Part III

Advanced Design and Analysis Techniques

§ Two Types of Techniques:

- Dynamic Programming
- Greedy Algorithms

- **Dynamic Programming**
  - subproblems *overlap* (→ repeated subproblems)
  - problem solved by *combining solutions* to its subproblems
  - *saving solutions* to common subproblems *in table(s)* for efficiency
  - commonly applicable to optimization problems

- **Greedy Algorithms**
  - following locally optimal choice (instead of exhaustive search)
  - simpler and more efficient approach
  - work for wide ranges of problems
  - yield optimal solutions to certain problems, e.g., combinatorial structures *matroids*

(where a matroid possesses linear independence.)

Dynamic Programming (continued)

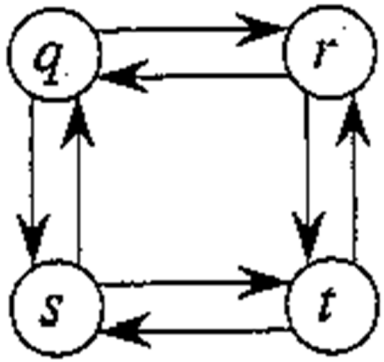
- **Four Steps Involved**
  - characterize structure of optimal solution **(check applicability,** i.e., if optimal substructures exist**)**
  - recursively define expression of optimal solution (**establish recursive formulation**)
  - compute value of optimal solution (**keep subproblem solutions in bottom-up way**)
  - construct optimal solution from computed information

- **Example for Optimal Rod Cutting**
  - solve each subproblem *once*
  - *time-memory tradeoff*: additional memory involved for time complexity reduction
  - full solution in polynomial time if the number of *distinct subproblems* is polynomial
  - two approaches for dynamic programming implementation:
    - top-down with memoization †
    - bottom-up method, with *smallest* subproblems solved first and kept in table(s)

† Here, memoization is from “memo” and *not* from “memory.”

Dynamic Programming (continued)

- **First of four Steps**
  – characterize the structure of optimal solution (**whether applicable**)



Longest simple path from *q* to *t:*
q* → *r* → *t
whereas longest simple path from *q* to *r:*
q* → *s* → *t* → *r
and longest simple path from *r* to *t:*
r* → *q* → *s* → *t

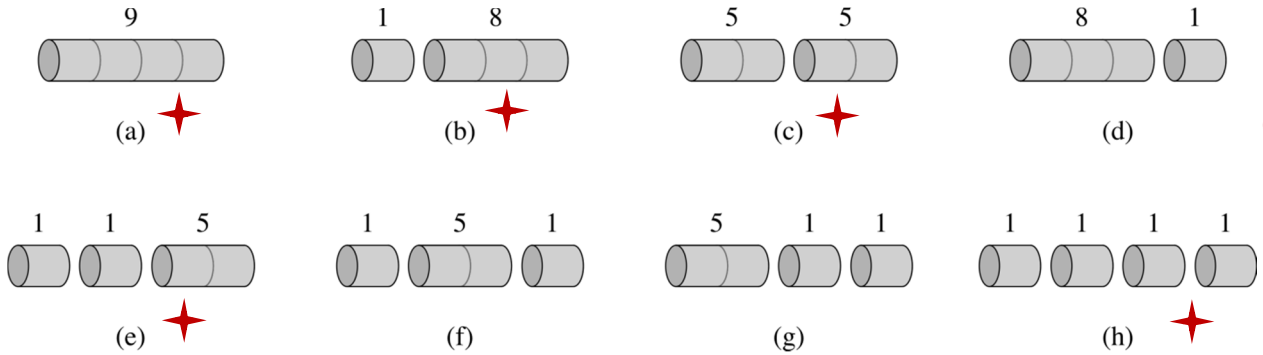
Figure 15.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

Dynamic programming is applicable *if and only if* optimal solution to it can be composed from optimal solutions to its subproblems. Note while problems solvable by divide-&-conquer are applicable, other problems may also be applicable, as long as their solutions can be obtained from the solutions to their subproblems.

Dyanmic Programming (continued)

- **Rod cutting problem**
  - + rod with length n has 2^{n-1} ways (which equal leaf nodes of recursion tree) to cut
  - + number of *distinct subproblems* far smaller, due to many identical subproblems
  - + this problem exhibits *optimal substructure*, i.e., optimal solution consists of optimal solutions to related subproblems, which can be solved independently

