Algorithm Design and Analysis

David N. JANSEN, Bohua ZHAN
名 姓

算法设计与分析

詹博华，杨大卫

This week's content

- Today Wednesday:
  - Introduction of teachers
  - Grading rules
  - Why study algorithms?
    Chapter 3: Growth of functions
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 4: Divide and Conquer
    Introduction of the remaining content

这周的内容

- 今天周三：
  - 老师介绍
  - 课程评分规则
  - 为什么学习算法?
    第三章：函数的增长
  - 函数的增长的练习
- 明天周四：
  - 练习题解答
  - 第四章：分治策略
    以后内容的介绍

Exercises 练习

- 1.2-2
  Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

- 假设我们正比较插入排序与合并排序在相同机器上的实现。对规模为n的输入，插入排序运行$8n^2$步，而合并排序运行$64n \lg n$步。问对哪些n值，插入排序优于合并排序?

Solution 1.2-2

- Note that in this book $\lg n = \log_2 n$.
- Insertion sort is faster if $8n^2 < 64n \lg n \iff n < 8 \lg n$.
- From the lecture, we know: $n < 10 \lg n \iff n < 59$.
- When is $n < 8 \lg n$? Guess a few values:
  $n = 40$? $40 < 8 \cdot \lg 40 \approx 42.6$ ➨ The boundary might be approximately 42.
  $n = 45$? $45 \not< 8 \cdot \lg 45 \approx 43.9$
  $n = 42$? $42 < 8 \cdot \lg 42 \approx 43.1$ ➨ $43 < 8 \cdot \lg 42 < 8 \cdot \lg 43$
  $n = 44$? $44 \not< 8 \cdot \lg 44 \approx 43.7$
- Answer: This implementation of insertion sort is faster if and only if the input size $n \leq 43$.

Exercises 练习

Problem 1-1: Comparison of running times
For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds (μsec).

思考题 1-1: 运行时间的比较
假设求解问题的算法需要f(n)微妙，对下表中的每个函数$f(n)$和时间t，确定可以在时间t内求解的问题的最大规模n。

t = 1 sec, 1 min, 1 hour, 1 day, 1month/月, 1 year, 100 years

$f(n) = \lg n, \sqrt{n}, n, n \lg n, n^2, n^3, 2^n, n!$ ($\lg n = \log_2 n$)

Exercises 练习

	1秒	1分	1小时	1天	1月	1年	1世纪
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

If $f(n) = \sqrt{n}$ microseconds,
find n such that $f(n) = 1$ month.

Exercises 练习

	1秒	1分	1小时	1天	1月	1年	1世纪
$\lg n$							
$\sqrt{n}$							
n	10^6	$6 \cdot 10^7$	$3.6 \cdot 10^9$	$8.6 \cdot 10^{10}$	$2.6 \cdot 10^{12}$	$3.2 \cdot 10^{13}$	$3.2 \cdot 10^{15}$
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

$$2^{10^6} = 10^{\log_{10} 2^{10^6}}$$
$$= 10^{\lg 2^{10^6} / \lg 10} = 10^{10^6 / \lg 10}$$
$$\approx 10^{301030}$$

ises 练习

		1分	1小时	1天	1月	1年	1世纪
lg *n*	10^{301030}				$10^{8\cdot 10^{11}}$		
√*n*	10^{12}				$6.9\cdot 10^{24}$		
n	10^6	$6\cdot 10^7$	$3.6\cdot 10^9$	$8.6\cdot 10^{10}$	$2.6\cdot 10^{12}$	$3.2\cdot 10^{13}$	$3.2\cdot 10^{15}$
n lg *n*	$6.27\cdot 10^5$				$7.3\cdot 10^{10}$		
n^2	10^3				$1.6\cdot 10^6$		
n^3	10^2				$1.4\cdot 10^4$		
2^n	19				41	44	51
n!	9				15	16	17

$\lg 10^{12} \approx 40$;
therefore guess values
around $2.6\cdot 10^{12} / 40 \approx 6.6\cdot 10^{10}$.
Or, even better, around
$2.6\cdot 10^{12} / \lg 6.6\cdot 10^{10}$.

