Design and Analysis of Algorithms

Review

西安电子科技大学
计算机科学与技术学院
刘 惠

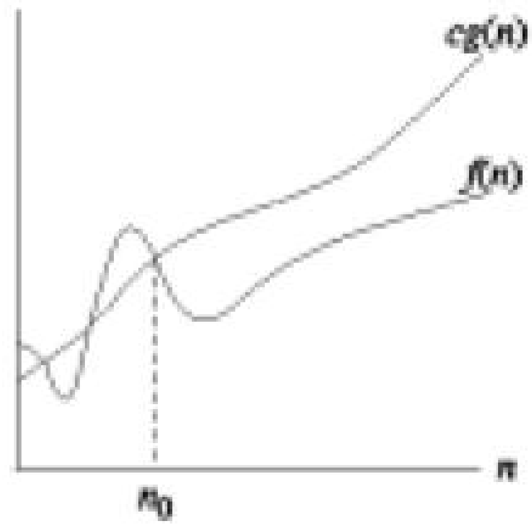
Analysis of Algorithms

O-notation

- $O(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

  --$O(.)$ is used to asymptotically upper bound a function.

  --$O(.)$ is used to bound worst-case running time.



O-notation

- **Examples:**

--$1/3n^2 - 3n \in O(n^2)$ **because** $1/3n^2 - 3n \leq cn^2$ **if** $c \geq 1/3 - 3/n$ **which holds for** $c = 1/3$ **and** $n > 9$

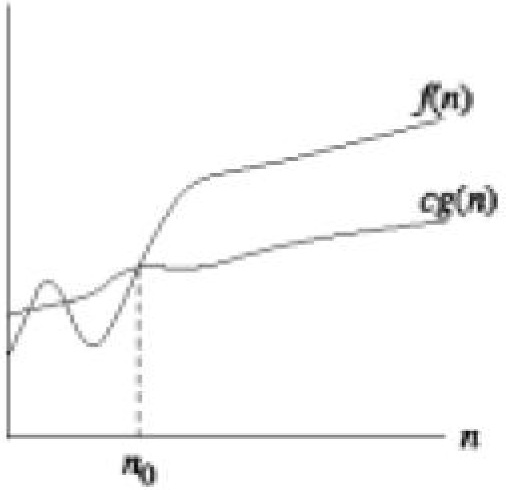
--$k_1n^2 + k_2n + k_3 \in O(n^2)$ **because** $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^2$ **and for** $c > k_1 + |k_2| + |k_3|$ **and** $n \geq 1$, $k_1n^2 + k_2n + k_3 \leq cn^2$

--$k_1n^2 + k_2n + k_3 \in O(n^3)$ **as** $k_1n^2 + k_2n + k_3 \leq (k_1 + |k_2| + |k_3|)n^3$ **(upper bound)**

Ω-notation

- $\Omega(g(n)) = \{f(n):$ **There exist positive constants** c **and** n_0 **such that** $0 \le cg(n) \le f(n)$ **for all** $n \ge n_0\}$

  --We use Ω**-notation to give a lower bound on a function.**



Ω-notation

- **Examples:**

--$1/3n^2 - 3n \in \Omega(n^2)$ **because** $1/3n^2 - 3n \geq cn^2$ **if** $c \leq 1/3-3/n$ **which holds for** $c = 1/6$ **and** $n > 18$ 【$c = 1/3$ **and** $n > 9$】

--$k_1n^2+k_2n+k_3 \in \Omega(n^2)$

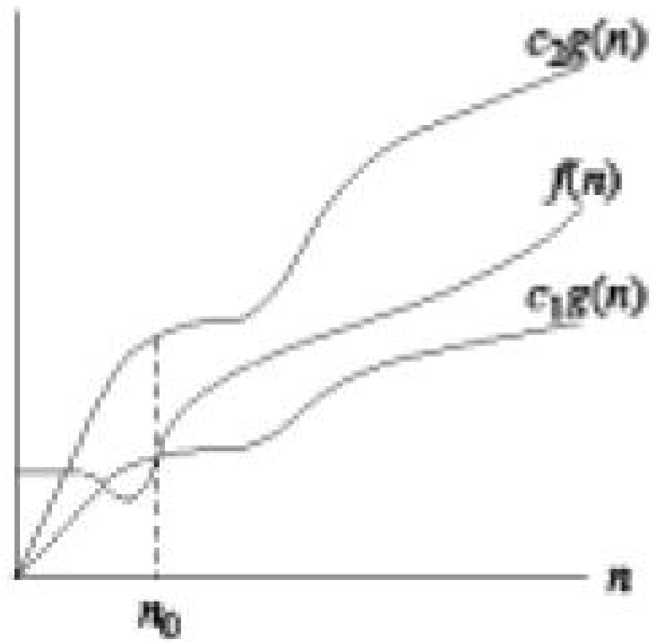
--$k_1n^2+k_2n+k_3 \in \Omega(n)$ **(lower bound)**

Θ-notation

- $\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

  --We use Θ-notation to give a tight bound on a function.

  -- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$



Θ-notation

- **Examples:**

-- $k_1n^2+k_2n+k_3 \in \Theta(n^2)$

--The worst case running time of insertion-sort is $\Theta(n^2)$

--$6nlgn + \sqrt{n}lg^2n=\Theta(nlgn)$

>>We need to find $c_1, c_2, n_0 > 0$ **such that** $c_1nlgn \le 6nlgn + \sqrt{n}lg^2n \le c_2nlgn$ **for** $n \ge n_0$**.**

>> $c_1nlgn \le 6nlgn + \sqrt{n}lg^2n \rightarrow c_1 \le 6 + lgn/\sqrt{n}$**, which is true if we choose** $c_1 = 6$ **and** $n_0 = 1$**.** $6nlgn + \sqrt{n}lg^2n \le c_2nlgn \rightarrow 6 + lgn/\sqrt{n} \le c_2$**, which is true if we choose** $c_2 = 7$ **and** $n_0 = 2$**. This is because** $\underline{lgn \le \sqrt{n}}$ **if** $n \ge 2$**. So** $c_1 = 6, c_2 = 7$ **and** $n_0 = 2$ **works.**

1. $\frac{1}{3}n^2-3n\in O(n^2)$

证: $f(n)=\frac{1}{3}n^2-3n$

$g(n)=n^2$

由O定义,要求

$0\le \frac{1}{3}n^2-3n\le cn^2$

$\frac{1}{3}n^2-3n\ge 0 \Rightarrow \frac{1}{3}n\ge 3 \Rightarrow \cancel{n\ge 9}\quad n>9$?

$\frac{1}{3}n^2-3n\le cn^2 \Rightarrow \frac{1}{3}-\frac{3}{n}\le c \Rightarrow c\ge\frac{1}{3}\Rightarrow c=\frac{1}{3}$

2. $k_1n^2+k_2n+k_3\in O(n^2)$

证: ∵ $k_1n^2+k_2n+k_3\le (k_1+|k_2|+|k_3|)n^2\le cn^2$

$c\ge k_1+|k_2|+|k_3|$ 且 $n\ge 1$.

3. $k_1n^2+k_2n+k_3\in O(n^3)$

证: 同上.

4. $\frac{1}{3}n^2-3n\in \Omega(n^2)$

证: $\frac{1}{3}n^2-3n\ge cn^2$

$\frac{1}{3}-\frac{3}{n}\ge c \Rightarrow c=\frac{1}{3},\ n>9$?