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



only five (5) distinct subproblems exist

Recursive top-down implementation

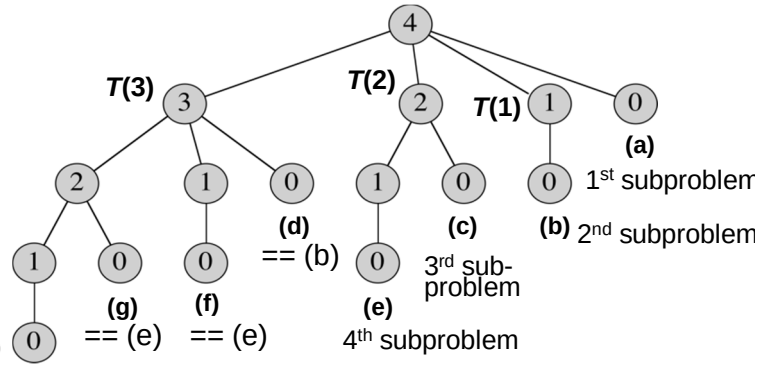
```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

root

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

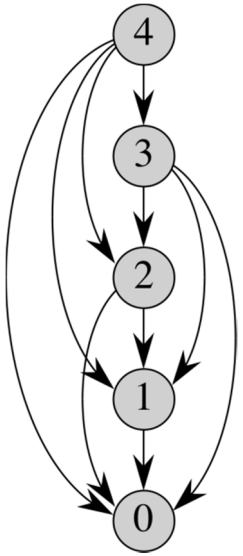
$= 2^n$ **(covers the whole tree, not just leaf nodes)**

Recursion tree showing recursive calls



Dynamic Programming (continued)

- **First three steps of typical dynamic programming**
  - characterize structure of optimal solution (**applicability**)
  - recursively define value of optimal solutions to subproblems (**recursion formulation**)
  - compute the value of an optimal solution to every subproblem (**keep subsolutions**)



Subproblem graph, being collapsed recursion tree

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

Running time complexity: $\Theta(n^2)$
(as it equals $n-1$, n-2, …,1 for i =1, 2 …, obtainable from recursion tree in preceding page, to visit leftmost path for calculation + the level just below the root for references)

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      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
```

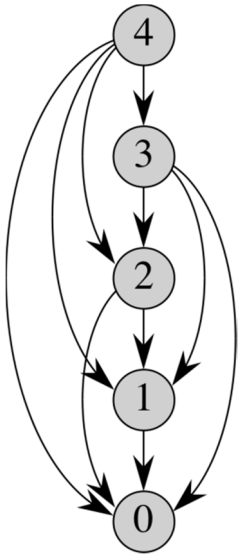
reference to subproblem solved before

top-down, same as divide-&-conquer method, but with memoization

Dynamic Programming (continued)

- Bottom-up method
  - same asymptotic complexity as the top-down method, with running time complexity of $\Theta(n^2)$
  - often with smaller constants, due to avoiding recursive calls (employed by top-down counterpart)

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



Subproblem graph, being collapsed recursion tree

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

```
let r[0..n] be 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]
```

bottom-up, permitting table lookups

Dynamic Programming (continued)

- **Fourth step of typical dynamic programming**
  - reconstruct a solution to the problem
  - the solution of rod-cutting problem includes the choice of every cut (the cut listed in array ***s*** below), besides the optimal value (given in array ***r*** below)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i        // keep the cut location
    r[j] = q
return r and s
```

Solution of the problem printed by:

