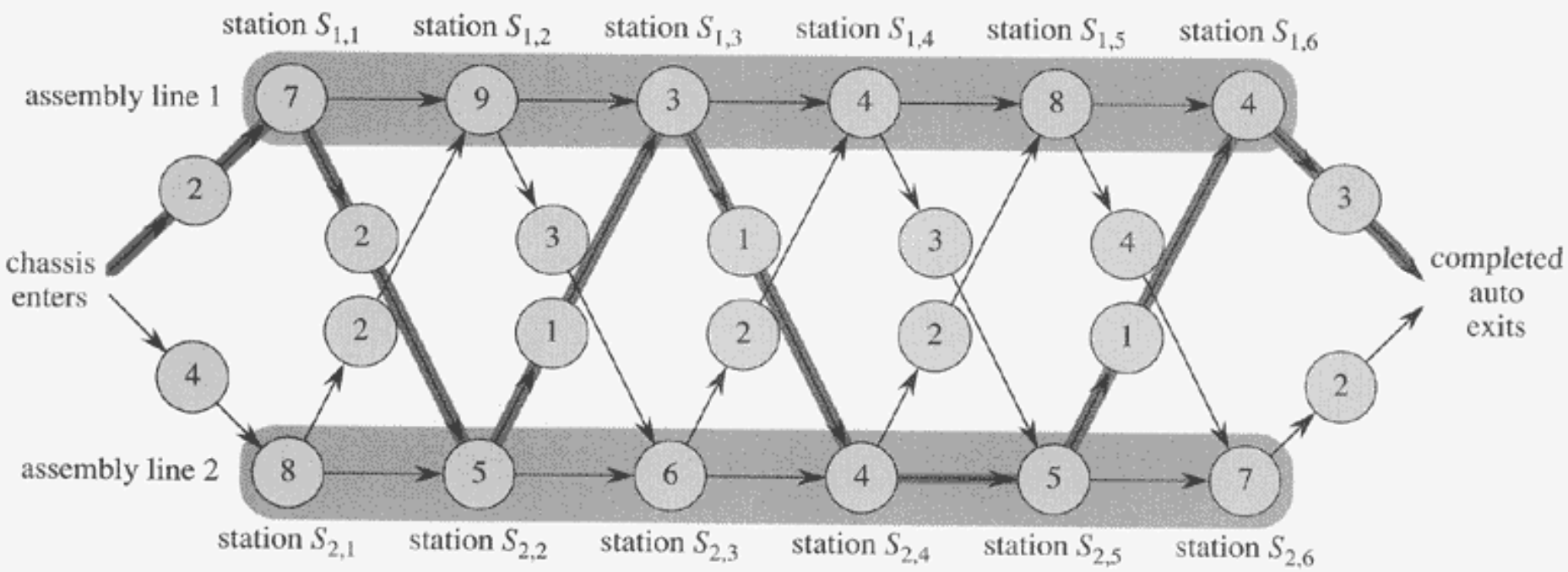


Figure 15.1 A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with n stations; the jth station on line i is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$. An automobile chassis enters the factory, and goes onto line i (where $i = 1$ or 2), taking e_i time. After going through the jth station on a line, the chassis goes on to the $(j+1)$st station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$. After exiting the nth station on a line, it takes x_i time for the completed auto to exit the factory. The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

Again think recursively

Express the cost of the remainder recursively.

Now assume you do not know the first choice!

Define $f_1[j]$ to be the cost of going to the j^{th} station on assembly line 1 from the start. Define $f_2[j]$ similarly for assembly line 2. Then

$$f_1[j] = e_1 + a_{1,1} \quad \text{if } j = 1$$

$$= \min \{f_1[j-1]+a_{1,j},\ f_2[j-1]+t_{2,j-1}+a_{1,j}\} \quad \text{if } j > 1$$

Similarly, for $f_2[j]$.

Finally

$$f^* = \min \{f_1[n]+x_1,\ f_2[n]+x_2\}$$

Constructing the solution

As before, we need some extra storage and record keeping to remember the choices made.

```
PRINT-STATIONS(l, n)
1   i ← l*
2   print "line " i ", station " n
3   for j ← n downto 2
4       do i ← l_i[j]
5          print "line " i ", station " j − 1
```

Running time

$f_1[j] = e_1 + a_{1,1}$ if $j = 1$

$= \min \{f_1[j-1]+a_{1,j}, f_2[j-1]+t_{2,j-1}+a_{1,j}\}$ if $j > 1$

Similarly, for $f_2[j]$. Finally

$f^* = \min \{f_1[n]+x_1, f_2[n]+x_2\}$

How much time does it take to compute f^*?

Constant amount of work to compute $f_1[j]$, $f_2[j]$, for each j, and for f^*.

Total running time $\Theta(n)$.