5. $6n\lg n+\sqrt{n}\lg^2 n=\Theta(n\lg n)$

证: $c_1n\lg n\le 6n\lg n+\sqrt{n}\lg^2 n\le c_2n\lg n$ 且 $n\ge n_0$, $c_1>0, c_2>0$

① $c_1n\lg n\le 6n\lg n+\sqrt{n}\lg^2 n$

$\Rightarrow c_1\le 6+\frac{\lg n}{\sqrt{n}}$

$\Rightarrow$ 可以选 $c_1=6,\ n_0=1$

② $6n\lg n+\sqrt{n}\lg^2 n\le c_2 n\lg n$

$\Rightarrow 6+\frac{\lg n}{\sqrt{n}}\le c_2$

∵ $\frac{\lg n}{\sqrt{n}}\le 1$

$\Rightarrow c_2=7,\ n_0=2$

$\frac{\lg 2}{\sqrt{2}}=\frac{0.693}{1.414}=0.49$

综合①和②得

$c_1=6\quad c_2=7\quad n_0=2$

Analysis of insertion sort

INSERT-SORT (A)		cost	times
1	for j ← 2 to length[A]	c_1	n
2	do key ← A[j]	c_2	$n-1$
3	▷ Insert A[j] into the sorted sequence A[1..j - 1]	0	$n-1$
4	i ← j – 1	c_4	$n-1$
5	while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6	do A[i + 1] ← A[i] ▷ move item back	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i + 1] ← key ▷ find the insertion position	c_8	$n-1$

***t_j* : the number of times the while loop test in line 5 is executed for the *j* value.**

Proof by Induction

- **Claim. $T(n)=n\log_2 n$ (when n is a power of 2).**

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ \underbrace{2T(n/2)}_{\text{Sorting both halves}}+\underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise}\end{cases}$$

$$\begin{aligned}T(2n)&=2T(n)+2n\\ &=2n\log_2 n+2n\\ &=2n(\log_2(2n)-1)+2n\\ &=2n\log_2(2n)\end{aligned}$$

- **Proof. (by induction on n)**

--Base case: $n = 1$.

--Inductive hypothesis: $T(n) = n\log_2 n$.

--Goal: show that $T(2n) = 2n\log_2(2n)$

Example of substitution

- $T(n) = 4T(n/2) + 100n$
  $\le 4c(n/2)^3 + 100n$
  $= (c/2)n^3 + 100n$
  $= cn^3 - ((c/2)n^3 - 100n)$ desired-residual
  $\le cn^3$ desired

- Whenever $(c/2)n^3 - 100n \ge 0$, for example, if $c \ge 200$ and $n \ge 1$

residual

Make a good guess

Changing Variables

- Use algebraic manipulation to make an unknown recurrence similar to what you have seen before.

-- Consider $T(n) = 2T(\sqrt{n}) + \lg n$,

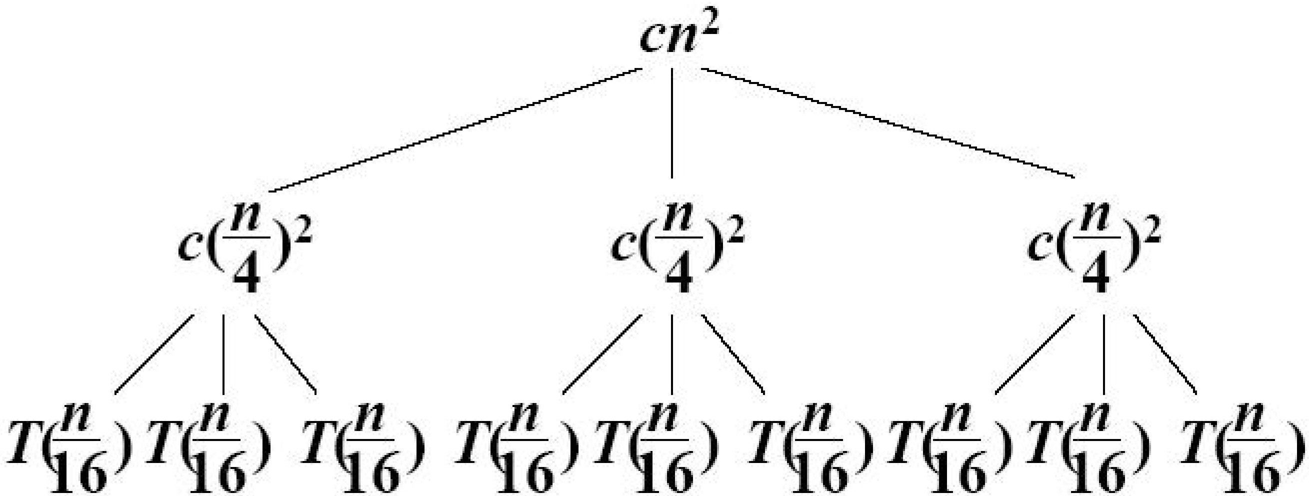
-- Rename $m = \lg n$ and we have
$T(2^m) = 2T(2^{m/2}) + m$.

-- Set $S(m) = T(2^m)$ and we have
$S(m) = 2S(m/2) + m \rightarrow S(m) = O(m\lg m)$

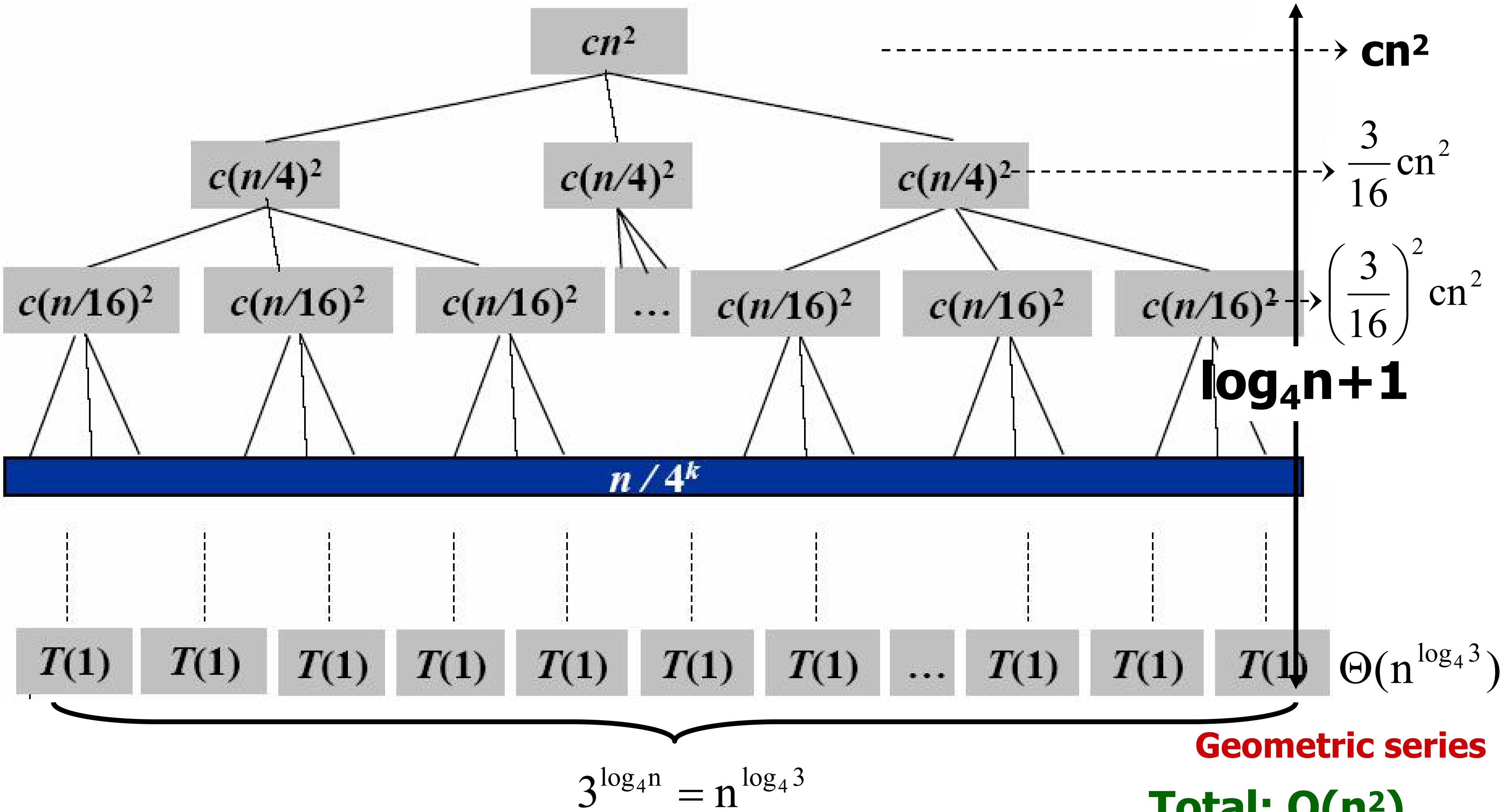
-- Changing back from $S(m)$ to $T(n)$, we have
$T(n) = T(2^m) = S(m) = O(m\lg m) = O(\lg n \lg\lg n)$