```
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]
```

‡ PRINT-CUT-ROD-SOLUTION(*p*, 3) prints “3” while PRINT-CUT-ROD-SOLUTION(*p*, 8) prints “2” & “6”

EXTENDED-BOTTOM-UP-CUT-ROD(*p*, 8) returns ***r*** and ***s***, as follows:

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

Dynamic Programming (continued)

- **Elements of dynamic programming**
  - two key elements for optimization problems to apply dynamic programming:
    + optimal substructure
    + overlapping subproblems
  - e.g., optimal parenthesization of matrix chain: $A_i \bullet A_{i+1} \bullet \ldots \bullet A_j$ consists of optimal solutions to split parenthesizing subproblems $A_i \bullet A_{i+1} \bullet \ldots \bullet A_k$ & $A_{k+1} \bullet A_{k+2} \bullet \ldots \bullet A_j$

Matrix-chain multiplication problem

Given a chain of n matrices $<A_1, A_2, \ldots, A_n>$, find full parenthesization of the product $A_1 \bullet A_2 \bullet \ldots \bullet A_n$ that minimizes the number of scalar multiplications.

Let $<A_1, A_2, A_3>$ are of dimensions 10x100, 100x5, and 5x50.
The number of multiplications equals: 10•100•5 + 10•5•50 = 7500 for $((A_1 \bullet A_2) \bullet A_3)$

and equals: 100•5•50 + 10•100•50 = 75000 for $(A_1 \bullet (A_2 \bullet A_3))$

Dynamic Programming (continued)

- **Matrix-chain multiplication problem**
  - given <A_1, A_2, A_3, A_4>, we have five distinct ways for full parenthesization:
    (A_1•(A_2•(A_3•A_4)))
    (A_1•((A_2•A_3)•A_4))
    ((A_1•A_2)•(A_3•A_4))
    ((A_1•(A_2•A_3))•A_4)
    (((A_1•A_2)•A_3)•A_4)
  - let $P(n)$ denote number of *alternative parenthesizations* of a sequence of n matrices and the two split subproducts be: A_1•A_2• … •A_K and A_{k+1}•A_{k+2}• … •A_n, we have the recurrence below:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n \geq 2 . \end{cases}$$

    whose time complexity is $\Omega(2^n)$, exponential time complexity
    (see pp. 386 for a proof by substitution).

Dynamic Programming (continued)

- **Applying dynamic programming to matrix-chain multiplication**

  + four-step sequence:
    1. characterize structure of optimal parenthesization **(applicability)**
    2. establish recursive solution approach **(recursive formulation)**
    3. compute optimal costs **(memoization or table storage)**
    4. construct an optimal solution

Recursive solution approach

– let $m[i, j]$ be the minimum number of scalar multiplications for matrix sequence $A_{i..j}$ with two split subsequences $A_i \bullet A_{i+1} \bullet \ldots \bullet A_k$ & $A_{k+1} \bullet A_{k+2} \bullet \ldots \bullet A_j$. We have:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \quad \text{for a given } k$$

– consider every possible k, $i \le k \le j$-1, we have

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

– finally, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization.

Dynamic Programming (continued)

- Recursive solution approach (without avoiding repeated subproblem computation)
  - from recurrence below directly leads to ***exponential complexity***:

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

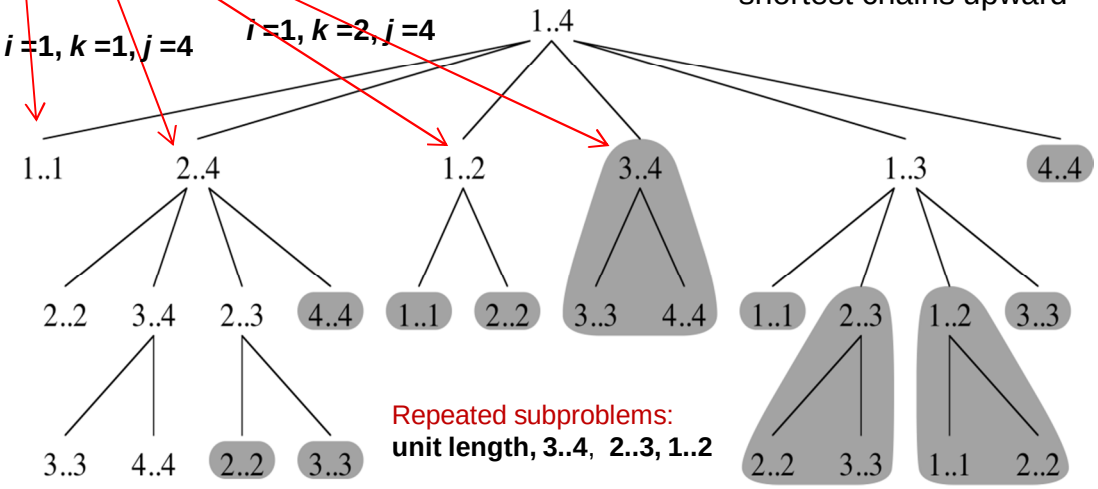
Time complexity of Line 5 (for $i = 1$ and $j = n$):

$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$

$\geq 2^{n-1}$ (proof details given in pp. 386)

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

] // recursive calls, following divide-&-conquer alone
// **without** properly memoizing results in *m*[*i*, *j*] from shortest chains upward



Dynamic Programming (continued)

- **Applying dynamic programming to matrix-chain multiplication**

Computing optimal costs

– relatively *few distinct* subproblems (i.e., subproblems are overlapping)

– two auxiliary tables: $m[1..n, 1..n]$ to store the $m[i, j]$ cost and $s[1..n\text{-}1, 2..n]$ to store k, $i \le k \le j\text{-}1$, at which a split yields the lowest cost in computing $m[i, j]$

MATRIX-CHAIN-ORDER(p)

There are $\Theta(n^2)$ *distinct* subproblems, to yield time complexity of $\Theta(n^3)$.

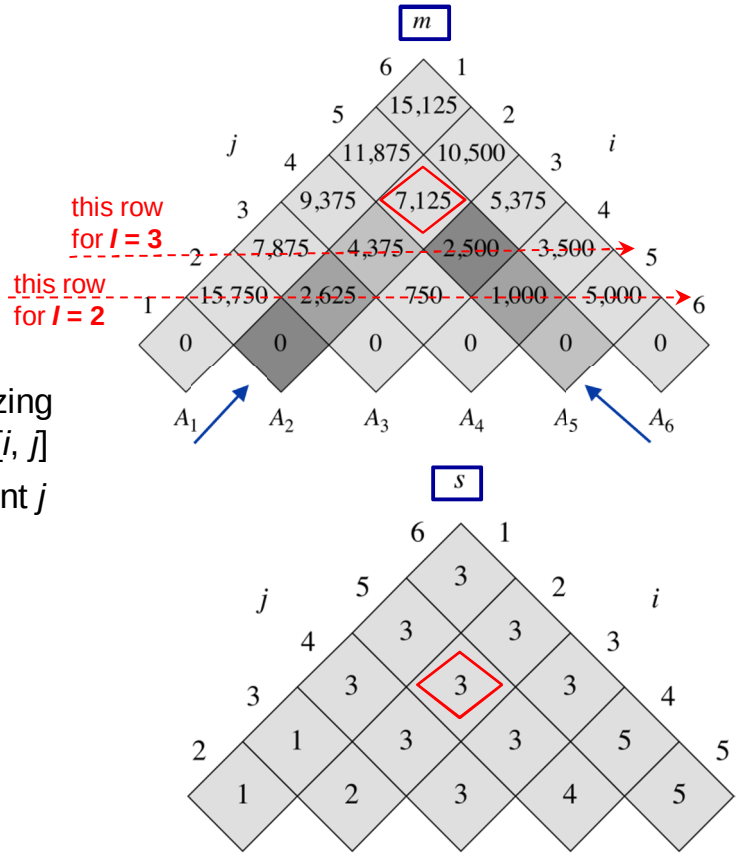
```
1  n = p.length − 1
2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3  for i = 1 to n
4      m[i, i] = 0                    // nil cost for unit length stored in m[i, i]
5  for l = 2 to n                     // l is the chain length, for memoizing results in m[i, j] from shortest chains upward
6      for i = 1 to n − l + 1
7          j = i + l − 1              // range from element i to element j for the chain length of l
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        // memoization
13                 s[i, j] = k
14 return m and s
```

Dynamic Programming (continued)

Computing optimal costs