Exercises 练习

- 2.1-3
  Consider the searching problem:
  Input: A sequence of n numbers $A = (a_1, a_2, ..., a_n)$ and a value v.
  Output: An index i such that $v = A[i]$, or the special value NIL if v does not appear in A.
  Write pseudocode for linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

- 考虑以下查找问题：
  输入：n个数的一个序列$A = (a_1, a_2, ..., a_n)$和一个值v。
  输出：下标i使得$v = A[i]$或者当v不在A中出现时，特殊值NIL。
  写出线性查找的伪代码，它扫描整个序列来查找v。使用一个循环不变式来证明你的算法的正确性。确保你的循环不变式满足三条必要的性质。

Linear search 线性查找

- specification: find an index i in the sequence $(a_1, a_2, ..., a_n)$ such that $v = a_i$. If v is not in the sequence, return NIL.

- 需求：在序列$(a_1, a_2, ..., a_n)$里，找到下标i使得$v = a_i$。当v不在序列中出现时，特殊值NIL。

sequence $(a_1, ..., a_{i-1})$ does not contain v

check whether $A[i] = v$

sequence $(a_1, ..., a_i)$ does not contain v

SEARCH($a_1, a_2, ..., a_n$: numbers; v : number)
for $i = 1$ **to** n
    if $v == a_i$
        return i
return NIL

2.1-3

- Loop invariant: sequence (a_1, ..., a_{i-1}) does not contain v.
  - Initialisation: When the loop starts (i = 1), the invariant holds (because the sequence is empty).
  - Maintenance: If the sequence (a_1, ..., a_{i-1}) does not contain v at the beginning of iteration i, then sequence (a_1, ..., a_i) does not contain v at the end of iteration i. This is true because the test $v = a_i$ and the **return** statement ensure that the end of iteration i is only reached if $v \neq a_i$.
  - Termination: When the loop ends ($i = n$), then (a_1, ..., a_n) does not contain v.
- Prove the specification using the loop invariant termination:
  - If v is in the sequence, then "return i" outputs a correct value.
  - The loop terminates if and only if v is not in the sequence, and "return NIL" outputs a correct value.

Exercises 练习

- 3-2.3
  Prove / 证明
  - $n! = \omega(2^n)$
  - $n! = o(n^n)$
  - $\log(n!) = \Theta(n \log n)$
- You may use Stirling's approximation / 可以使用斯特林近似公式:

  $n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + \Theta(1/n))$

3.2-3

- $n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + \Theta(1/n))$ means that we can use the following two inequalities:
  - $n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + O(1/n))$,
    so there exist n_1 and c_1 such that $n! \le \sqrt{2\pi n}\,(n/e)^n\,(1 + c_1/n)$ if $n \ge n_1$.
  - $n! = \sqrt{2\pi n}\,(n/e)^n\,(1 + \Omega(1/n))$,
    so there exist n_1 and c_2 such that $\sqrt{2\pi n}\,(n/e)^n\,(1 + c_2/n) \le n!$ if $n \ge n_2$.

3.2-3

- Assume that n is large.

- $\log (n!) \leq \log [\sqrt{2\pi n}\, (n/e)^n\, (1 + c_1/n)]$
  $= \tfrac{1}{2}\log (2\pi) + \tfrac{1}{2}\log n + n \log n - n \log e + \log (1 + c_1/n)$
  $\leq 2\, n \log n,$
  so $\log (n!) = O(n \log n)$.

- Other parts of the exercise are proven similarly.

Divide and Conquer

Latin: Divide et Impera

- Julius Caesar’s maxim 凯撒的格言 for winning wars:
  divide the enemies into small groups
  and then win one small battle at a time.

- Computer scientist’s principle for solving problems:
  divide the problem into smaller subproblems
  and then solve them one at a time.

Example: Merge Sort 合并排序

How to sort a sequence with n elements:

- If $n \geq 2$ then
  - Divide the sequence to be sorted into two halves with $n/2$ elements
  - Sort the two parts independently
    (using merge sort for shorter sequences)
  - Combine the sorted parts into one sequence

recursion: ok
because $n/2 < n$

Example: Merge Sort 合并排序

How to sort a sequence with n elements:

- If $n \geq 2$ then
  - Divide the sequence to be sorted into two halves with $\leq (n+1)/2$ elements
  - Sort the two parts independently
    (using merge sort for shorter sequences)
  - Combine the sorted parts into one sequence

recursion: ok
because $(n+1)/2 < n$

Example: Merge Sort

- How do we combine two sorted sequences into one?
- The smallest element in both sequences together
  is always at the beginning of one of the two sequences.
  ➥ only need to compare the beginning of the two sequences