Multiplying Matrices

- Two matrices, A – n x m matrix and B – m x k matrix, can be multiplied to get C with dimensions n x k, using nmk scalar multiplications

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \dots & \dots & \dots \\ \dots & c_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \qquad c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$$

- Problem: Compute a product of many matrices efficiently
- Matrix multiplication is associative:

  (AB)C = A(BC)

Multiplying Matrices (2)

- The parenthesization matters
- Consider A × B × C × D, where
  - A is 30 × 1, B is 1 × 40, C is 40 × 10, D is 10 × 25
- Costs:
  - (AB)C)D = 1200 + 12000 + 7500 = 20700
  - (AB)(CD) = 1200 + 10000 + 30000 = 41200
  - A((BC)D) = 400 + 250 + 750 = 1400
- We need to optimally parenthesize

$$A_1 \times A_2 \times \ldots \times A_n, \text{ where } A_i \text{ is a } d_{i-1} \times d_i \text{ matrix}$$

Multiplying Matrices (3)

- Let M(i,j) be the minimum number of multiplications necessary to compute $\prod_{k=i}^{j} A_k$
- Key observations
  - The outermost parenthesis partition the chain of matrices (i,j) at some k, ($i \le k < j$): $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$
  - The optimal parenthesization of matrices (i,j) has optimal parenthesizations on either side of k: for matrices (i,k) and (k+1,j)

Multiplying Matrices (4)

- We try out all possible k. Recurrence:

$$M(i,i) = 0$$

$$M(i,\, j) = \min_{i \le k < j} \left\{ M(i,k) + M(k+1,\, j) + d_{i-1} d_k d_j \right\}$$

- A direct recursive implementation is exponential – there is a lot of duplicated work (why?)

- But there are only $\binom{n}{2} + n = \Theta(n^2)$ different subproblems (i,j), where $1 \le i \le j \le n$

Multiplying Matrices (5)

Thus, it requires only $\Theta(n^2)$ space to store the optimal cost M(i,j) for each of the subproblems: half of a 2-d array M[1..n,1..n].

```
Matrix-Chain-Order(d0…dn)
1  for i←1 to n do
2     M[i,i] ← 0
3  for l←2 to n do
4     for i←1 to n-l+1 do
5        j ← i+l-1
6        M[i,j] ← ∞
7        for k←i to j-l do
8           q ← M[i,k]+M[k+1,j]+di-1dkdj
9           if q < M[i,j] then
10             M[i,j] ← q
11             s[i,j] ← k
12 return M, s
```

Multiplying Matrices

- After execution: M[1,n] contains the value of the optimal solution and c contains optimal subdivisions (choices of k) of any subproblem into two subsubproblems.
- A simple recursive algorithm Print-Optimal-Parents(c, i, j) can be used to reconstruct an optimal parenthesization.
- Exercise: Hand run the algorithm on

  d = [10, 20, 3, 5, 30]

Multiplying Matrices

- Running time
  - we are filling up a table with n^2 entries; each take $\Omega(n)$ work.
  - So, the running time is $\Omega(n^3)$.
- From exponential time to polynomial.

Memoization

- If we still like recursion very much, we can structure our algorithm as a recursive algorithm:
  - Initialize all M[i,j] to ∞ and call Lookup-Chain(d, i, j)

```
Lookup-Chain(d,i,j)
1  if M[i,j] < ∞ then
2     return m[i,j]
3  if i=j then
4     m[i,j] ←0
5   else for k ← i to j-1 do
6           q ← Lookup-Chain(d,i,k)
7               + Lookup-Chain(d,k+1,j)+d_{i-1}d_k d_j
7           if q < M[i,j] then
8              M[i,j] ← q
9   return M[i,j]
```

Applicability

Can we always apply dynamic programming?

No, certain conditions must hold.

Dynamic Programming

To apply dynamic programming, we have to:

1. Show optimal substructure – an optimal solution to the problem contains within it optimal solutions to sub-problems
    - Solution to a problem:
        - Making a choice out of a number of possibilities (look what possible choices there can be)
        - Solving one or more sub-problems that are the result of a choice (characterize the space of sub-problems)
    - Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal (use "cut-and-paste" argument)

Dynamic Programming (2)

2. Write a recurrence for the value of an optimal solution
   - $M_{opt} = \min_{\text{over all choices } k}$ {(Sum of M_{opt} of all sub-problems, resulting from choice *k*) + (the cost associated with making the choice *k*)}
   - Show that the number of different instances of sub-problems is bounded by a polynomial

Dynamic Programming (3)

3. Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary sub-results pre-computed (or use memoization)

– See if it is possible to reduce the space requirements, by "forgetting" solutions to sub-problems that will not be used any more

4. Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution).