```
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 for memoizing
 6      for i = 1 to n − l + 1                results in m[i, j]
 7          j = i + l − 1             // range from element i to element j
 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
```

Time complexity = $\Theta(n^3)$



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

$m[1, 6]$ = 15,125

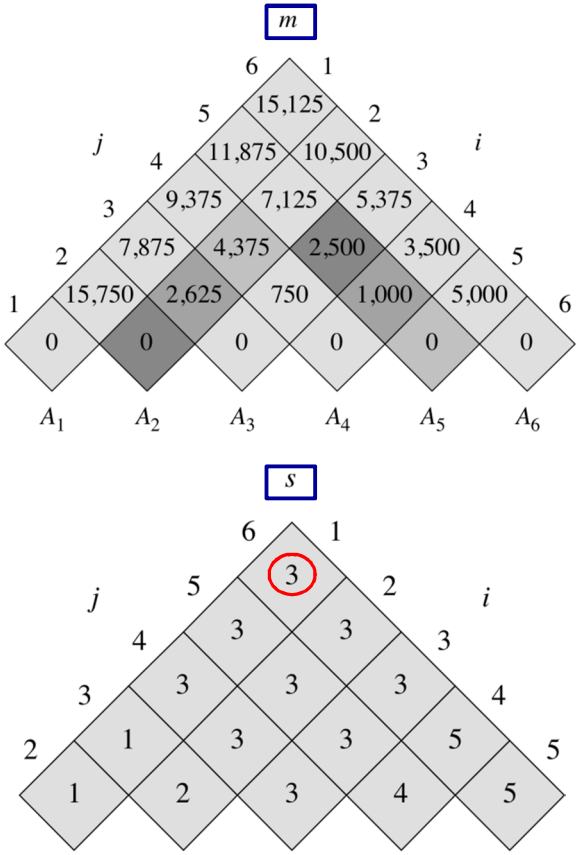
its computation relies on
$m[1, 1]$, $m[2, 6]$, $m[1, 2]$, $m[3, 6]$,
$m[1, 3]$, $m[4, 6]$, $m[1, 4]$, $m[5, 6]$
$m[1, 5]$, $m[6, 6]$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

Dynamic Programming (continued)

Constructing an optimal solution

– Entry $s(i, j)$ keeps k where optimal parenthesization splits between A_k and A_{k+1}