Example: Merge Sort

MERGE(L, n_L, R, n_R, A) // merge the sorted sequence in L (with n_L elements)
// with the sorted sequence in R (with n_R elements)
// and store the result in A

$L[n_L + 1] = \infty$; $R[n_R + 1] = \infty$ // an element that is larger than anything else

i = 1 ; j = 1

for k = 1 **to** $n_L + n_R$

    if $L[i] \leq R[j]$

        $A[k] = L[i]$

        $i = i + 1$

    else

        $A[k] = R[j]$

        $j = j + 1$

Merge Example

- Merge the sequences (2, 4, 5, 7) and (1, 2, 3, 6).

Merge Sort: Main Algorithm

MERGE-SORT(A, n_A)
if $n_A > 1$
    $q = \lfloor n_A / 2 \rfloor$
    MERGE-SORT($A[1]$ ... $A[q]$, q)
    MERGE-SORT($A[q + 1]$ ... $A[n_A]$, $n_A - q$)
    Copy $A[1]$ ... $A[q]$ to a new array L (with $q + 1$ elements)
    Copy $A[q + 1]$ ... $A[n_A]$ to a new array R (with $n_A - q + 1$ elements)
    MERGE(L, q, R, $n_A - q$, A)

Merge Sort Example

- Sort the sequence (5, 2, 4, 7, 1, 3, 2, 6) according to merge sort.

Timing Analysis

- Recursive algorithms are often analyzed using a recurrence relation 递归式 (a function where $T(n)$ is defined using the value of $T(m)$, for some $m < n$)
- Let $T(n)$ = running time of MERGE-SORT(A, n)
- $T(n) = T(\lfloor n / 2 \rfloor) + T(n - \lfloor n / 2 \rfloor) + \Theta(n) \quad \text{if } n > 1$
  $T(1) = 1$
- Solution: $T(n) = O(n \log n)$

How to solve a recurrence

Three methods:

- (Substitution method 代入法) Guess and verify
- Recursion tree method 递归树法
- Master method 主方法

Guess and verify

- Merge sort recurrence: $T(n) = T(\lfloor n / 2 \rfloor) + T(n - \lfloor n / 2 \rfloor) + \Theta(n)$ if $n > 1$
  $T(1) = 1$
- Guess: $T(n) = O(n \log n)$
- Verify: Assume that $T(n) \leq c_1 n \lg n + 1$ (for some constant $c_1 > 0$)
  and prove this claim by strong induction.
- Induction base: $T(1) = 1 \leq c_1\, 1 \lg 1 + 1 = 1$ ✔.

Guess and verify

Induction step: Assume that $T(n') \le c_1 n' \lg n' + 1$ for all $n' < n$.
We shall prove $T(n) \le c_1 n \lg n + 1$.
If $n \ge 2$ is even, $T(n) = 2T(n/2) + \Theta(n)$

$$\begin{aligned}
&\le 2c_1 n/2 \lg(n/2) + 2 + an + b, \text{ for some constants } a \text{ and } b,\\
&= c_1 n(\lg n - 1) + 2 + an + b\\
&= c_1 n \lg n + 1 - c_1 n + 1 + an + b\\
&\le c_1 n \lg n + 1 \quad \text{if} \quad c_1 \ge ½ + a + b/2.
\end{aligned}$$

$\Theta(n)$ can be bounded by $an + b$

If $n \ge 3$ is odd, $T(n) \le T((n-1)/2) + T((n+1)/2) + \Theta(n)$

$$\begin{aligned}
&\le c_1(n-1)/2 \lg((n-1)/2) + 1 + c_1(n+1)/2 \lg((n+1)/2) + 1 + an + b\\
&\le c_1(n-1)/2 (\lg n - 1) + c_1(n+1)/2 (\lg n + \lg 4 - \lg 3 - 1) + 2 + an + b\\
&= c_1 n \lg n + c_1 n (-1 + \lg ⅔)/2 + c_1(1 + \lg ⅔)/2 + 2 + an + b\\
&\le c_1 n \lg n + 1 - c_1 n (3 \lg 3 - \lg ⁴⁄₃)/6 + 1 + an + b\\
&\le c_1 n \lg n + 1 \quad \text{if} \quad c_1 \ge (2 + 6a + 2b)/(3 \lg 3 - \lg ⁴⁄₃)\\
&\qquad\qquad\qquad\qquad\qquad \approx 0.46 + 1.38a + 0.46b
\end{aligned}$$

