Algorithms
演算法
Advanced Design and Analysis Techniques (1)

—— *Dynamic Programming and Greedy Algorithms* ——

Professor Chien-Mo James Li 李建模

Graduate Institute of Electronics Engineering
National Taiwan University

Optimization Problems

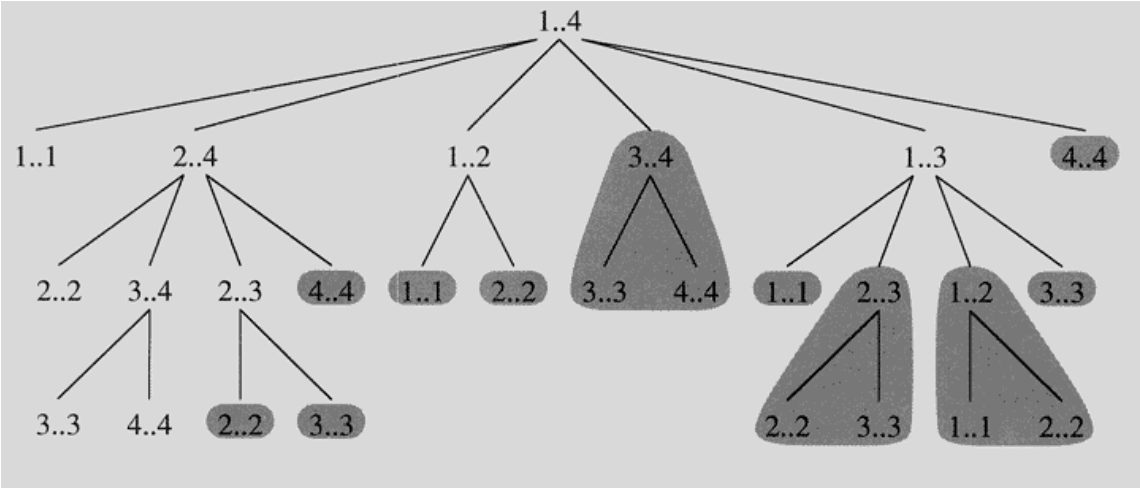
- *Optimization Problems*
  - Find an optimal solution from many possible solutions
  - NOTE: optimal solution *may not* be unique
- Applications of optimization problems:
  - EDA: *electronic design automation*
  - control, operation, our daily lives ...
- Approaches in this lecture:
  - *Dynamic Programming*: (usually bottom up)
    - divide the problem into subproblems, which *overlap*
      - unlike D&C, where subproblems are *disjoint*
    - solve each subproblem just once
    - record the best solution in a table
  - *Greedy Algorithms*: (usually top-down)
    - make a choice that look best at the moment
    - Do NOT always yield an optimal solution.
      - But sometimes they do

Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Dynamic Programming*

- DP solves problems by combining the solution to subproblems
  - subproblems *overlap* (subproblems share subproblems)
    - Unlike divide and conquer, where subproblems are *disjoint*
- DP solve each subproblem *once* and store the results in a table



* "programming" means using a tabular solution to an optimization problem

Dynamic Programming

- Four steps
  - 1. Characterize the structure of an optimal solution.
    - *optimal substructure*
  - 2. *Recursively* define the value of an optimal solution
  - 3. Compute the value of an optimal solution,
    - typically in a *bottom-up* fashion.
    - memorize solutions to many *overlapping subproblems*
  - 4. Construct an optimal solution from step 3.

- Example optimization problems
  - Rod cutting
  - Longest common subsequence
  - Optimal Binary Search Tree

Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue
  - Each cut is free.
  - Rod lengths are always an integral number of inches.
- Input: A rod of length n and a prices table p_i , for $i = 1, 2, ..., n$.

- Output: maximum revenue r_n for rods whose lengths sum to n,
  - computed as the sum of the prices for the individual rods

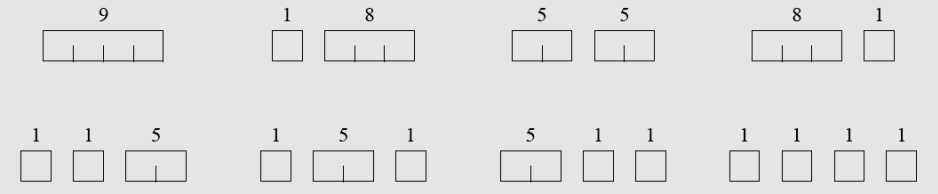
- FFT: how many different ways to cut an n-inch rod? What is the complexity of exhaustive search?
  - assume all cut are integers

Example

- **Price Table: Fig. 15.1**

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

- **8 different ways to cut an 4-inch rod : 2+2 is optimal**



- **optimal solutions n= 1~8**

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6

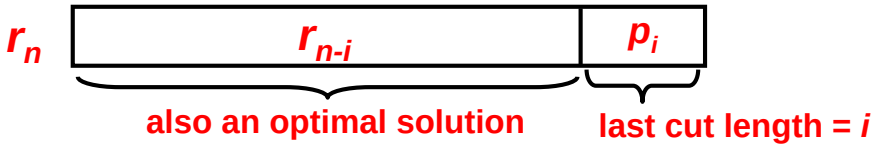
Note:
If p_n is large enough, optimal solution might require no cut

Step 1: Optimal Substructure

- **Idea: Divide this optimization problems into subproblems**
- **suppose r_n = optimal revenue of rod of length n**
- **we can obtain r_n by taking the maximum of**
  - **p_n: the price we get by not cutting,**
  - **p_1+r_{n-1}: maximum revenue from a 1-inch rod and n-1 inches rod,**
  - **p_2+ r_{n-2}: maximum revenue from a 2-inch rod and a n-2 inches rod**
  - **...**
  - **p_{n-1} + r_1**
- **so, r_n = max (p_n, p_1+r_{n-1}, p_2 + r_{n-2},..., p_{n-1} + r_1)**
- **Q: does this approach gurantee to find the optimal solution?**
  - **can you find a better solution that is NOT included in these n subproblems?**

Step 1: Optimal Substructure (2)

- *Optimal Substructure*:
  - To solve the original problem of size *n*, solve subproblems on smaller sizes.
  - The optimal solution to the original problem incorporates optimal solutions to the subproblems.
  - We may solve the subproblems independently.
- For rod cutting, if r_n is optimal solution for length *n*, then the solution to its subproblem (r_{n-i}) is also an optimal cut of length *n-i*
  - Proof:
    - if r_{n-i} is NOT an optimal cut, then replace it by an optimal cut
    - this violate our assumption that r_n is an optimal cut



- NOTE: this statement is NOT reversible
  - putting small optimal solutions together does NOT give a overall optimal solution

Step 2: Recursive Solution

- However, we do not know what was the last cut *i*
  - so we must try all possible values of last cut
  - we are guaranteed to find the optimal solution after trying all *i*
- Recursive version of the equation for r_n

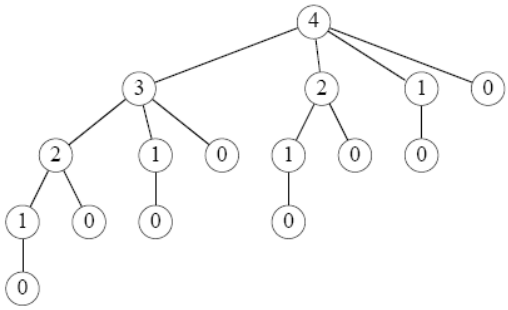
$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Step3: Compute Opt. Solution (Top-Down)

- A direct implementation: CUT-ROD returns the optimal revenue r_n

```
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
```

- Works but inefficient
  - calls itself repeatedly, even on subproblems already solved.
- Example: *n*=4 Solve the subproblem for
  - size 2 twice,
  - size 1 four times,
  - size 0 eight times.



Complexity Analysis

- T (*n*) equal the number of calls to CUT-ROD

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 1. \end{cases}$$

```
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
```

- $T(n) = 2^n$
  - Exponential growth!

Top-down with Memorization

- Solve recursively, but store each result in an array, *r*[*n*]
  - time-memory trade-off.

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

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                     // r[n] has been solved
    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
```


Step3: Compute Opt. Solution (bottom-up)

- Sort the subproblems by size and solve the smaller ones first.