```
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 ")"
```

PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

PRINT-OPTIMAL-PARENS(s, 1, s[1, 6] = 3)

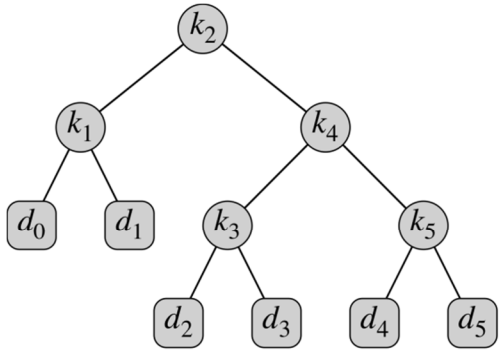
PRINT-OPTIMAL-PARENS(s, s[1, 6]+1=4, 6)

Dynamic Programming (continued)

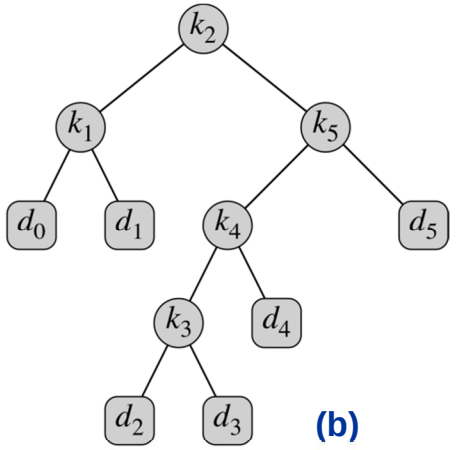
- **Optimal binary search trees (O-BSTs)**
  - given a set of sorted keys $K = <k_1, k_2, \ldots k_n>$, with p_i being search visit probability of k_i, build a binary search tree with the *mean search cost* minimized
  - n+1 "dummy keys" (i.e., d_j) required to cover those value ranges outside key set K

contribution = $(\text{depth}_T(k_i)+1) \cdot p_i$

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



(a)



(b)

Figure 15.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search cost 2.75. This tree is optimal.

Dynamic Programming (continued)

- **Optimal binary search trees (O-BSTs)**
  - number of binary trees with n nodes: $\Omega(n^4/n^{3/2})$ // see Prob. 12-4, pp. 306
  - optimal solution to the problem obtainable from optimal solutions to its subproblems:
    root k_r has left subtree, k_i, k_{i+1}, … k_{r-1}, and right subtree, k_{r+1}, k_{r+2}, …, k_j
  - recursive solution: let $e[i, j]$ denote mean cost of searching O-BST with k_i, k_{i+1}, …, k_j

    then, $e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$, with k_r the tree root,

    due to tree height raised by 1

    where $w(i, j)$ is *expected search prob.* over the tree, equal to $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$

    We have final $e[i, j]$ as follows, due to $w(i, j) = w(i, r-1) + p_r + w(r+1, j)$:

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

due to tree height raised by 1

There are relatively fewer distinct subproblems (i.e., O-BST involving k_i, k_{i+1}, … k_j):

$C(n, 2) + n = \Theta(n^2)$

selecting *i* & *j* out of *n values*

due to *i* = *j*

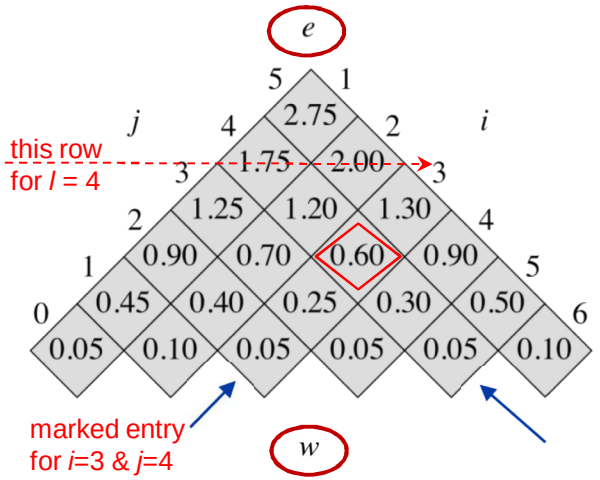
Dynamic Programming (continued)

- **Optimal binary search trees (O-BSTs)**
  - compute the expected search cost of an O-BST
  - we store the $e[i, j]$ values in a table $e[1..n+1, 0..n]$.
  - keep optimal BST for key subset of k_i, k_{i+1}, …, k_j in *root*

```
OPTIMAL-BST(p, q, n)
 1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
          and root[1..n, 1..n] be new tables
 2  for i = 1 to n + 1
 3      e[i, i − 1] = q_{i−1}
 4      w[i, i − 1] = q_{i−1}
 5  for l = 1 to n
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j − 1] + p_j + q_j
10          for r = i to j        // search subset of k_i .. k_j with l consecutive keys
11              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Note. p_j & q_j



Greedy Algorithm

- **Basics**
  - make locally optimal choice, so only a single subproblem is solved in each step
  - simpler and more efficient
  - powerful and applicable to wide ranges of problems

An Example

Activity selection problem. To find a maximal subset of *compatible activities* from a set of activities, each with a starting time and an ending time.