The Construction of a Recursion Tree

- Solve $T(n) = 3T(n/4) + \Theta(n^2)$, we have



Construction of Recursion Tree



- The fully expanded tree has $\lg_4 n+1$ levels, i.e., it has height $\lg_4 n$

3. Master Method

- It provides a "cookbook" method for solving recurrences of the form:

$$T(n) = a\,T(n/b) + f(n)$$

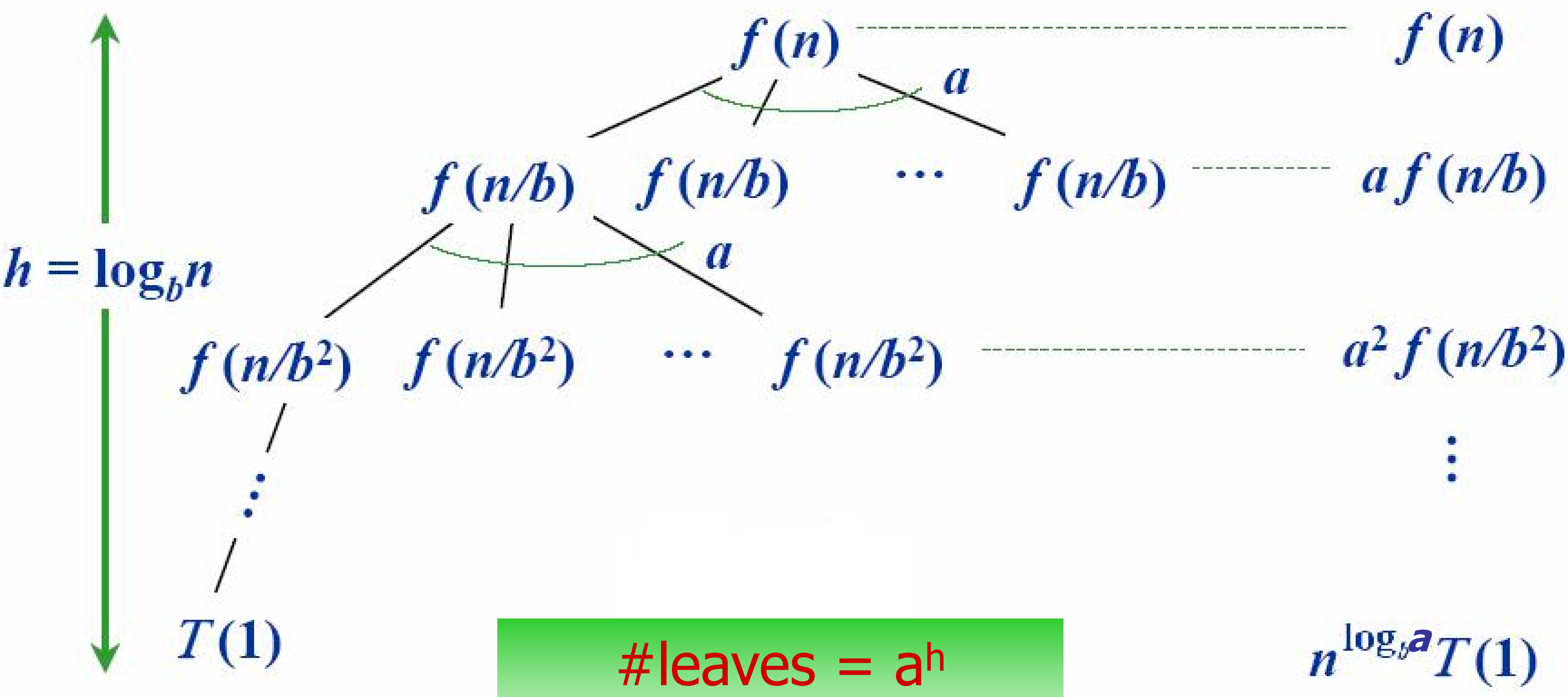
Where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

$$T(n) = cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n}\right) + n^{\log_4 3}$$

$$= \underbrace{cn^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{\infty}\right)}_{\text{等比级数收敛.}} + n^{\log_4 3}$$

Idea of master theorem

- Recursion tree



$$
\begin{aligned}
\#\text{leaves} &= a^h \\
&= a^{\log_b n} \\
&= n^{\log_b a}
\end{aligned}
$$

Three common cases

- Compare f(n) with $n^{\log_b a}$:
  - 1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
    - » f(n) grows polynomially slower than $n^{\log_b a}$ (by an n^ε factor),
    - » Solution: $T(n) = \Theta(n^{\log_b a})$
  - 2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$
    - » f(n) and $n^{\log_b a}$ grow at similar rates,
    - » Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
  - 3. $f(n) = \Omega((n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$
    - » f(n) grows polynomially faster than $n^{\log_b a}$ (by an n^ε factor),
    - » and f(n) satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1$
    - » Solution: $T(n) = \Theta(f(n))$

Design of Algorithms

Sort

Divide - and - Conquer

- **To solve P:**

-- Divide P into smaller problems P_1, P_2, …, P_k.

-- Conquer by solving the (smaller) subproblems recursively.

-- Combine the solutions to P_1, P_2, …, P_k into the solution for P.