```
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                    // memorization
return r[n]
```

- running time $\Theta(n^2)$
  - doubly nested loop
  - same as Top-down approach
    - use aggregate analysis in ch 17

Step 4: Reconstruct Optimal Solution

- **Modify BOTTOM-UP-CUT-ROD**
  - **s[*j*] = the optimal size of first piece to cut for rod size *j***

```
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        // store size
    r[j] = q
return r and s
```


Exercise

- **how to modify the top-down approach to save size?**

Step 4: Reconstruct Optimal Solution (2)

- print size s[*n*], then continue with *n* = *n*-s[*n*]

```
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]
```

- Example: *n*=7
  - s[7] = 1
  - s[6] = 6

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

- Exercise: *n* = 9
- Exercise: *n* = 10

Four steps of DP (revisit)

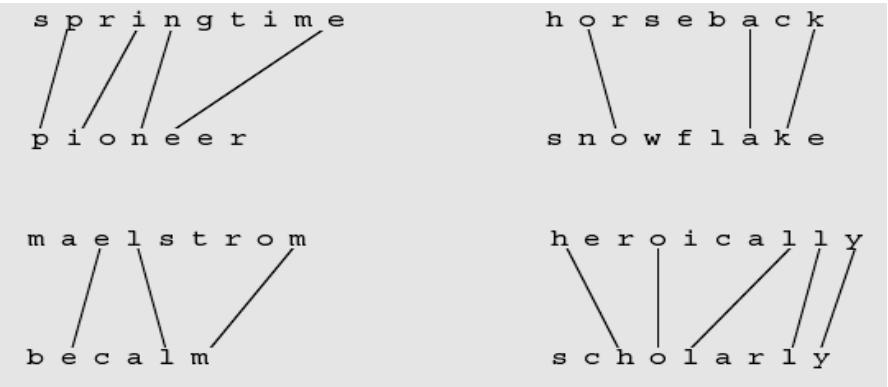
- Four steps
  - 1. Characterize the structure of an optimal solution.
    - *optimal substructure*
    - imagine that *DP God* promised to (1) give you a magic tool that can solve a smaller problem (2) to tell you the *last choice* in the optimal solution...
  - 2. *Recursively* define the value of an optimal solution
    - image that *DP God* is too busy to tell you the last choice .... so you have to search all possibilities
  - 3. Compute the value of an optimal solution,
    - typically in a *bottom-up* fashion.
    - memorize solutions to many *overlapping subproblems*
  - 4. Construct an optimal solution from step 3.

Outline

- **Dynamic Programming, CH15**
  - **rod cutting**
  - **longest common subsequence (LCS)**
  - **optimal binary search tree**
  - **elements of DP**
  - **Matrix Chain (self Study)**
  - **Conclusion**
- **Greedy Algorithms, CH16**
- **Amortized Analysis, CH17**

Longest Common Subsequence (LCS)

- **Problem: Given 2 sequences, $X = \langle x_1, ..., x_m \rangle$ and $Y = \langle y_1, ..., y_n \rangle$.**
  - **Find a subsequence common to both whose length is longest**
  - **subsequence doesn't have to be consecutive**
  - **but it has to be in order.**
- **Example1:**



- **Example 2:**

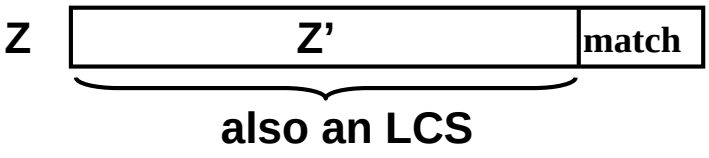
S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
S2=GTCGTTCGGAATGCCGTTGCTCTGTAAA
LCS=GTCGT CGGAA GCCG GC C G AA

Brute Force Approach

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

Step 1: Optimal Substructure

- Notation:
  - X_i = prefix $\langle x_1, \ldots, x_i \rangle$
  - Y_i = prefix $\langle y_1, \ldots, y_i \rangle$
- Theorem 15.1
  - Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of $X=\langle x_1, \ldots, x_m \rangle$ and $Y=\langle y_1, \ldots, y_n \rangle$
  - 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 $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y
  - 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}
- *Optimal substructure*: If Z is an LCS of X and Y, and Z contains a subsequence Z', then Z' must be LCS of prefix of X and Y
  - Proof: cut and paste
    - If Z' is not LCS, replace it by LCS so Z is not longest



Step 2: Recursive Solution

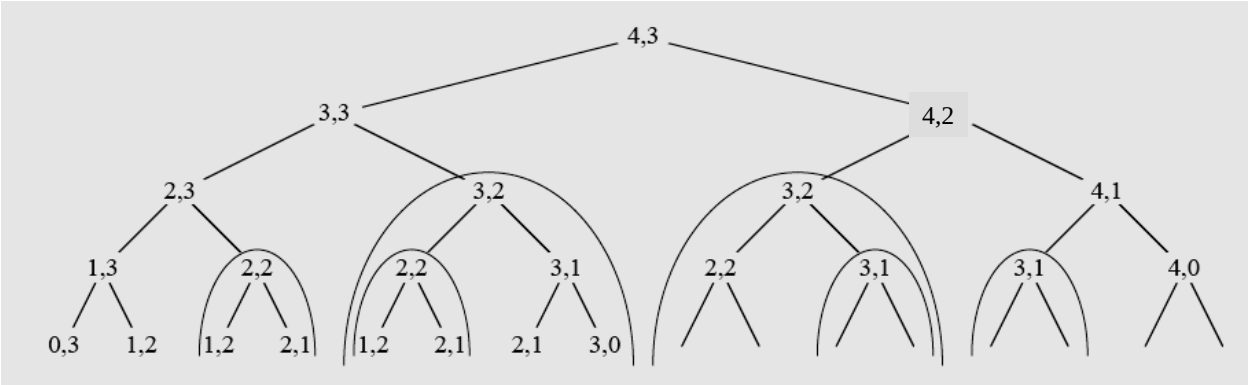
- ***c[i, j]*= LCS length of X_i and Y_j**

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

match: c++

no match:
c = best solution so far

- **Example: bozo, bat**
  - **many overlapping subproblems**



Step 3: Compute Optimal Solution (1)

```
LCS-LENGTH(X, Y, m, n)
let b[1..m, 1..n] and c[0..m, o..n] be new tables
for i = 1 to m                          // initialize b c
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        else if c[i − 1, j] ≥ c[i, j − 1]
                c[i, j] = c[i − 1, j]
                b[i, j] = "↑"
            else c[i, j] = c[i, j − 1]
                b[i, j] = "←"
return c and b
```

i		j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

b[i, j] points to optimal solution of subproblem

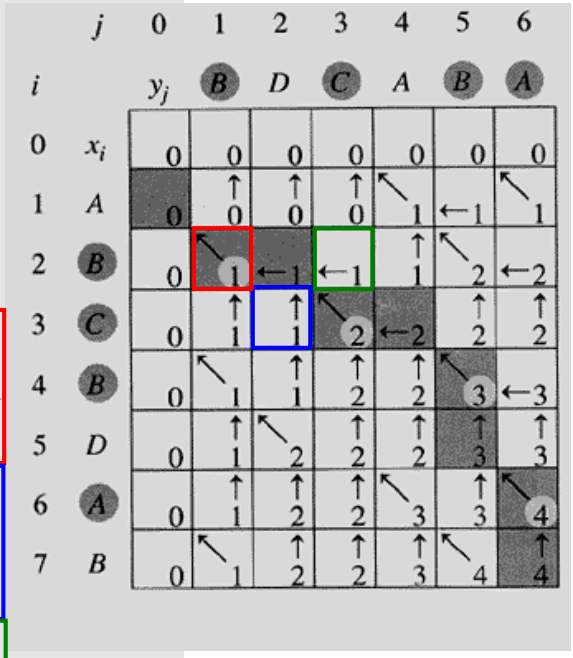
Step 3: Compute Optimal Solution (2)

```
LCS-LENGTH(X, Y, m, n)
let b[1..m, 1..n] and c[0..m, o..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        else if c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