Given the following set S of activities, **sorted according to their finish times**

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Compatible activities, a_k and a_l, satisfy $f_k \le s_l$ or $s_k \ge f_l$.

Largest compatible sets: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$.

This cannot be identified via a greedy algorithm.

Greedy Algorithm (continued)

Given a set of activities, S, sorted according to their finish times, i.e.,

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n$$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Let S_{ij} denote the set of activities that start after a_i finishes and that finish before a_j starts.
Find out a maximum set of compatible activities in S_{ij}, with such a set represented by A_{ij}.

If A_{ij} contains a_k, then the solution is left with two subproblems: maximum set of compatible activities in S_{ik} and the maximum set of compatible activities in S_{kj} , namely, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.

Let the size of an optimal solution for S_{ij} be denoted by $c[i, j]$, with a_k in the optimal solution set.
We have the recurrence of

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Consider every possible a_k, we have:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

Note: S_i denotes the subset of activities $\{a_{i+1}, a_{i+2}, \ldots a_n\}$ that start after a_i finishes.

One may adopt dynamic programming to solve the above recurrence (either top-down with memoization or bottom-up with auxiliary table(s)) , or preferably to solve *just one* desirable subproblem (instead of *all*) in each step via a greedy choice, called the greedy approach.

Greedy Algorithm (continued)

- **Greedy Choice**
  - activity selection problem chooses the activity which ***finishes the earliest***
  - typically, one follows top-down manner to choose an activity for adding to optimal solution, then finds maximal compatible activities for left subproblem
  - the greedily chosen activity always part of some optimal solution, as follows:

Theorem 1.

If S_k is nonempty and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

Unlike S_{kv}, S_k denotes the subset of activities $\{a_{k+1}, a_{k+2}, \ldots a_n\}$ that start after a_k finishes.

Recursive greedy algorithm

REC-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$
while $m \le n$ and $s[m] < f[k]$ // find the first activity in S_k (compatible with a_k)
    $m = m + 1$
if $m \le n$
    return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

Greedy Algorithm (continued)