Insertion Sort

```
INSERT-SORT (A)
for j ← 2 to length[A]
    do key ← A[j]
        ▷ Insert A[j] into the sorted sequence A[1..j - 1]
        i ← j – 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]   ▷ move item back
               i ← i – 1
        A[i + 1] ← key    ▷ find the insertion position
```

Merge-Sort (A, p, r)

- **INPUT**: **a sequence of n numbers stored in array A**
- **OUTPUT**: **an ordered sequence of n numbers**

MERGE-SORT (A, p, r)

if p < r

then q ← $\lfloor (p+r)/2 \rfloor$

MERGE-SORT(A, p, q)

MERGE-SORT(A, q + 1, r)

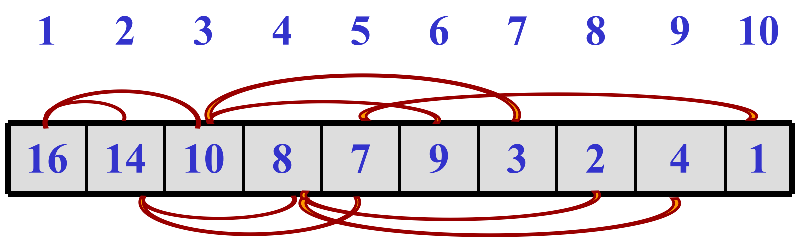
MERGE(A, p, q, r)

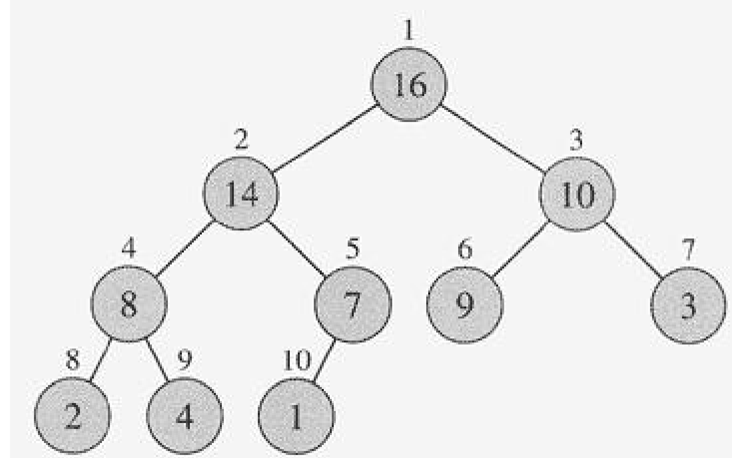
Merge (A, p, q, r)

```
MERGE (A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
create arrays L[1..n1 + 1] and R[1..n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i - 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
                then A[k] ← L[i]
                     i ← i + 1
                else A[k] ← R[j]
                     j ← j + 1
```

Heap Sort

- Viewed as a binary tree, it is completely filled on all levels except possibly the last.
- *PARENT(i) =* $\lfloor i/2 \rfloor$ *, LEFT(i) = 2i*, and *RIGHT(i) = 2i + 1*.





(a)

Quick Sort

QUICKSORT

```
QUICKSORT(A, p, r)
if p < r
  then q ← PARTITION( A, p, r)
          QUICKSORT(A, p, q-1)
          QUICKSORT(A, q+1, r)
```

Initial call Quicksort(A, 1, n)

Lower Bound for decision-tree Sorting

Theorem. Any comparison sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case

Proof. Worst case dictated by tree height h.

--$n!$ different orderings.

--One (or more) leaves corresponding to each ordering.

--Binary tree with $n!$ leaves must have height

$$
\begin{aligned}
\therefore h &\geq \lg (n!) && \text{lg is mono. increasing} \\
&\geq \lg (n/e)^n && \text{Stirling's formula} \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n)
\end{aligned}
$$

Counting Sort

- **Counting sort: No comparisons between elements.**
  - **Input:** ***A[1..n]***, **where** ***A[j]*** $\in$***{1,2,…,k}***
  - **Output:** ***B[1..n],*** **sorted.**
  - **Auxiliary storage:** ***C[1..k].***
- **For** ***x*** **in A, if there are** ***17*** **elements less than** ***x*** **in A, then** ***x*** **belongs in output position** ***18*****.**
- **How if several elements in A have the same value?**
  - **Put the** ***1st*** **in position** ***18*****,** ***2nd*** **in position** ***19*****,*****3rd*** **in position** ***20*****,…**
- **How if there are** ***17*** **elements not greater than** ***x*** **in A?**
  - **Put the** ***last one*** **in position** ***17*****, the** ***penultimate one*** **in position** ***16*****,…**

Counting-sort example

COUNTING SORT

```
COUNTING-SORT(A, B, k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]]+1
//C[i] now contains the number of elements equal to i.
for i ← 2 to k
    do C[i] ← C[i]+ C[i-1]
//C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]]-1
```

A:

1	2	3	4	5
4	1	3	4	3

C:

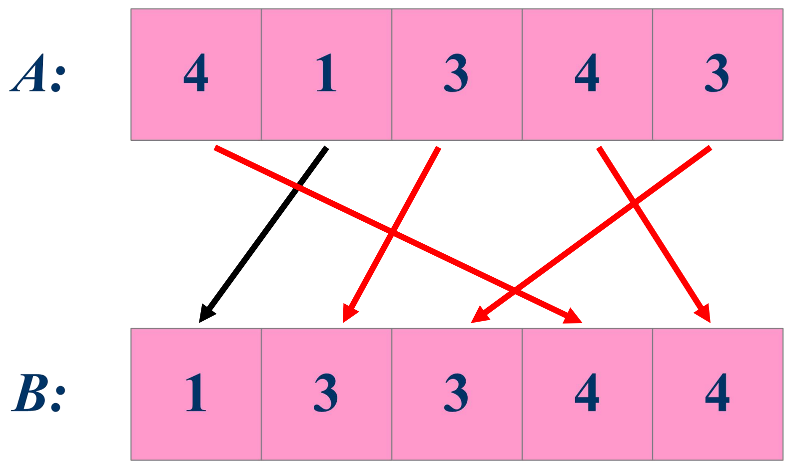
1	2	3	4
1	1	3	5

B:

1	2	3	4	5
		3		

Stable sorting

- Counting sort is a stable sort: it preserves the input order among equal elements.