match, c++
move upper left

no match
last match was y

no match
last match was x



Time Complexity

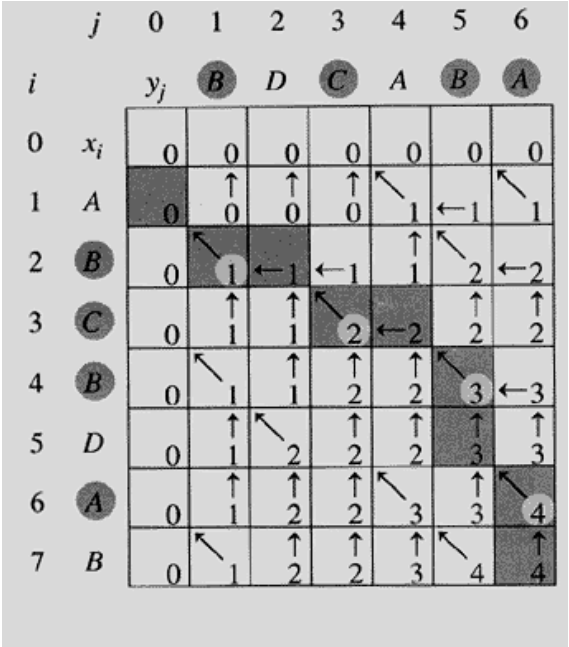
```
LCS-LENGTH(X, Y, m, n)
let b[1..m, 1..n] and c[0..m, o..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        else if c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

- doubly nested loop
  - $\Theta(mn)$

Step 4: Constructing Optimal Solution

- X=ABCBDAB
- Y=BDCABA
- LCS = BCBA
- O(*m*+*n*)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j = 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```



Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Optimal Binary Search Tree (BST)

- $K = \langle k_1, k_2, \ldots, k_n \rangle$ are n distinct keys in sorted order
  - $k_1 < k_2 < \ldots < k_n$
- $P = \langle p_1, p_2, \ldots, p_n \rangle$ are probability that k_i is searched
- $D = \langle d_0, d_1, \ldots, d_n \rangle$ are $n+1$ *dummy keys* for unsuccessful searches
- $Q = \langle q_0, q_1, q_2, \ldots, q_n \rangle$ are probability that d_i is searched

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

- expected *search cost*

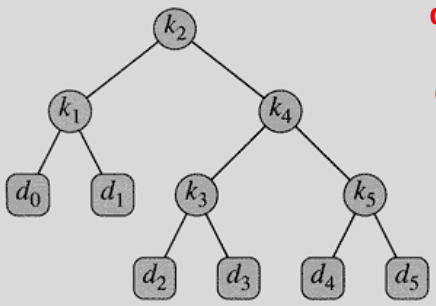
$$E[\text{search cost in } T] = \sum_{i=0}^{n} (depth_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i$$

- *Optimal BST* = the BST of lowest search cost

Example

- Fig 15.9. ITW $d_0, k_1, d_1, k_2, \ldots, d_4, k_5, d_5$



(a)

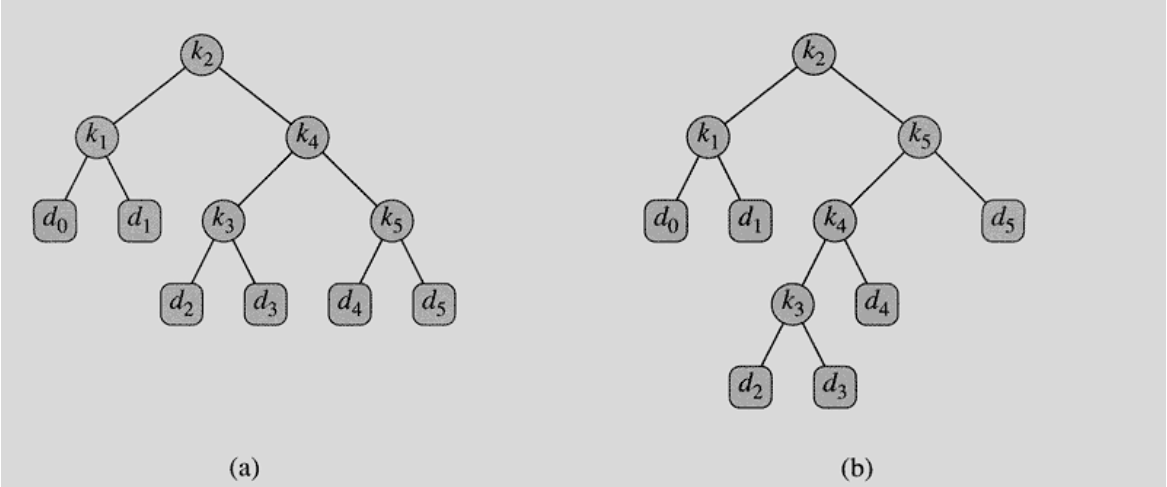
node	Depth	Probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
…			
d_0	2	0.05	0.15
d_1	2	0.10	0.30
…			
total			2.80

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

Example (cont'd)

- cost of left tree = 2.80; cost of right tree = 2.75 → optimal BST



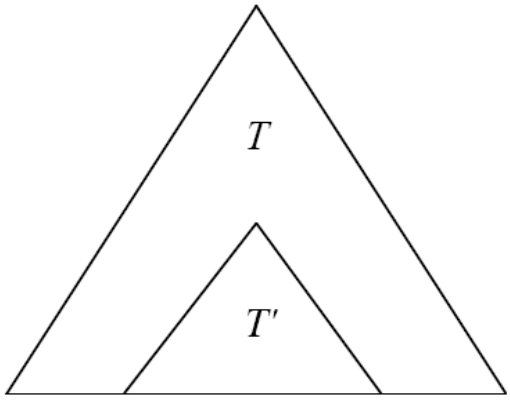
Two binary search trees for a set of $n = 5$ keys with the following probabilities:

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

(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.

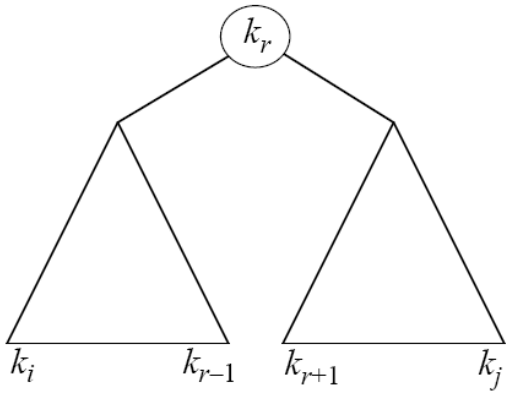
Step 1: Optimal Substructure

- If an optimal BST *T* contains subtree *T'* with keys $k_i,\ldots, k_j$,
  - *T'* must be optimal BST for keys $k_i,\ldots, k_j$, and dummy keys $d_{i-1},\ldots,d_j$
- Proof: *cut and paste*
  - If *T'* is NOT optimal, then we just replace *T'* by an optimal one
    - Violate our assumption that *T* is optimal



Cost of *T'*

- Given a subtree *T'*, k_r is the root, where $i \le r \le j$
  - Left subtree of k_r contains k_i ,..., k_{r-1}
  - Right subtree of k_r contains k_{r+1} ,..., k_j
- we're guaranteed to find an optimal subtree *T'* for k_i,..., k_j as long as
  - 1. we examine all candidate roots k_r , $i \le r \le j$, and
  - 2. we determine all optimal BSTs containing k_i,..., k_{r-1} and containing k_{r+1},..., k_j ,



Step 2: Recursive Solution

- *e[i, j]* = expected search cost of optimal subtree *T'*, k_i,..., k_j
- *w[i,j]* = sum of probabilities of subtree *T'* , k_i,..., k_j

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

- If k_r is the root of optimal subtree *T'*

$$e[i, j] = \underbrace{p_r}_{\text{root}} + \underbrace{(e[i, r-1] + w(i, r-1))}_{\text{left subtree (optimal)}} + \underbrace{(e[r+1, j] + w(r+1, j))}_{\text{right subtree (optimal)}}$$

$$\because w(i, j) = w(i, r-1) + p_r + w(r+1, j)$$

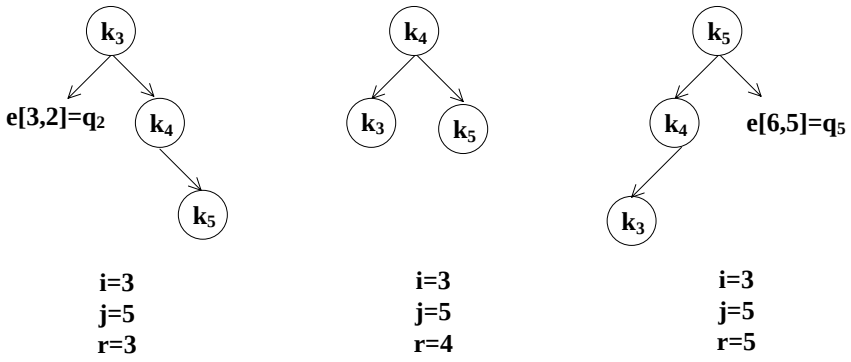
$$\therefore e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$$

Step 2: Recursive Solution (cont'd)

- **But we do not know which *r* is optimal, so try all *r* (*eq* 15.14)**

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

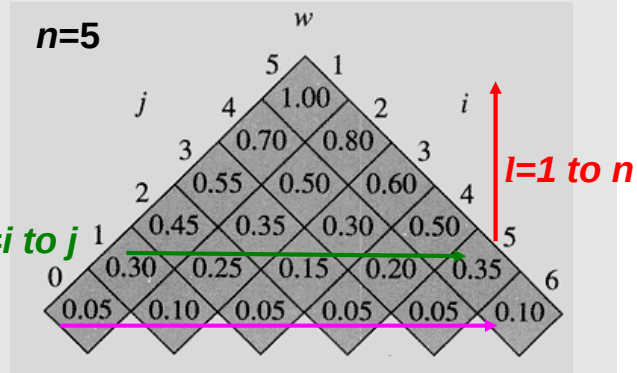
NOTE: *j=i-1* is an *empty subtree* containing only d_{i-1} so cost = q_i



i=3 j=5 r=3 | i=3 j=5 r=4 | i=3 j=5 r=5

Step 3: Compute Optimal Solution (1)

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
for i = 1 to n+1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}          empty subtree
for l = 1 to n
    for i = 1 to n-l+1
        j = i+l-1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```



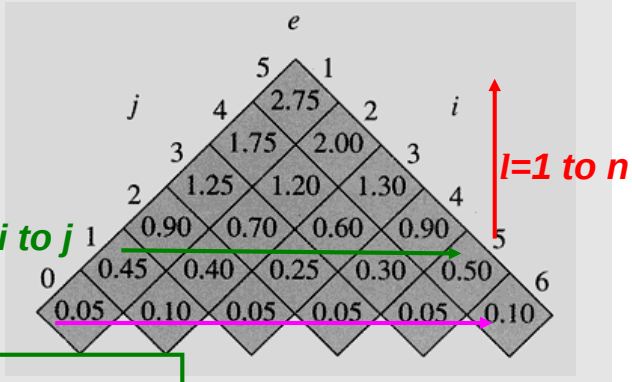
$w_{11}=q_0+p_1+q_1=0.3$
$w_{12}=w_{11}+p_2+q_2=0.45$

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

Step 3: Compute Optimal Solution (2)

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
for i = 1 to n+1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n-l+1
        j = i+l-1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

empty subtrees

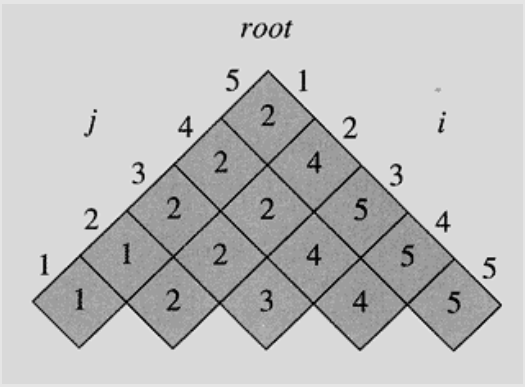


$e_{11}=e_{10}+e_{21}+w_{11}$
$e_{12}=min(e_{10}+e_{22}+w_{12}, e_{11}+e_{32}+w_{12})$

recursive solution (eq 15.14)

Step 3: Compute Optimal Solution (3)

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
for i = 1 to n+1
    e[i, i-1] = 0
    w[i, i-1] = 0
for l = 1 to n
    for i = 1 to n-l+1
        j = i+l-1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```



roots of optimal subtrees

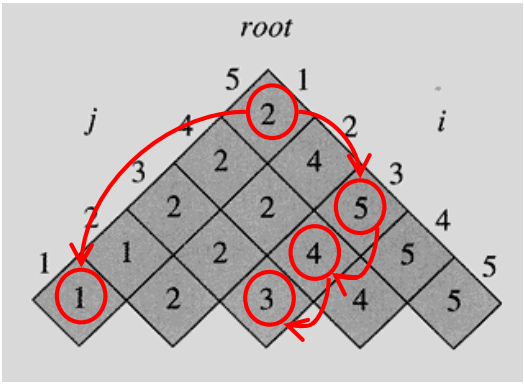
Running Time

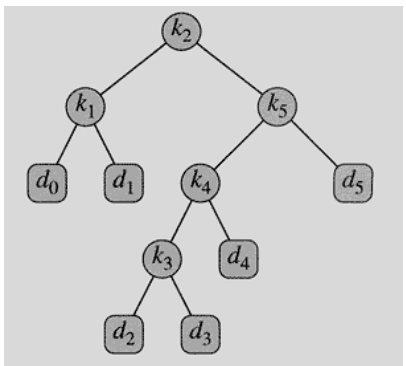
- three-level nested loops
  - $\Theta(n^3)$
  - Exercise 15.2-5

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i − 1] = 0
    w[i, i − 1] = 0
for l = 1 to n
    for i = 1 to n − l + 1
        j = i + l − 1
        e[i, j] = ∞
        w[i, j] = w[i, j − 1] + p_j
        for r = i to j
            t = e[i, r − 1] + e[r + 1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```


Step 4: Construct Optimal BST

- Your exercise!





Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Elements of DP

- Two key elements
  - *optimal substructure*
    - the solutions to the subproblems used within the optimal solution must themselves be optimal.
  - *overlapping subproblems*
    - recursively solve the same subproblems over and over again
      - not brand new subproblems

Optimal Substructures

- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal
  - How to prove? Usually use *"cut-and-paste"*
- Two questions to ask:
  - 1. How many subproblems are used in an optimal solution
  - 2. How many choices in determining which subproblem(s) to use
- Example:
  - rod cutting:
    - 1 subproblem of size $n-i$, for $1 \le i \le n$
    - less than n choices for each subproblem
  - LCS
    - 1 subproblem
    - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
    - 2 choices (if $x_i \neq y_j$, LCS of X_{i-1} and Y , and LCS of X and Y_{j-1})
  - optimal BST
    - 2 subproblems ($k_1,..., k_{r-1}$) and ($k_{r+1},..., k_j$)
    - $j-i+1$ choices for root in $k_i,...,k_j$

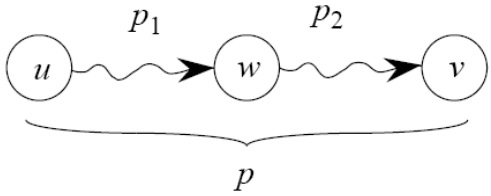
Running Time

- Running time depends on
  - (# of total subproblems) times (# of choices in each subprblems)

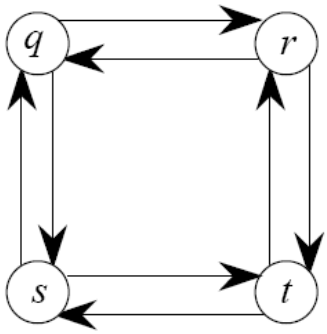
- Examples:
  - rod cutting
    - $\Theta(n)$ subproblems, $\le n$ choices for each
    - $O(n^2)$ running time
  - LCS
    - $\Theta(mn)$ subproblems, 2 choices for each
    - $\Theta(mn)$ running time
  - OBST