- **Recursive greedy algorithm**

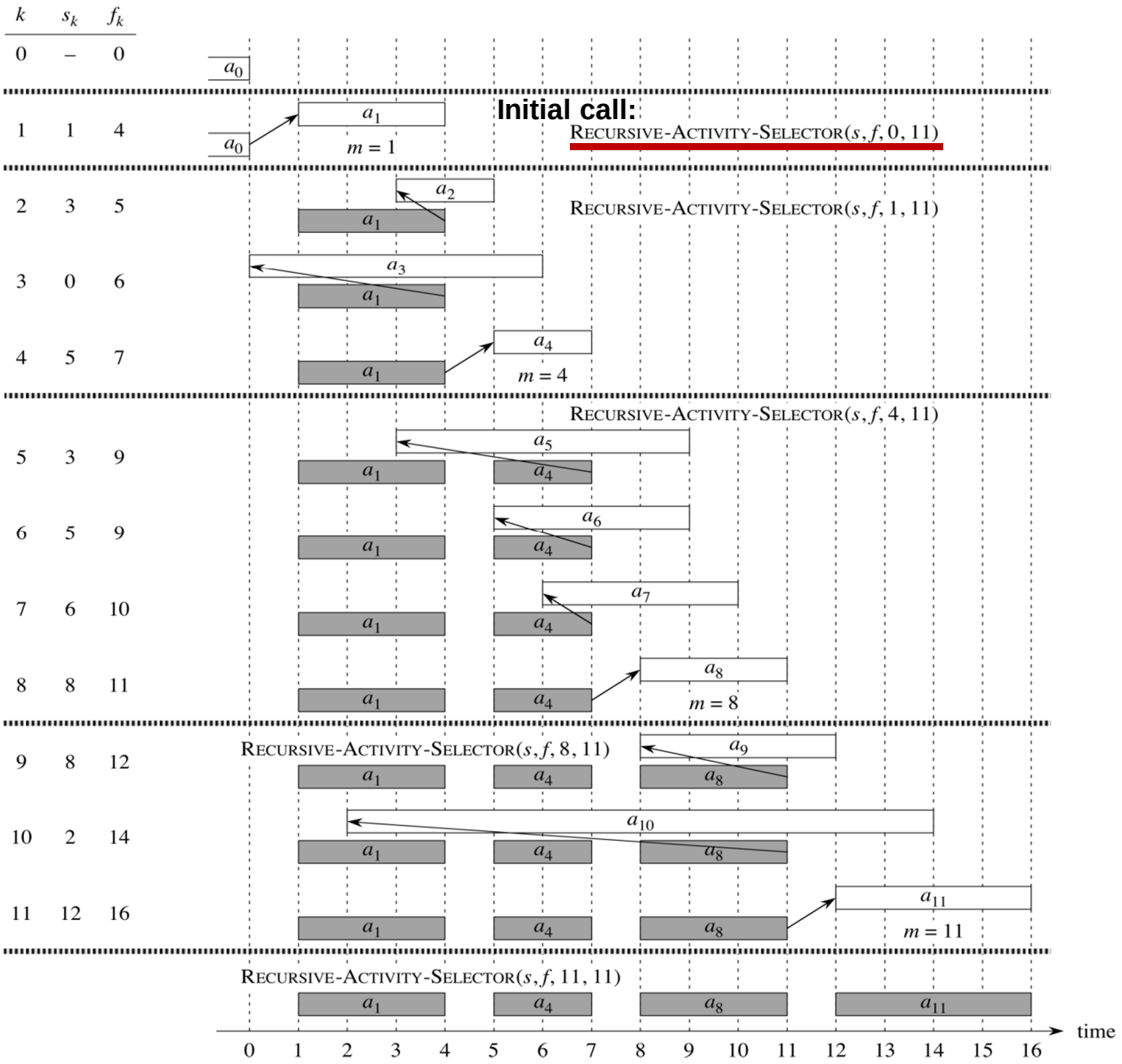
Idea

The **while** loop checks $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_k (need $s_m \geq f_k$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve S_m,
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Time

$\Theta(n)$—each activity examined exactly once, assuming that activities are already sorted.



Greedy Algorithm (continued)

- **Iterative version of greedy algorithm**

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}        // Activity a_m being most recently added to A.
        k = m
return A
```

Greedy Algorithm (continued)

- **Elements of greedy strategy**
  - a sequence of choices, each of which is best at the moment
  - key elements: greedy-choice property and optimal substructure
  - greedy algorithm follows three steps listed below (to yield optimal solutions for matriods, i.e., linear independence)

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem (with linear independence)

***Greedy* strategy typically will**

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Greedy Algorithm (continued)

- **Huffman codes**
  - for efficient data compression using variable code length
  - fixed-length code is simple but larger in its encoded footprint
  - prefix codes: no codeword is a prefix of any other codeword

Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

← totally, 100K

Fixed length: 3 bits per symbol type or 300,000 bits for 100K symbols

Variable length: $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits

savings of ~25%

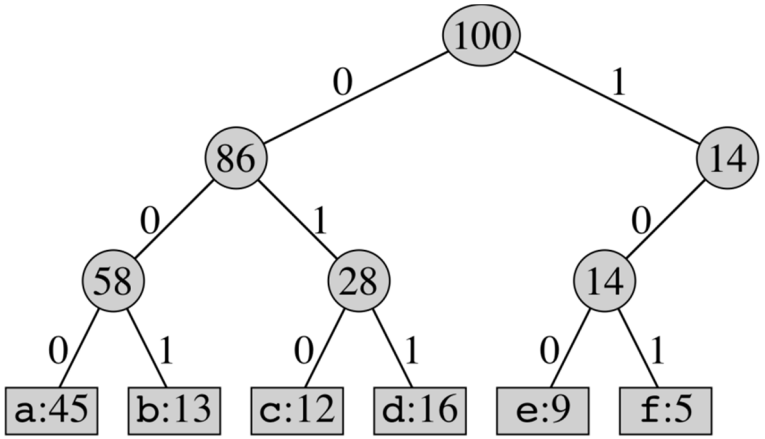
Greedy Algorithm (continued)

- **Prefix codes**
  - no codeword is a prefix of any other codeword
  - simplifying decoding
  - achieving *optimal* data compression for character coding

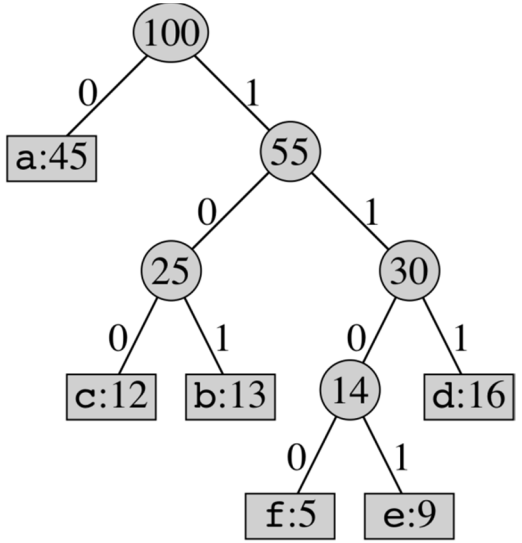
Tree representations of character coding

Number of bits (cost) required of ***T***: $B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$, where $d_T(c)$ is codeword length.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a) Fixed-length coding, with same height for all codewords.



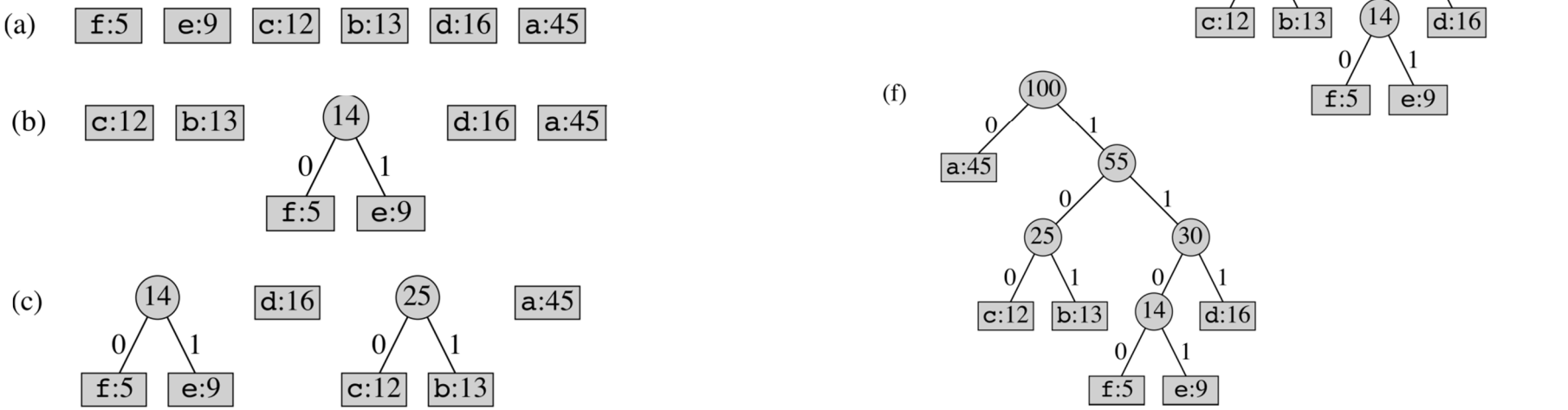
(b) Optimal coding, always denoted by full binary tree with variable heights.

Greedy Algorithm (continued)

- **Huffman code construction**