Longest Common Subsequence

- Two text strings are given: X and Y
- There is a need to quantify how similar they are:
  - Comparing DNA sequences in studies of evolution of different species
  - Spell checkers
- One of the measures of similarity is the length of a Longest Common Subsequence (LCS)

LCS: Definition

- Z is a subsequence of X, if it is possible to generate Z by skipping some (possibly none) characters from X
- For example: X =“ACGGTTA”, Y=“CGTAT”, LCS(X,Y) = “CGTA” or “CGTT”
- To solve LCS problem we have to find “skips” that generate LCS(X,Y) from X, and “skips” that generate LCS(X,Y) from Y

LCS: Optimal substructure

- We make Z to be empty and proceed from the ends of X_m="$x_1\, x_2 \ldots x_m$" and Y_n="$y_1\, y_2 \ldots y_n$"
  - If $x_m = y_n$, append this symbol to the beginning of Z, and find optimally LCS(X_{m-1}, Y_{n-1})
  - If $x_m \neq y_n$,
    - Skip either a letter from X
    - or a letter from Y
    - Decide which decision to do by comparing LCS(X_m, Y_{n-1}) and LCS(X_{m-1}, Y_n)
  - "Cut-and-paste" argument

LCS: Recurrence

- The algorithm could be easily extended by allowing more "editing" operations in addition to copying and skipping (e.g., changing a letter)
- Let c[i,j] = LCS(X_i, Y_j)

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

- Observe: conditions in the problem restrict sub-problems (What is the total number of sub-problems?)

LCS: Compute the optimum

```
LCS-Length(X, Y, m, n)
1  for i←1 to m do
2     c[i,0] ← 0
3  for j←0 to n do
4     c[0,j] ← 0
5  for i←1 to m do
6     for j←1 to n do
7        if x_i = y_j then
8           c[i,j] ← c[i-1,j-1]+1
9           b[i,j] ← "copy"
10       else if c[i-1,j] ≥ c[i,j-1] then
11                c[i,j] ← c[i-1,j]
12                b[i,j] ← "skip x"
13             else
14                c[i,j] ← c[i,j-1]
15                b[i,j] ← "skip y"
16 return c, b
```

LCS: Example

- Lets run: X =“ACGGTTA”, Y=“CGTAT”
- How much can we reduce our space requirements, if we do not need to reconstruct LCS?

0 / 1 Knapsack: the greedy algorithm fails

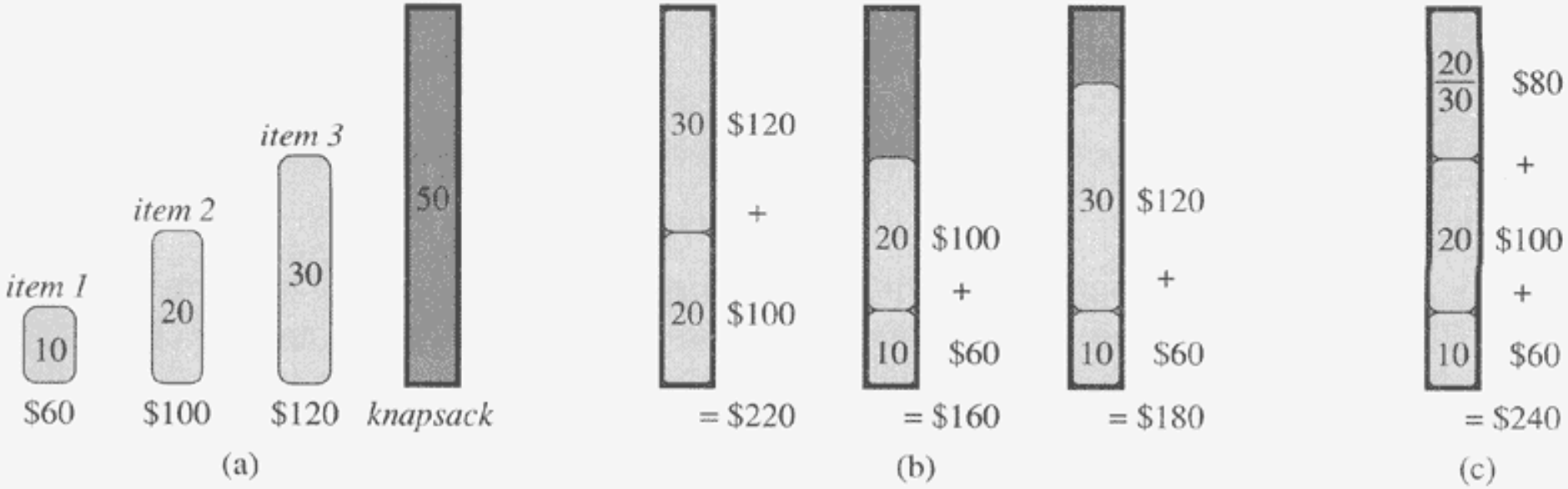


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

0 / 1 Knapsack

Optimal substructure:

Recurrence

$c[i,w] = 0$ if $i=0$ or $w=0$

$= c[i-1,w]$ if $w_i > w$

$= \max\{v_i + c[i-1,w-w_i], c[i-1,w]\}$ if $i > 0$, $w \geq w_i$

0 / 1 Knapsack – Dynamic Programming solution

```
DP-0/1-Knapsack(v,w,n,W)
for w =0 to W { c[0,w] = 0 }
for i=1 to n {
    c[i,0]=0
    for w=1 to W {
        if(w_i ≤ w) {
                if(v_i+c[i-1,w-w_i]>c[i-1,w])
                        c[i,w] = v_i + c[i-1,w-w_i]
                else c[i,w] = c[i-1,w]
         }//end if
     else c[i,w] = c[i-1,w]
} //end for w } //end for i
```