    - $\Theta(n^2)$ subproblems, $\Theta(n)$ choices for each
    - $\Theta(n^3)$ running time

Subtleties

- *Shortest path* :find *simple path (*no cycle) *u→v* with fewest edges
  - it has optimal substructure
    - any subpath is optimal
      - p_1 is shortest
      - p_2 is shortest



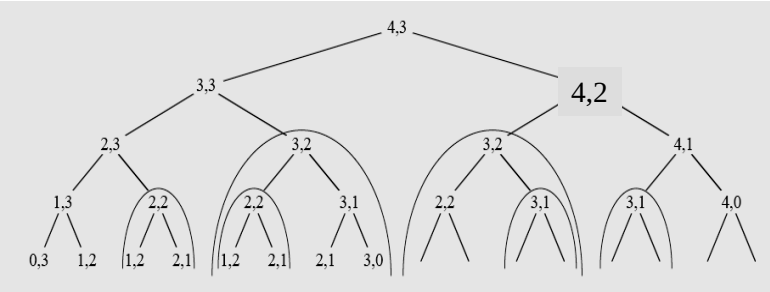
- *Longest simple path :* find *simple path u→v* with most edges
  - it does NOT has optimal substructure
    - Q1: what is longest path
      - *q→t*
    - Q2: what are longest paths
      - *q→r*
      - *r→t*
    - Q1 does NOT contain Q2



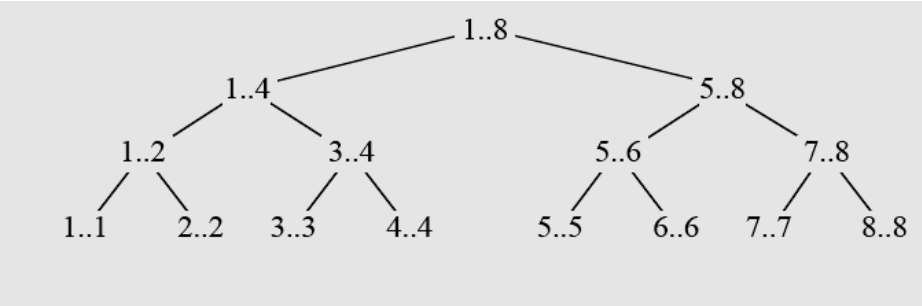
- Why? because simple longest paths are *dependent*
  - Solution to one subproblem affects solution to other sub problems
    - Since first subproblem chooses q→s→t→r
    - s and t cannot be used in the second subproblem

Overlapping Subproblems

- DP recursively solve the same subproblems over and over again
  - Example: rod cutting



- NOTE: divide-and-conquer generates brand new subproblems
  - Example: merge sort is not DP



Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Matrix Chain Multiplication

- Given $A = A_1 x A_2 x \ldots x A_n$
  - Find A using the minimum number of multiplications
- A product of Matrices is *fully parenthesized* if it is a single matrix or the produce of 2 fully parenthesized matrix products
- Example:
  - $(A_1(A_2(A_3A_4)))$
  - $(A_1((A_2A_3)A_4)$
  - ...
- *C=AB*
  - if size *A* is *pxq*, szie *B* is *qxr* then *pqr* multiplications is needed
- The order of multiplication makes a big difference
  - Example: A1: 10x100, A2: 100x5, A3: 5x50
    - (A1A2)A3 needs 7,500 multiplications
    - A1(A2A3) needs 75,000 multiplications!

Brute Force

- Check all possible orders
- *P*(*n*): number of ways to multiply *n* matrices.

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

- *P*(*n*) is a sequence of *Catalan* numbers, which grows exponentially!
  - $\Omega(4^n / n^{3/2})$
- How to solve it smartly? using DP

Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Compare Two Approaches

- 1. *Bottom-up* iterative approach
  - Start with recursive divide-and-conquer algorithm.
  - Find the dependencies between the subproblems (which solutions are needed for computing a subproblem).
  - Solve the subproblems in the correct order.
- 2. *Top-down* recursive approach (memorization)
  - Start with recursive divide-and-conquer algorithm.
  - Keep top-down approach of original algorithms.
  - Save solutions to subproblems in a table
  - Recurse only on a subproblem if the solution is not already available in the table.
- If all subproblems must be solved *at least once*, bottom-up DP is better due to less overhead for recursion and for maintaining tables.
- If many (but not all) subproblems need not be solved, top-down DP is better since it computes only those required.

When to Use DP

- DP computes recurrence efficiently by storing partial results
  - It is efficient only when small number of partial results.
- DP is NOT suitable for :
  - *n!* permutations of an *n*-element set,
  - 2^n subsets of an *n*-element set, etc.
- DP is suitable for:
  - contiguous substrings of an *n*-character string,
  - $n(n+1)/2$ possible subtrees of a binary search tree, etc.
- DP works best: objects that are linearly ordered and cannot be rearranged
  - Matrices in a chain
  - Characters in string
  - DNA sequence

Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

Activity Selection Problem

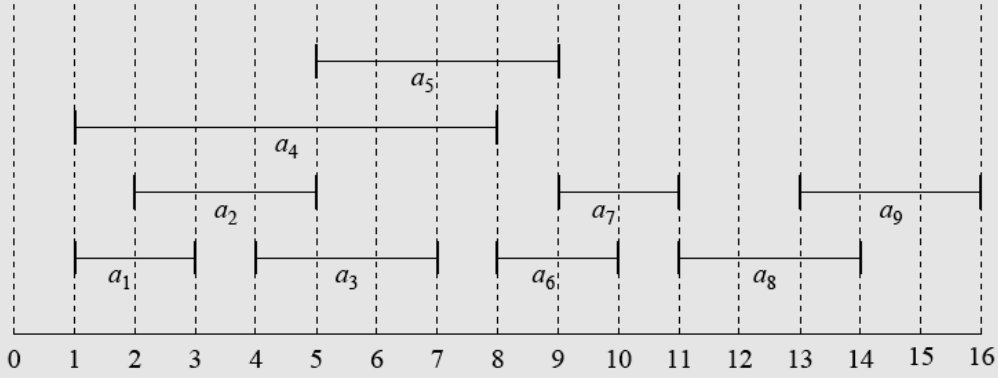
- ***n*** **activities require exclusive use of a common resource**
  - **For example, scheduling the use of a classroom**
- **Set of activities** $S = \{a_1 \ldots a_n\}$
  - a_i **needs resource during period** $[s_i, f_i]$,
    - **where** s_i **= start time and** f_i **= finish time**
- **Goal: Select the largest possible set of non-overlapping activities**
  - a_i **and** a_j **are *compatible* if their intervals do not overlap**

- **Assume that activities are sorted by finish time:**
  - $f_1 \le f_2 \le f_3 \le \ldots f_{n-1} \le f_n$

Example

- ***S*** **sorted by finish time**

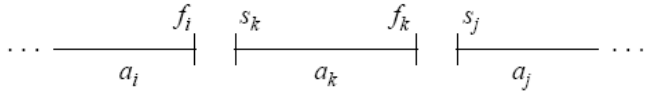
i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



- **One possible solution {a_1, a_3, a_9}: not maximum**
- **Maximum-size mutually compatible set: {a_1, a_3, a_6, a_8}.**
  - **Not unique: {a_2, a_5, a_7, a_9} also maximum**

Step1: Optimal Substructure

- **S_{ij} =set of activities that**
  - **starts after a_i finished and finishes before a_j starts**
  - **{$a_k \in S$; $f_i \le s_k < f_k \le s_j$ }**
- **Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}**
- **Suppose we already know $a_k \in A_{ij}$. Then we have two subproblems:**
  - **Find $A_{ik} = A_{ij} \cap S_{ik}$ = set of activities in A_{ij} that finish before a_k**
  - **Find $A_{kj} = A_{ij} \cap S_{kj}$ = set of activities in A_{ij} that start after a_k**
  - **Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$**
- **Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj} .**
- **Proof: cut and paste**
  - **Suppose we could find a better set A_{kj}' in S_{kj} , where $|A_{kj}'|>|A_{kj}|$**
  - **Then use A_{kj}' instead of A_{kj}**



Step2: Recursive Solution

- Let $c[i,j] = |A_{ij}|$ = size of optimal solution in S_{ij}
  - $c[i, j] = c[i, k] + c[k, j] + 1$
- But actually we do not know a_k
  - exhaustively try all possible a_k

$$c[i,j] = \begin{cases} 0 & if\ s_{ij} = \phi \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & if\ s_{ij} \neq \phi \end{cases}$$

- We could solve the problem like DP
  - but we ignore some important characteristic of this problem
  - can we solve the problem *before* solving the subproblems?

Early Bird Always Wins

- (Theorem 16.1) 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
  - Proof: cut and paste
  - Let A_k be an optimal solution to S_k
  - Let a_j have the earliest finish time of any activity in A_k
    - If $a_j = a_m$, done
    - Otherwise, let $A_k' = A_k - \{a_j\} \cup \{a_m\}$
      - substitute a_m for a_j .
  - activities in A_k' are disjoint because activities in A_k are disjoint
    - a_j is first activity in A_k to finish so $f_m \leq f_j$
  - Since $|A_k'| = |A_k|$, conclude that A_k' is an optimal solution to S_k, and it includes a_m

- So, don't need dynamic programming. Don't need to work bottom up
- Instead, can just repeatedly choose the activity that finishes first,
  - keep only the activities that are compatible with that one, and repeat until no activities remain

Recursive Greedy Algorithm

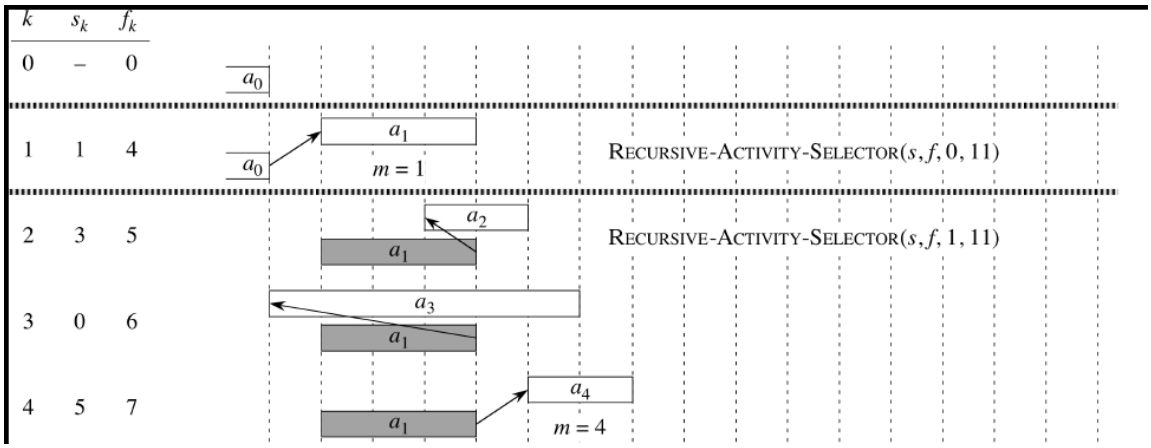
- **s =start time array, s[0]=0**
- ***f* = finish time array, sorted in monotonically increasing order**
- ***n* = numbers of total activities**
- ***k* = index of currently selected activity, k=1 to *n***
- **initial call REC-ACTIVITY-SELECTOR (*s, f, 0, n*)**