```
HUFFMAN(C)        C is a set of n characters and each character c ∈ C
                  is an object with an attribute c.freq
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

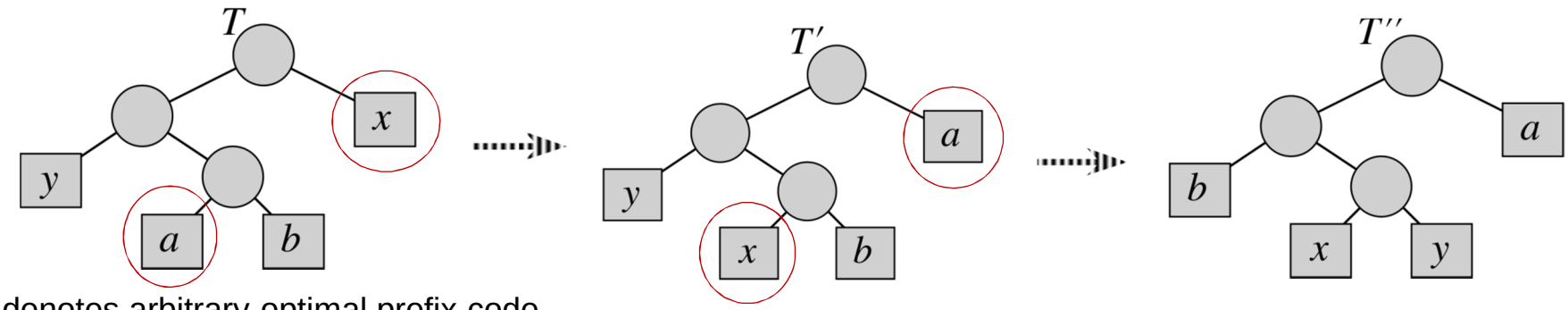
Example



Greedy Algorithm (continued)

Lemma 1

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



T denotes arbitrary optimal prefix code

$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 \text{ , therefore, } B(T) \geq B(T')
\end{aligned}
$$

But, $B(T') \geq B(T)$, given that T is optimal, so $B(T') \equiv B(T)$.

Similarly, $B(T') \geq B(T'')$ to yield $B(T) \geq B(T'')$, since $B(T') \equiv B(T)$.

Given that T is optimal, we have $B(T'') \equiv B(T)$, implying that ***T"*** is another optimal prefix code with x and y differing only in the last bit.

Greedy Algorithm (continued)

Lemma 2

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define $freq$ for C' as for C, except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Theorem

Procedure HUFFMAN produces an optimal prefix code.