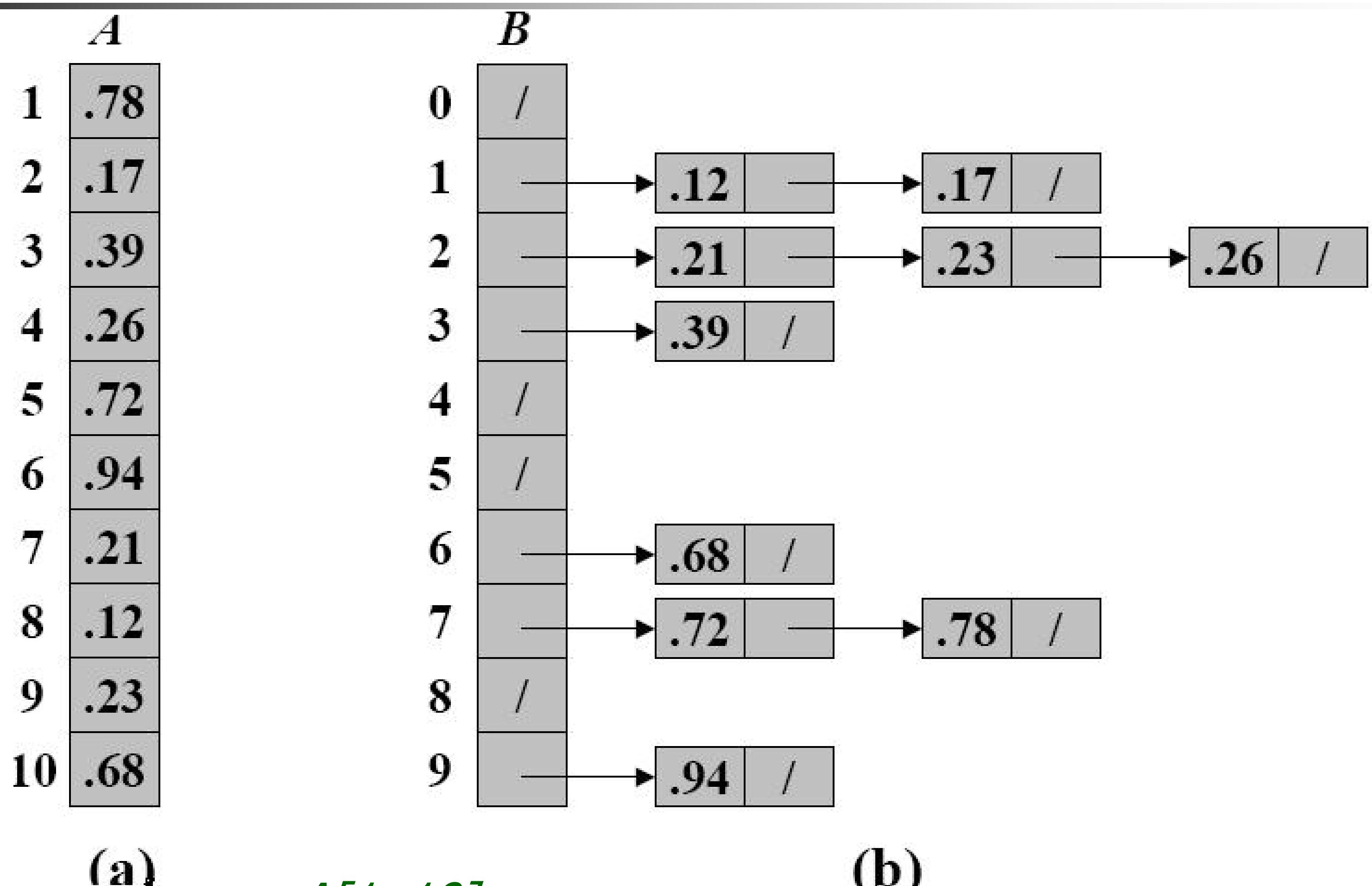


- Exercise: Where other sorts have this property?

radix sort

3 2 9		7 2 0		7 2 0		3 2 9
4 5 7		3 5 5		3 2 9		3 5 5
6 5 7		4 3 6		4 3 6		4 3 6
8 3 9	Stable sort	4 5 7	Stable sort	8 3 9	Stable sort	4 5 7
4 3 6		6 5 7		3 5 5		6 5 7
7 2 0		3 2 9		4 5 7		7 2 0
3 5 5		8 3 9		6 5 7		8 3 9

Bucket Sort



(a) The input array *A[1..10]*
(b) The array *B[0..9]* of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval *[i / 10, (i + 1) /10)*

Summary of sorting algorithms

Sorting methods	Worst Case	Best Case	Average Case	Application
Insert Sort	n^2	n	n^2	Very fast when n<50
Bubble Sort	n^2	n	n^2	Very fast when n<50
Merge Sort	n lgn	n lgn	n lgn	Need extra space; good for external sort
Heap Sort	n lgn	n lgn	n lgn	Good for real-time app.
Quick Sort	n^2	n lgn	n lgn	Practical and fast
Counting Sort	k+n	k+n	k+n	Small, fixed range; Need extra space
Radix Sort	d(k+n)	d(k+n)	d(k+n)	Fixed range; Need extra space
Bucket Sort	n	n	n	Uniform distribution

- Which in place?
- Which stable?

Summary of sorting algorithms

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

Design of Algorithms

Dynamic Programming

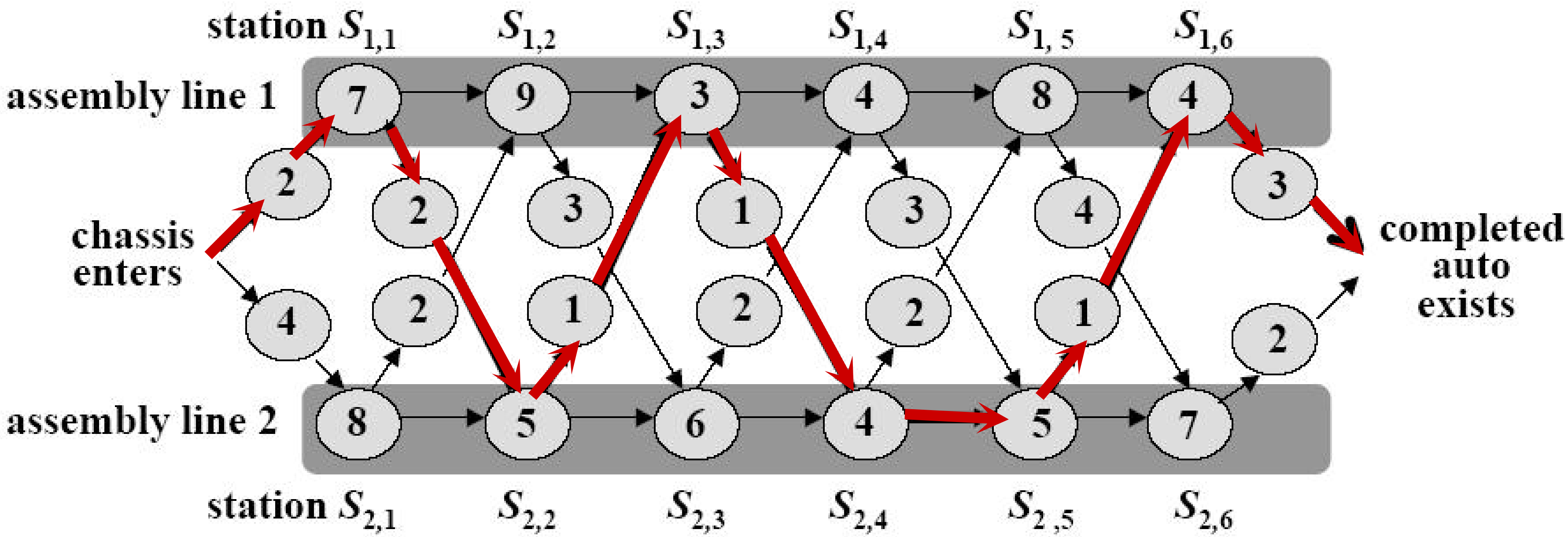
Optimization Problems

- A design technique, like divide-and-conquer.
- Works bottom-up rather than top-down.
- Useful for optimization problems.
- Four-step method:

1. Characterize the structure of the optimal solution.
2. Recursively define the value of the optimal solution.
3. Compute the value of the solution in a bottom-up fashion.
4. Construct the optimal solution using the computed information.

Construct an optimal solution

Can we avoid some computations?