```
REC-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]          // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

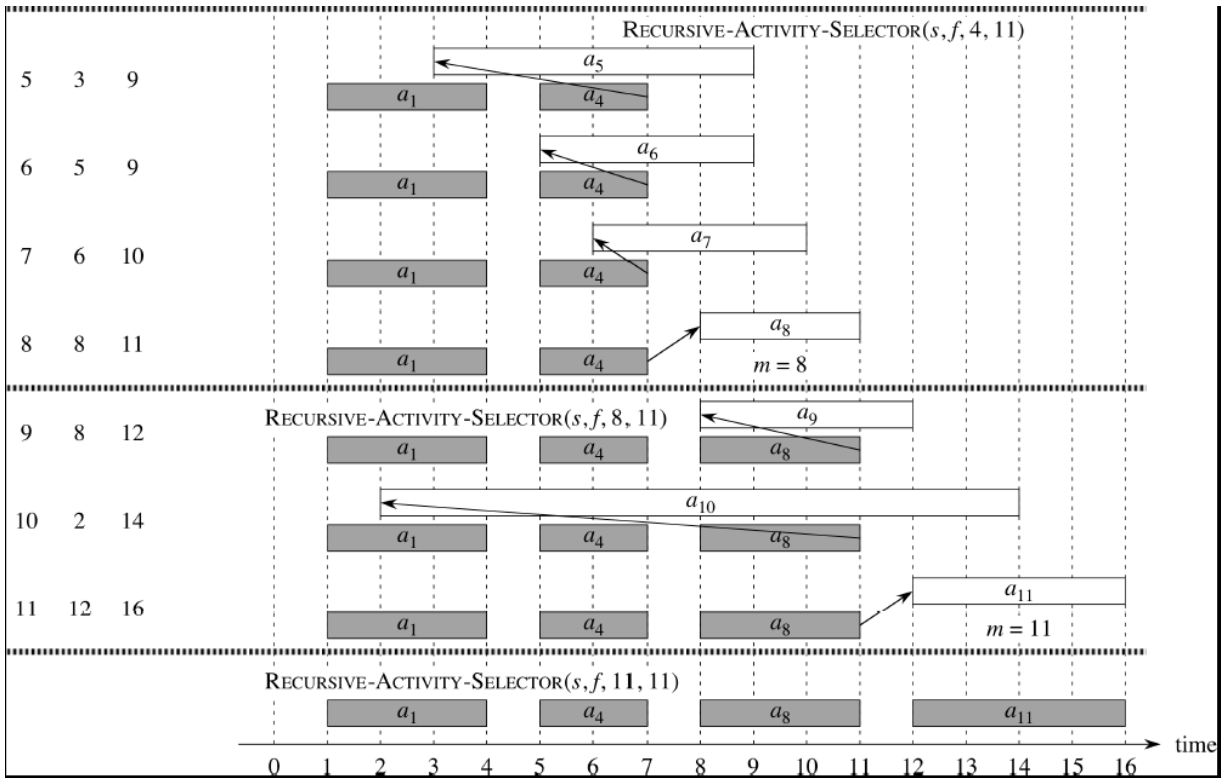
- **time complexity = ?**

Operation

- **Fig 16.1**



Operation (2)



Iterative Greedy Algorithm

- **s = start time array**
- ***f* = finish time array, sorted in monotonically increasing order.**
- **same time complexity as before**
- **but less overhead of recursive call of functions**