$\lg 4 - \lg 3 - 1 = \lg(4 / 3 / 2) = \lg ⅔$

Recursion tree method

- Sketch the tree of recursive function calls
- sum (an upper bound of) the work at every level.
  Do not use $O(...)$ during this summation (to avoid mixing different function estimates).
- Example: merge sort again

Recursion tree method

$T(n) = \qquad + an + b$

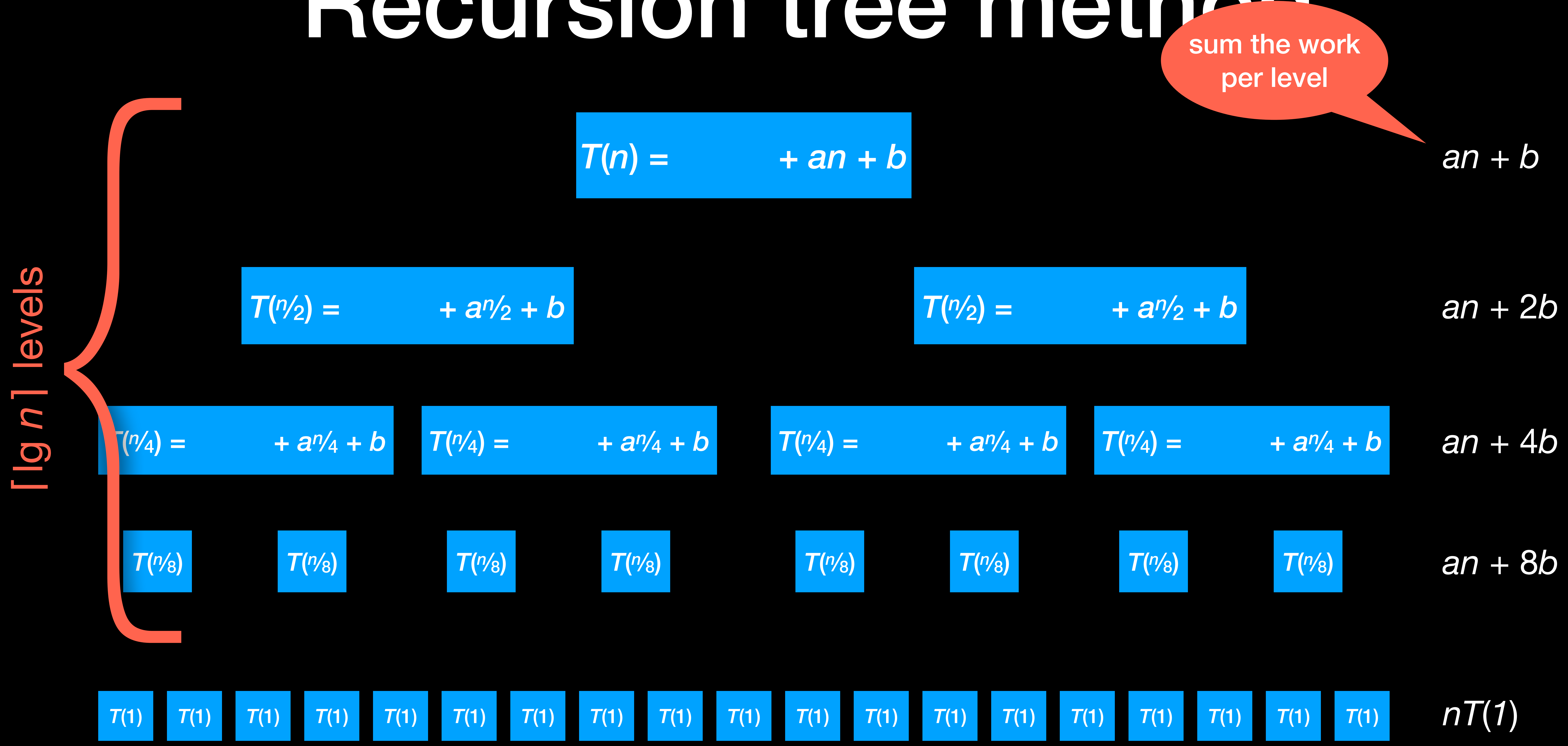
$T(n/2) = \qquad + a n/2 + b$ $\qquad$ $T(n/2) = \qquad + a n/2 + b$