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

$l^* = 1$

Recursive Formula

- Let ***$f_i[j]$*** denote the fastest possible time (which is the values of optimal solution, optimal substructure) to get the chassis through ***$S_{i,j}$***
- Have the following formulas:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

Using symmetric reasoning, we can get the fastest way through station $S_{2,j}$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

- Total time:

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

Matrix-Chain Multiplication

- We would like to find the split that uses the minimum number of multiplications. Thus,

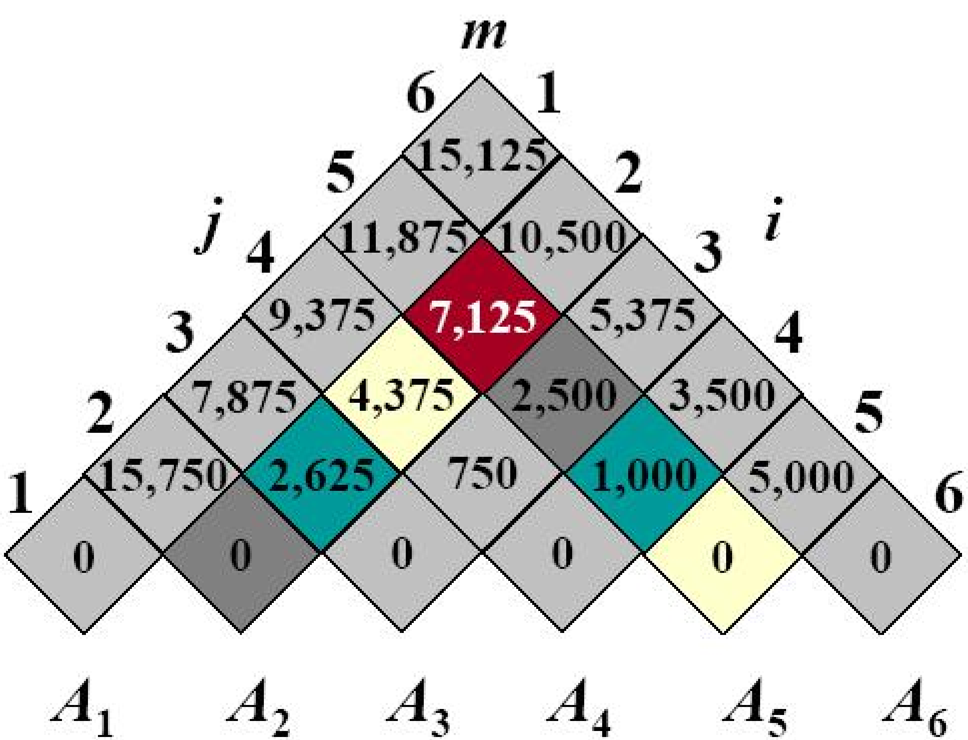
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

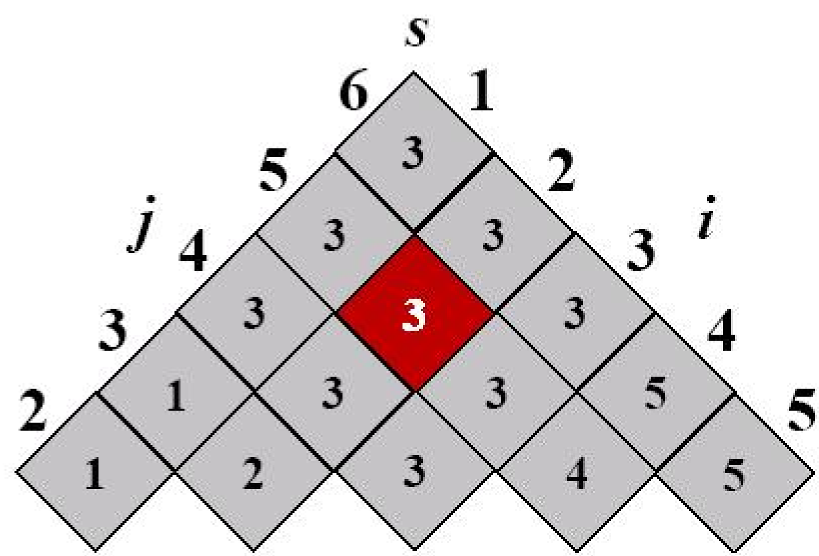
  - $m[i, k]$ = optimal cost for $A_i \times \ldots \times A_k$
  - $m[k+1, j]$ = optimal cost for $A_{k+1} \times \ldots \times A_j$
  - $p_{i-1} p_k p_j$ = cost for $(A_i \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j)$

- To obtain the actual parenthesization, keep track of the optimal ***k*** for each pair ***(i,j)*** as ***s[i,j]***.

Example: DP for CMM

- The optimal solution is $((A_1(A_2A_3))((A_4A_5)A_6))$





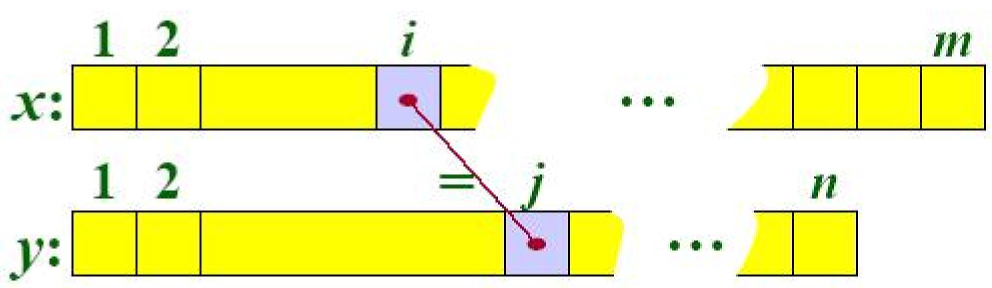
matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

LCS

- Theorem.

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

- Proof. Case *x[i] = y[j]*:



- Let *z[1..k] = LCS(x[1..i],y[1..j])*, where *c[i,j] = k*. Then *z[k]=x[i],* or else *z* could be extended. Thus, *z[1..k-1]* = is *CS* of *x[1..i-1]* and *y[1..j-1]*

Computing the length of an LCS

- The sequences are X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>

- Compute *c[i,j]* row by row for *i=1..m*, *j=1..n*. Time=*Θ(mn)*

- Reconstruct LCS by tracing backwards. Space = *Θ(mn)*.

	j	0	1	2	3	4	5	6
i		y_i	*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

DP for 0-1 Knapsack Problem

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i > w \\ \max(c[i-1, w - w_i] + v_i,\ c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

DYNAMIC PROGRAMMING FOR 0/1 KNAPSACK

```
KNAPSACK-DP(v, w, n, W)
for w ← 0 to W
   do c[0,w] = 0
for i ← 1 to n
   do c[i,0] ← 0
      for w ← 1 to W
          do if w[i] ≤ w
                 then if v[i] + c[i-1,w-w[i]] > c[i-1,w]
                          then c[i,w] ← v[i] + c[i-1,w-w[i]]
                          else c[i,w] ← c[i-1,w]
                 else c[i,w] ← c[i-1,w]
```

Design of Algorithms

Greedy

Greedy Method

- **For many optimization problem, Dynamic Programming is overkill. A greedy algorithm always make the choice that looks best at every step. That is, it makes local optimal solution in the hope that this choice will lead to a globally optimal one.**
  - **I make the shortest path to the target at each step. Sometime I win, sometime I lose.**