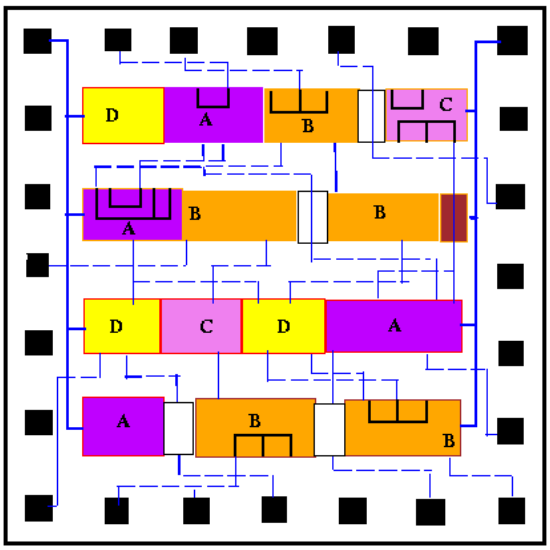
```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

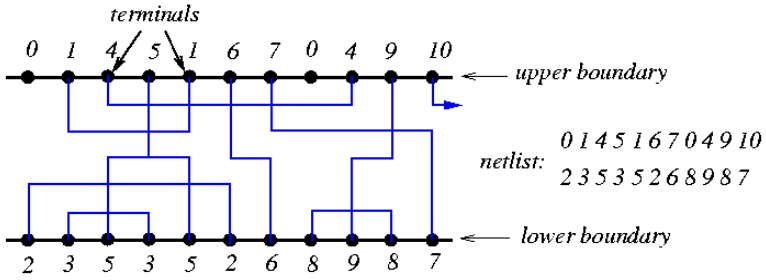
Standard Cell Design

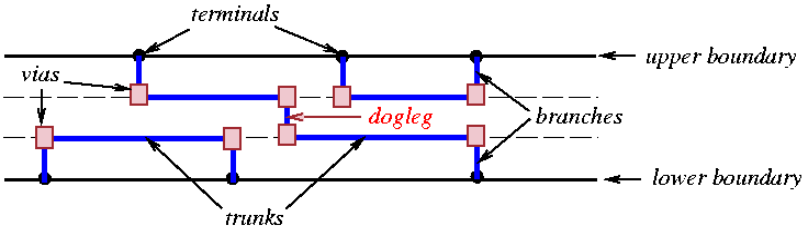
- Logic gates are pre-designed cells of the same height
- Interconnects are routed in *routing channels*
- very flexible and scalable design style



Terminology for Channel Routing

- ***Terminals*** **of the same number should be connected together**
- **Horizontal segments are *trunks***
- ***Branches*** **are vertical segment between trunk and terminal**
- ***Doglegs*** **are small vertical segments between two trunks**





Channel Routing Problem

- **Assign horizontal segments (trunks) to *tracks***
  - **All terminals must be connected**
- **Constraints**
  - **Assume one vertical layer and one horizontal layer**
  - ***Horizontal constraints***
    - **two trunks in the same track CANNOT overlaps each other**
  - **Vertical constraints**
    - **Two braches CANNOT overlap each oterh**
- **Objective: Channel height is minimized**
  - **minimized the number of tracks**
- **Very important EDA problem**

Left-Edge Algorithm

- **Hashimoto & Stevens, "Wire routing by optimizing channel assignment within large apertures," DAC-71.**

- **Treat each horizontal segment as an *interval***
- **Intervals are sorted according to their left-end *x*-coordinates**
- **Intervals are routed one-by-one according to the order**
- **For a net, tracks are scanned from top to bottom**
  - **first track that accommodate the net is assigned to the net**
- **Always produces an optimal routing solution with the minimum # of tracks**
  - **if no vertical constraint**

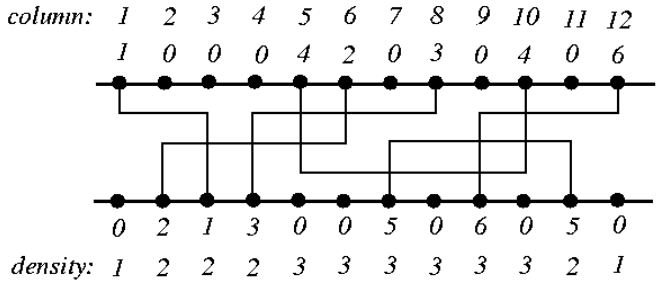
Left-Edge Algorithm

Left-Edge (U, *track*[j])
// U= set of unassigned intervals (nets) $I_1, \ldots, I_n$;
// I_j=[s_j, e_j]: interval j with left-end x-coordinate s_j and right-end e_j;
// *track*[j]: track to which net j is assigned

$U = \{I_1, I_2, \ldots, I_n\}$;
$t = 0$;
while ($U \neq \varnothing$) **do**
    $t = t + 1$;
    watermark = 0; // watermark is right end of this track so far
    while (there is an $I_j \in U$ s.t. $s_j >$ *watermark*) **do**
      Pick the interval $I_j \in U$ with $s_j >$ *watermark*, nearest *watermark*;
      track[j] = t;
      watermark = e_j;
      $U = U - \{I_j\}$;
end

Basic Left-Edge Example

- $U = \{I_1, I_2, \ldots, I_6\}$
  - $I_1 = [1, 3]$, $I_2 = [2, 6]$, $I_3 = [4, 8]$, $I_4 = [5, 10]$, $I_5 = [7, 11]$, $I_6 = [9, 12]$.
- t =1:
  - Route I_1: *watermark* = 3;
  - Route I_3 : *watermark* = 8;
  - Route I_6: *watermark* = 12;
- t = 2:
  - Route I_2 : *watermark* = 6;
  - Route I_5 : *watermark* = 11;
- t = 3: Route I_4



Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

Elements of Greedy Algorithms

- Six steps to develop a greedy algorithm
  - 1. Determine the *optimal substructure*
  - 2. Develop a recursive solution.
  - 3. Show that if we make the greedy choice, only one subproblem remains
  - 4. Prove that it's always safe to make the greedy choice
    - *greedy choice property*
  - 5. Develop a recursive greedy algorithm
  - 6. Convert it to an iterative algorithm

Key Ingredients of Greedy

- 1. G*reedy-choice Property*
  - Can assemble a globally optimal solution by making locally optimal (greedy) choices
  - Example: activity selection:
    - Look at an optimal solution
    - If it includes the greedy choice, done
    - Otherwise, modify the optimal solution to include the greedy choice, yielding another solution that's just as good
  - different from DP

- 2. Optimal Substructure
  - Optimal solution to subproblem is included in optimal solution to the whole problem
  - same as DP

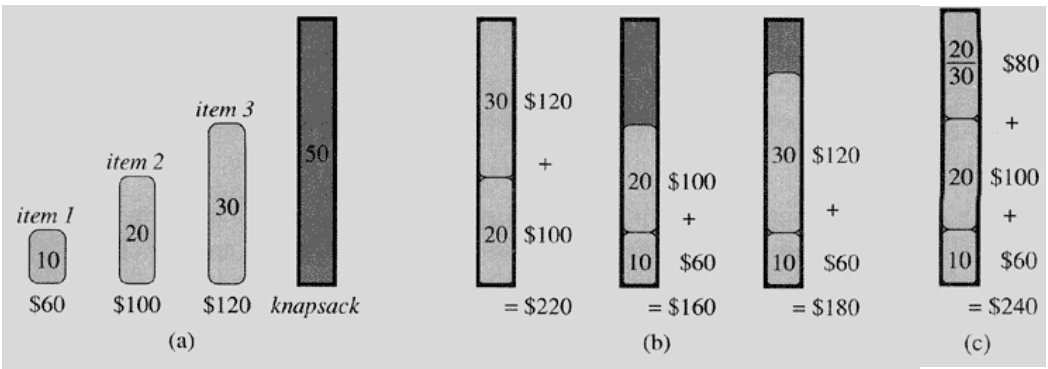
DP vs. Greedy

- Dynamic programming
  - Make a choice at each step
  - Choice depends on knowing optimal solutions to subproblems
  - Solve subproblems first — *bottom-up*
- Greedy
  - Make a choice at each step
  - Make the choice *before* solving the subproblems
  - Solve problems *top-down*

- Both greedy and DP exploits optimal substructure
  - sometimes we can get confused
- Two examples to demonstrate the difference
  - *Fractional Knapsack Problem*
  - *0-1 Knapsack Problem*

Fractional Knapsack Problem

- A thief has *n* items to steal
- Item *i* is worth $\$v_i$, weighs w_i pounds
- Find a most valuable subset of items with total weight $\leq W$
- *Can take a fraction of an item*
- Example: Fig. 16.2
  - optimal solution: item 1+ item2 + 2/3 item 3
- Greedy algorithm applies