Fractional Knapsack

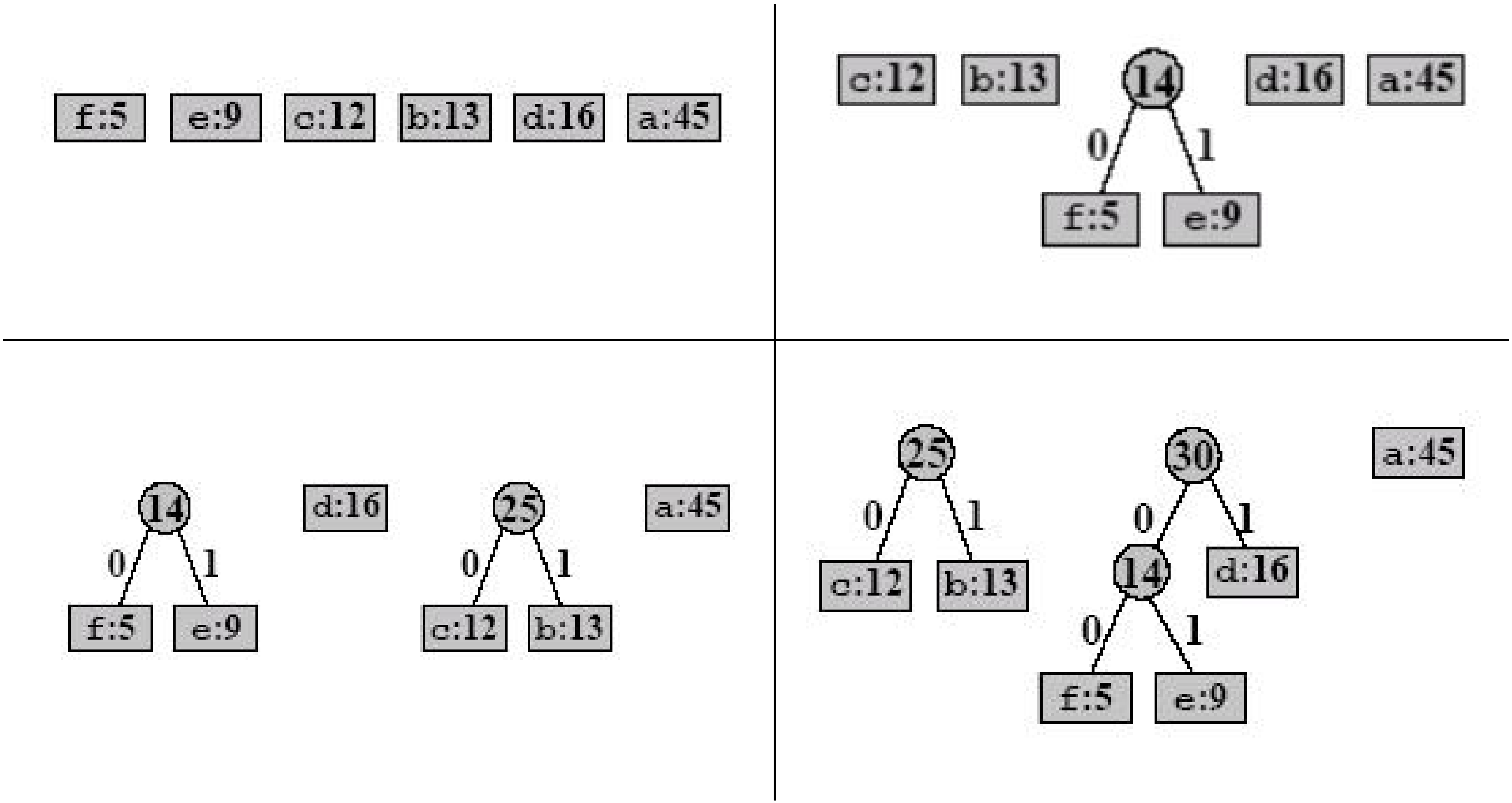
- There are 5 items that have a value and weight list below, the knapsack can contain at most 100 Lbs.

value($US)	20	30	65	40	60
weight(Lbs)	10	20	30	40	50
value/weight	2	1.5	2.1	1	1.2

- Method 1 choose the least weight first
  - Total Weight = 10 + 20 + 30 + 40 = 100
  - Total Value = 20 + 30 + 65 + 40 = 155
- Method 2 choose the most expensive first
  - Total Weight = 30 + 50 + 20 = 100
  - Total Value = 65 + 60 + 20 = 145
- Method 3 choose the most value/ weight per unit first
  - Total Weight = 30 + 10 + 20 + 40 = 100
  - Total Value = 65 + 20 + 30 + 48 = 163

What can you draw?

Example of Huffman codes



Design of Algorithms

Graph

Shortest paths

Single-source shortest paths

-- Nonnegative edge weights

>>Dijkstra's algorithm: O(E+VlgV)

-- General

>>Bellman-Ford: O(VE)

All-pairs shortest paths

-- Nonnegative edge weights

>> Dijkstra's algorithm |V| times: O(VE+V^2lgV)

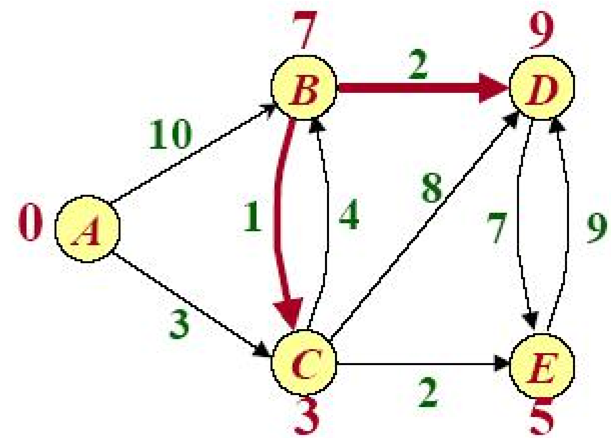
-- General

>> Three algorithms today

Floyd-Warshall algorithm

Example of Dijkstra’s algorithm

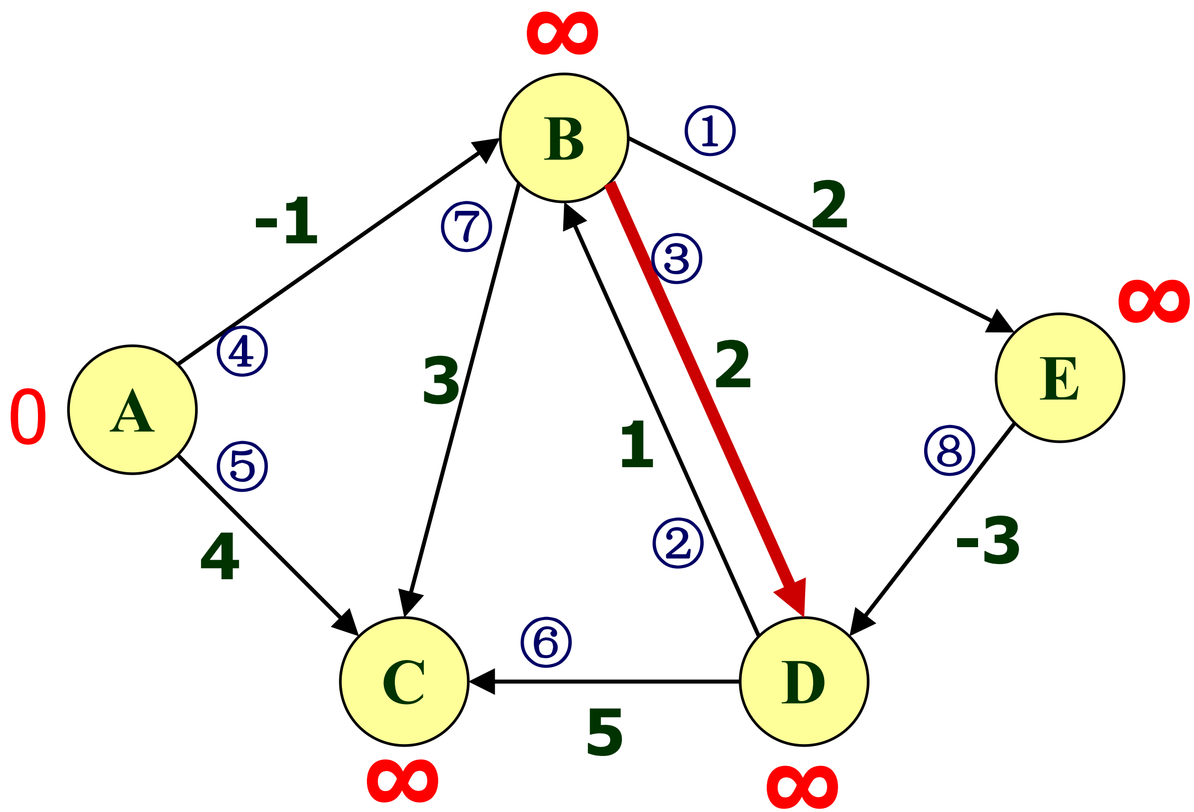
“D” ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