Greedy Algorithm

- Rank items by value/weight: v_i/w_i
- Take items in decreasing order of value/weight
- Possibly a fraction of the item
- Time complexity:
  - O(*n* lg *n*) to sort, O(*n*) thereafter

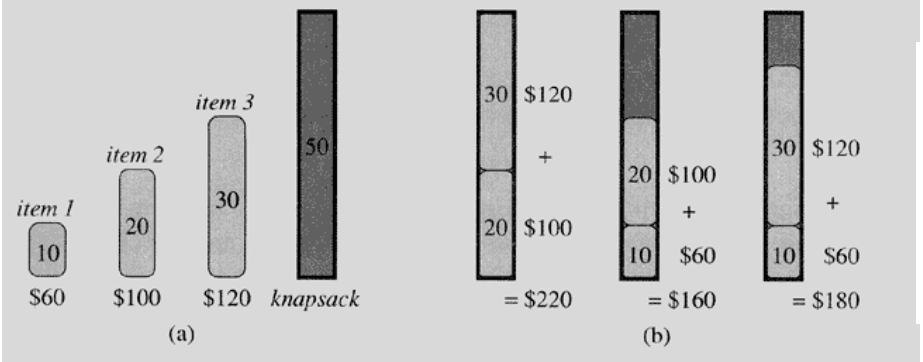
```
FRACTIONAL-KNAPSACK(v, w, W)
load = 0
i = 1
while load < W and i ≤ n
    if w_i ≤ W − load
        take all of item i
    else take (W − load)/w_i of item i
    add what was taken to load
    i = i + 1
```

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50.$

0-1 Knapsack Problem

- Have to either take an item or not take it — can't take part of it
- Example: Fig. 16.2
  - optimal solution: item3 + item2
- Greedy strategy does NOT apply
- Need to use DP
  - O(*nW*) time, exercise 16.2-2

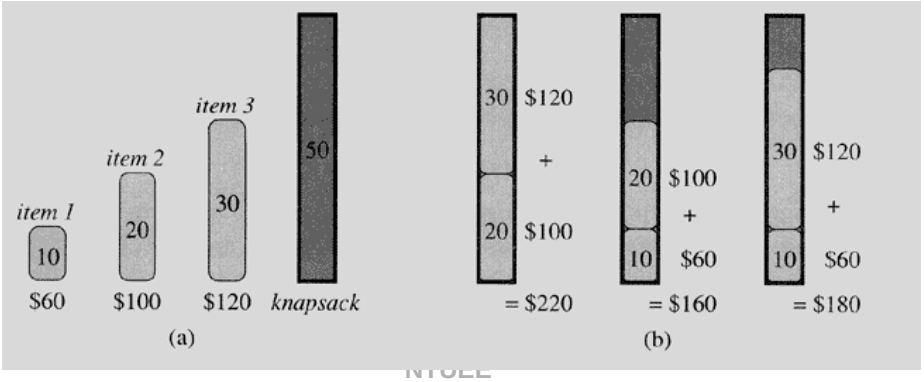


i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50.$

WHY?

- Both problems have optimal substructure
- But fractional knapsack problem has the greedy-choice property
  - the 0-1 knapsack problem does not
- FFT
  - In 0-1 Knapsack problem, what if we take items in the order of their values? Does greedy algorithm work in this way?
    - Yes, prove the greedy-choice property
    - No, give an counter example

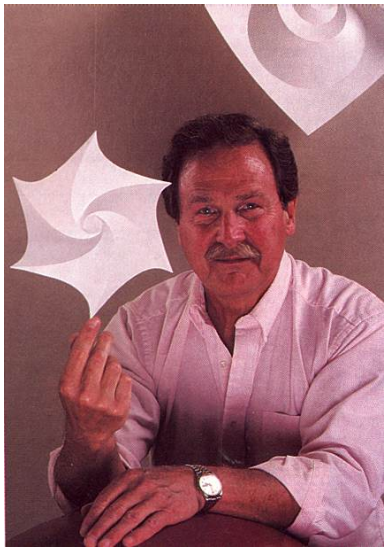


Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

David A. Huffman [1925-1999 USA]

- In 1951 Huffman (aged 25) and his classmates in an electrical engineering graduate course on information theory were given the choice of a term paper or a final exam. Huffman's professor, Robert M. Fano, had assigned what at first appeared to be a simple problem. Students were asked to find *the most efficient method of representing numbers*, letters or other symbols using a binary code.
- Huffman worked on the problem for months, developing a number of approaches, but failed. *Just as he was throwing his notes in the garbage,* the solution came to him.
- Huffman says he might never have tried his hand at the problem *if he had known that Fano, his professor, and Claude E. Shannon, the creator of information theory, had struggled with it*.



Coding

- Application: data compression, error correction, encryption etc
- *Binary Character Code*: each character represented by a unique binary string (*codeword*)
  - *Fixed-length code:* codeword are the same length
    - e.g. a=000, b=001, ... f=101
      - afb=000 101 001
  - *Variable-length code:* codeword are different in length
    - e.g. Huffman code
      - frequent characters are represented by shorter codeword
- Example: Fig 16.3, 100K characters (a ~ f)
  - fixed length coding: 300K bits
  - variable length coding: 224 K bits

	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

Prefix Code

- ***Prefix Code*: No code is a prefix of some other code**
- **Example**
  - ***C={a, b, c, d, e, f}* *alphabet*, the set of characters in use**

	a	b	c	d	e	f
Prefix code	0	101	100	111	1101	1100
Non-prefix	000	0001	001	0011	111	1110

- ***Encoding* is always simple**
  - **just concatenation**
    - **e.g. afb=0 1100 101**
- ***Decoding* is easy when prefix code is used**
  - **decode each character is unambiguous**
    - **e.g. 001011101 → 0 0 101 1101**
  - **Decoding for non-prefix code is ambiguous**

Tree Representation

- **Decoding process of prefix code**
  - **Codeword is represented by a simple path from root to leave**

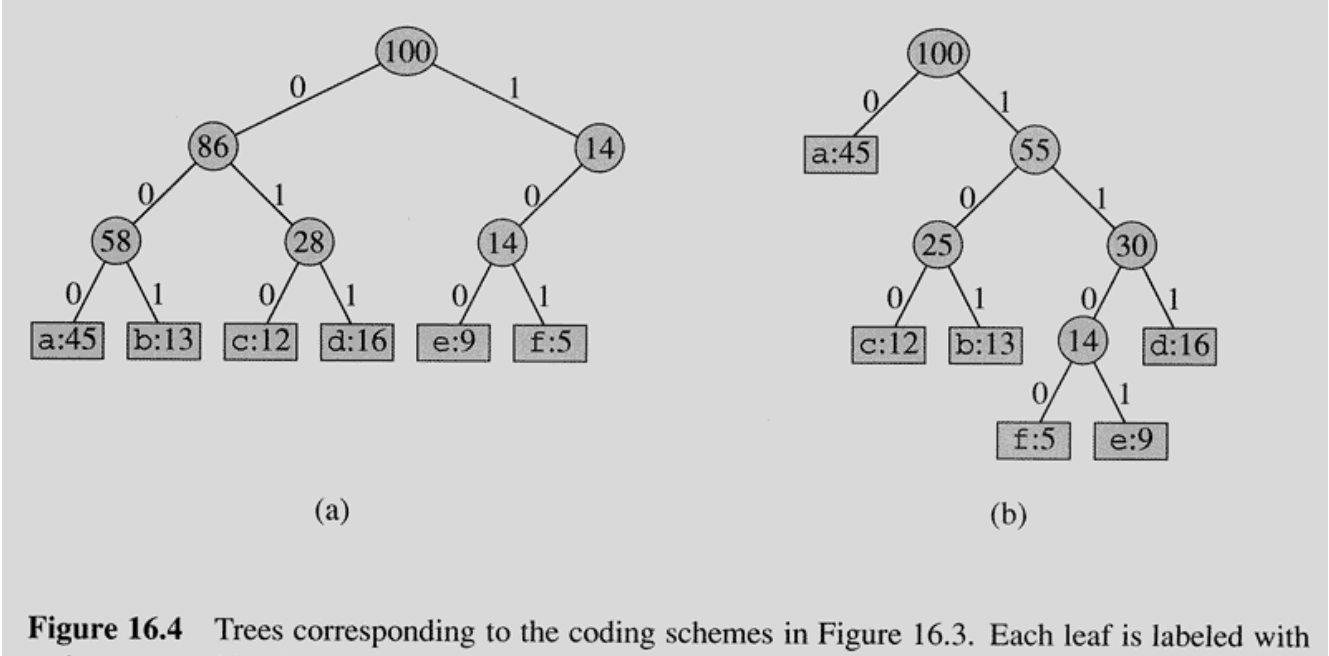
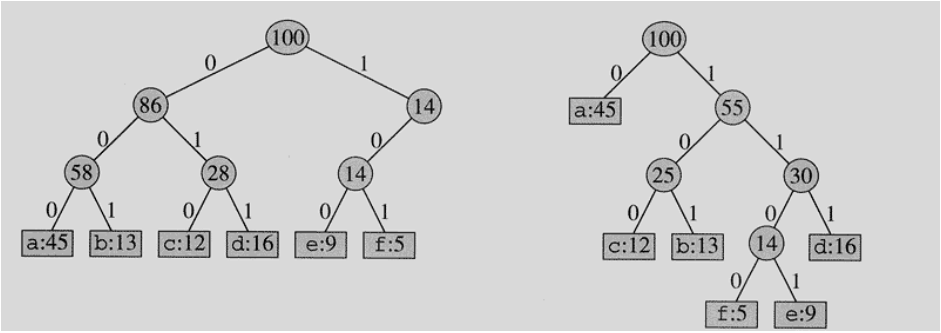


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with

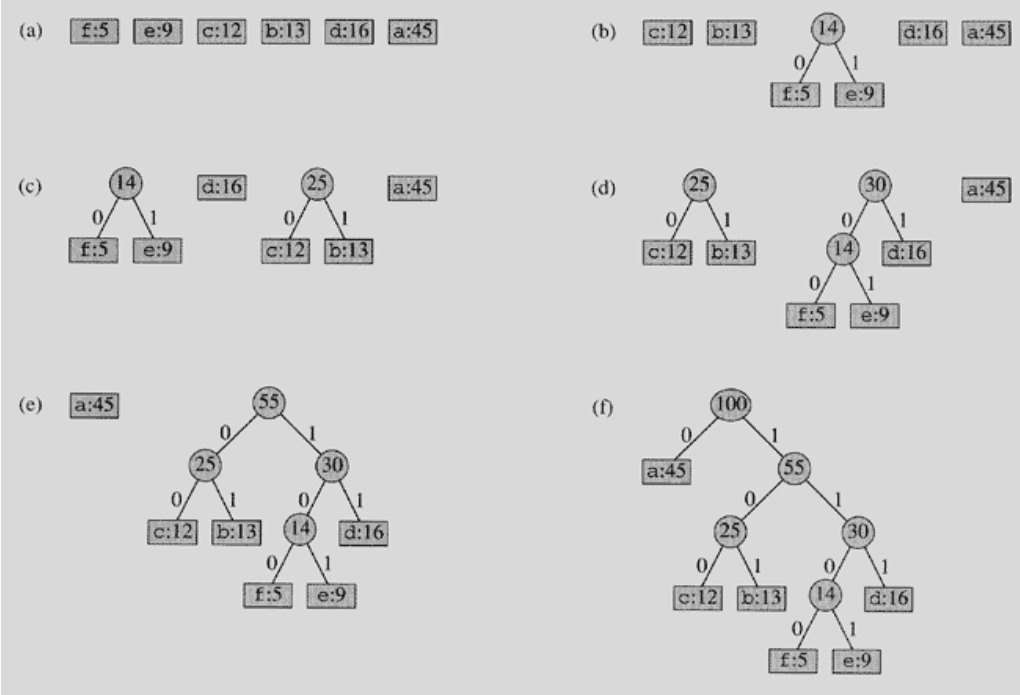
Optimal Prefix Code Design Problem

- **Input: given a set of *alphabet* and the frequency of each character**
- **Output: find a binary tree such that cost is minimized**
  - ***c.freq* is the frequency of appearance of character *c***
  - ***$d_T(c)$* is depth of *c's* leaf in the tree = length of *c*'s codeword**
  - ***cost of the tree*** $B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$
- **Optimal prefix code is always represented by a *full binary tree***
  - **every non-leaf node has two children**
  - **optimal tree has |*C*| leaves and |*C*|-1 nodes**
- **Example (a) cost = 300, not full (b) =224, optimal cost, full tree**



Huffman's Algorithm

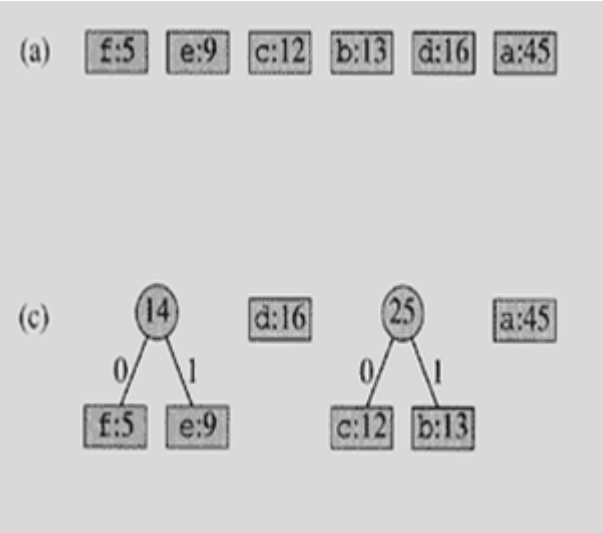
- **Idea: more frequent characters are deeper in the tree**
- **Bottom up: combine two nodes with the least cost at each step**



Huffman's Pseudo Code

- **Time complexity: *Q* is implemented in min-heap**
  - **loop *n*-1 times, each EXTRACT-MIN is *O*(lg *n*)**
  - **totally O(*n* lg *n*)**