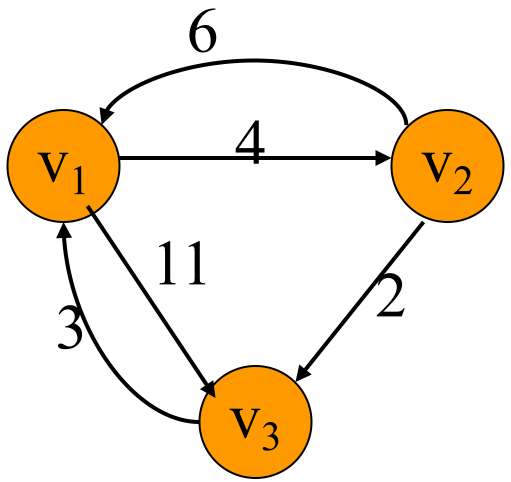
S: {A, C, E, B, D}

Example of Bellman-Ford



Floyd-Warshall algorithm

$C^k(i,j)=\min\{C^{k-1}(i,j), C^{k-1}(i,k)+C^{k-1}(k,j)\}$



$C^0 = A =$

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$C^1 =$

	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

$C^2 =$

	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

$C^3 =$

	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Johnson' algorithm

Johnson' algorithm

$h(v) \le h(u) + w(u,v)$;
$0 \le w(u,v) + h(u) - h(v)$;

```
JOHNSON(G)
1   compute G', where V[G'] = V[G] ∪ {s}
        E[G'] = E[G] ∪ {(s, v) : v∈V[G], and
        w(s,v)=0 for all v∈V[G]
2   if BELLMAN-FORD(G', w, s) = FALSE
3       then print "the input graph contains a negative-weight cycle"
4       else for each vertex v∈V[G]
5                do set h(v) to the value of δ(s, v)
                       computed by the Bellman-Ford algrithom
6            for each edge (u, v)∈E[G]
7                do w'(u, v)← w(u, v) + h(u) - h(v)
8            for each vertex u∈V[G]
9                 do run DIJKSTRA(G, w', u) to compute δ'(u, v) for all v∈V[G]
10           for each v∈V[G]
11                do d_uv ← δ'(u, v) + h(v) - h(u)
```

Design of Algorithms

others

Back-Tracking Algorithm for n-Queen problem

NQUEENS

```
NQUEENS(n)
    X(1)←0;k←1
    while k>0 do
        X(k)←X(k)+1
        while X(k) ≤ n and not PLACE(k) do
            X(k)←X(k)+1
        if X(k) ≤ n
            then  if k=n
                      then print (X)
                      else k←k+1;X(k)←0
            else k←k-1;
```

k : current row;
X(k) : column NO of queen k

Finding next valid column NO using bound function PLACE.

Found or not?

Found! Is it an answer? Print the answer, or shift to the next row and find a place for next queen.

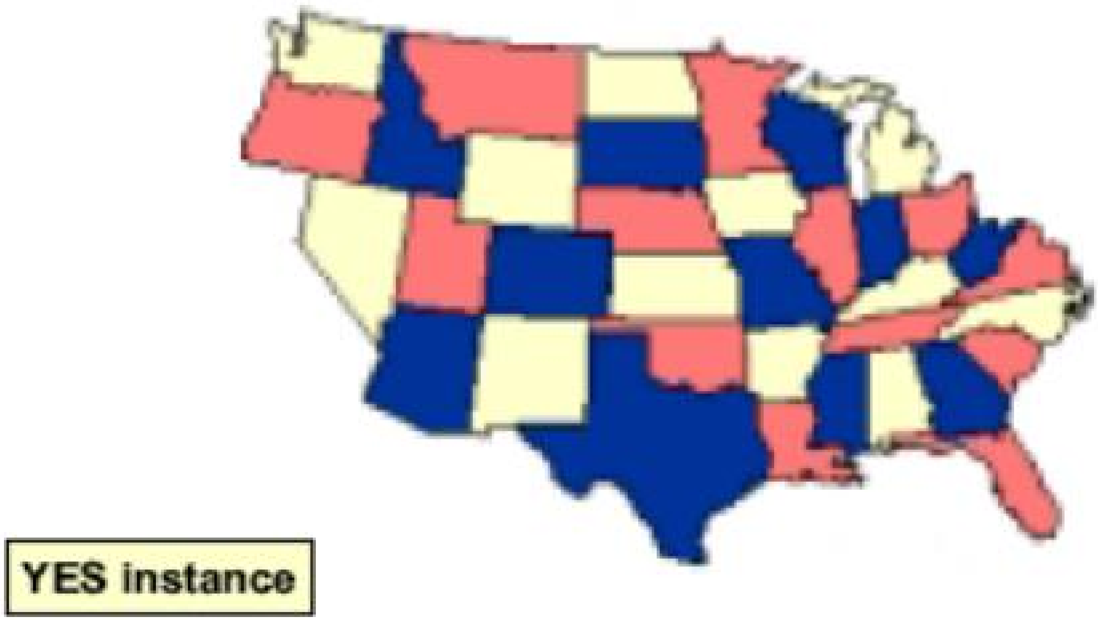
No! Tracking back.

Design of Algorithms

NP Problem

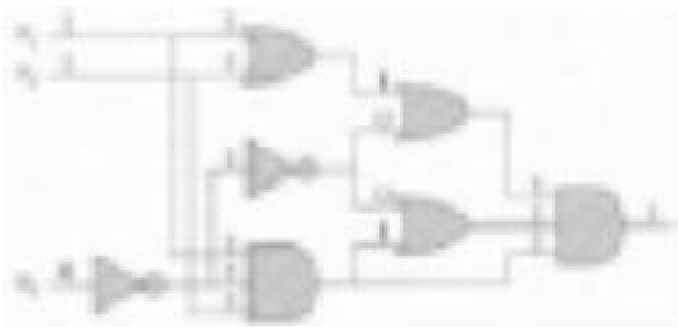
Some Hard Problems

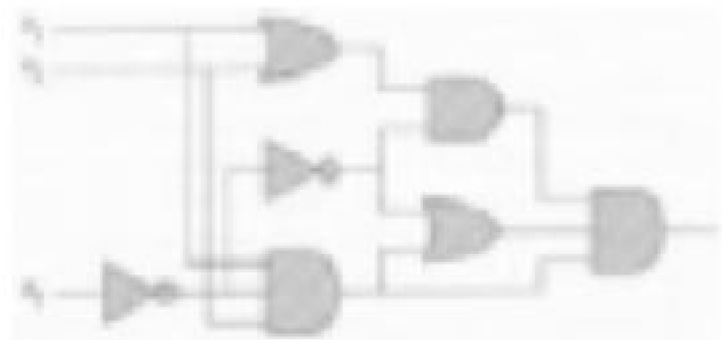
- 3-COLOR: Given a planar map, can it be colored using 3 colors so that no adjacent regions have the same color?



Some Hard Problems

- CIRCUIT-SAT: Is there a way to assign inputs to a given Boolean (combinational) circuit that makes it true?





YES instance

NO instance

Some Hard Problems

- TSP: A traveling salesperson needs to visit N cities. Is there a route of length at most D?