```
HUFFMAN (C)
n=|C|
Q=C
for i = 1 to n-1
   allocate a new node z
   z.left = x = EXTRACT-MIN (Q)
   z.right= y = EXTRACT-MIN(Q)
   z.freq=x.freq+y.freq
   INSERT(Q,z)
return EXTRACT-MIN(Q)
```



Greedy-choice Property

- **Lemma 16.2: Two characters *x* and *y* with the lowest frequencies must have the same length and differ only in the last bit**
  - **Proof: suppose tree *T* is optimal, x and y are NOT at bottom**
    - **swapping *x* with *a*, *y* with *b* (does not increase cost)**
    - **resulting in *T''* with lower cost than *T***
- **Therefore, building an optimal tree can start with greedy choice of merging together two characters of lowest frequency**

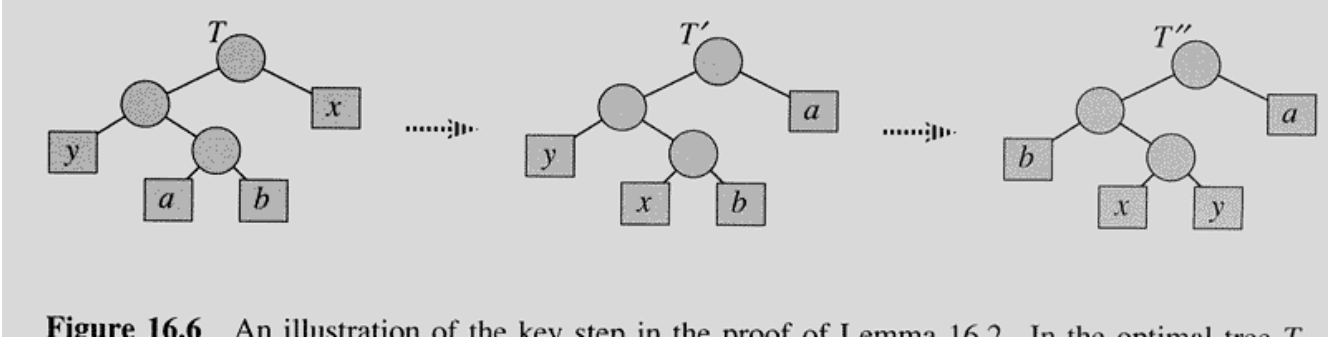
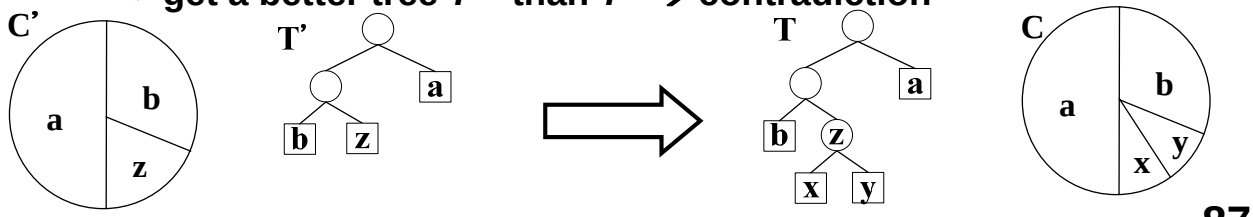


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree *T*

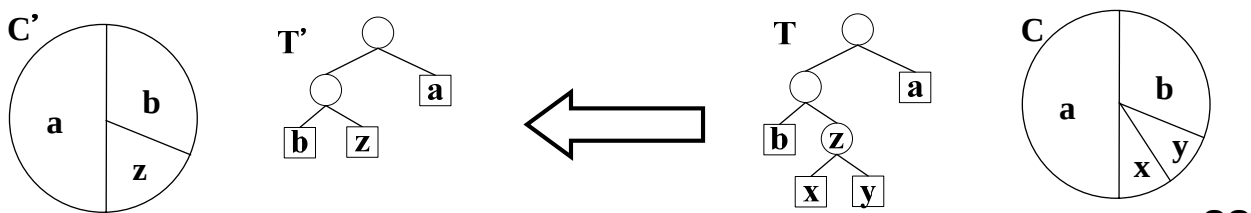
Optimal Substructure Lemma 16.3 V1

- Tree *T* is the tree representing code over *C*
  - *z* is parent of characters *x* and *y* of minimum frequency
    - *z.freq =x.freq+y.freq*
- Tree *T'* is a tree representing code over *C'*
    - *C' = C - {x, y} ∪{z}*
- If tree *T'* represents optimal prefix code for *C'*,
  - then tree *T* also represents optimal prefix code *C*
- Proof: cut and paste
  - if there is a better tree *T''* than *T* *i.e. B(T'') < B(T)*
  - both *T''* and *T* have leaves *x* and *y* as siblings (lemma 16.2)
  - we can replace *x* and *y* by *z* in *T''*
    - get a better tree *T'''* than *T'* → contradiction



Optimal Substructure Lemma 16.3 V2

- Tree *T* is the tree representing code over *C*
  - *z* is parent of *x* and *y* of minimum frequency
    - *z.freq = x.freq+y.freq*
- Tree *T'* is a tree representing code over *C'*
    - *C'= C - {x, y} ∪{z}*
- If tree *T* represents optimal prefix code for *C*,
  - then tree *T'* also represents optimal prefix code *C'*
- Proof: cut and paste
  - if there is a better tree *T''* than *T'* *i.e. B(T'') < B(T')*
  - then we add leaves *x* and *y* under *z* in *T''*
    - get a better tree *T'''* than *T* → contradiction



Finally

- Theorem 16.4: HUFFMAN produces an optimal prefix code
  - results from lemma 16.2 and 16.3

- Huffman coding today is often used as back-end to other compression methods
  - PKZIP, JPEG and MP3 uses Huffman coding
- Huffman coding beat his teacher's method, *Shannon-Fano coding*
  - you can beat your teachers too!

Reading

- CH15
- CH16