$T(n/4) = \qquad + a n/4 + b$ $\qquad$ $T(n/4) = \qquad + a n/4 + b$ $\qquad$ $T(n/4) = \qquad + a n/4 + b$ $\qquad$ $T(n/4) = \qquad + a n/4 + b$

$T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$ $\quad$ $T(n/8)$

$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$

Recursion tree method



Recursion tree method

- Sketch the tree of recursive function calls
- sum (an upper bound of) the work at every level.
  Do not use $O(\ldots)$ during this summation (to avoid mixing different function estimates).
- Example: merge sort again
  Sum of the work per level:

$$\sum_{i=1}^{\lceil \lg n \rceil} an + 2^{i-1} b + nT(1) = an\lceil \lg n \rceil + b(\underbrace{2^{\lceil \lg n \rceil}}_{\approx n} - 1) + nT(1) = O(n \log n)$$

Master method

Theorem 4.1: Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined by the recurrence

$T(n) = aT(n/b) + f(n)$ (where "n/b" can mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$).

Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = O(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Example: Merge sort

- $T(n) = 2T(n/2) + \Theta(n)$, so $a = 2$ and $b = 2$.
- Choose the case: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. We have the 2nd case.
- So, $T(n) = O(n^{\log_b a} \log n) = O(n \log n)$.

Example: Strassen's Matrix Multiplication

- Assume given two $n \times n$-matrices A and B. Compute the product $C = AB$:

  $c_{ij} = \sum a_{ik} b_{kj}$

- This requires n^3 multiplications.
- Strassen found an algorithm with $\Theta(n^{\lg 7})$ multiplications ($\lg 7 \approx 2.81$).

Simple Recursive Matrix Multiplication

- If $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$, then the product can be written as:

$$AB = \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

- requires 8 multiplications of $n/2 \times n/2$-matrices and n^2 additions.
- Recurrence for timing analysis: $T(n) = 8T(n/2) + O(n^2)$

Simple Recursive Matrix Multiplication

- Recurrence for timing analysis: $T(n) = 8T(n/2) + O(n^2)$

- Master method: $a = 8$, $b = 2$
  $f(n) = O(n^2) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon = 1$, so case 1 applies.
  ➥ $T(n) = O(n^{\log_b a}) = O(n^3)$.

Strassen's recursive matrix multiplication

- Make more additions but only 7 multiplications.
- Recurrence for timing analysis: $T(n) = 7T(n/2) + O(n^2)$
- Master method: $a = 7$, $b = 2$
  $f(n) = O(n^2) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon \approx 0.2$, so case 1 applies.
  ➥ $T(n) = O(n^{\log_b a}) = O(n^{\lg 7})$.

$\lg 7 \approx 2.81$

Strassen's recursive matrix multiplication

1. Divide the input matrices A and B into $\frac{n}{2} \times \frac{n}{2}$-submatrices.

2. Create matrices $S_1, ..., S_{10}$ by adding or subtracting some of these submatrices. Requires work $O(n^2)$.

3. Recursively compute 7 product matrices $P_1, ..., P_7$ from the submatrices $A_{11}, ..., A_{22}, B_{11}, ..., B_{22}, S_1, ..., S_{10}$. Requires work $7T(n/2)$.

4. Add or subtract some of the product matrices $P_1, ..., P_7$ to calculate the parts of the product matrix AB. Requires work $O(n^2)$.

Proof Sketch of the Master Theorem

- To present the proof idea simply, assume $f(n) = n^c$ for some constant c.
- Look at the recursion tree.

Proof sketch of the master theorem

$$T(n) = aT(n/b) + n^c$$

$T(n)$ requires time n^c + recursive calls

Proof sketch of the master theorem

$T(n) = \qquad\qquad + n^c$

$T(n/b) = aT(n/b^2) + (n/b)^c$ $\qquad$ $T(n/b) = aT(n/b^2) + (n/b)^c$ $\qquad$ $T(n/b) = aT(n/b^2) + (n/b)^c$

$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}$

a times $T(n/b)$ requires time $(a/b^c)\ n^c$ + recursive calls

Proof sketch of the master theorem

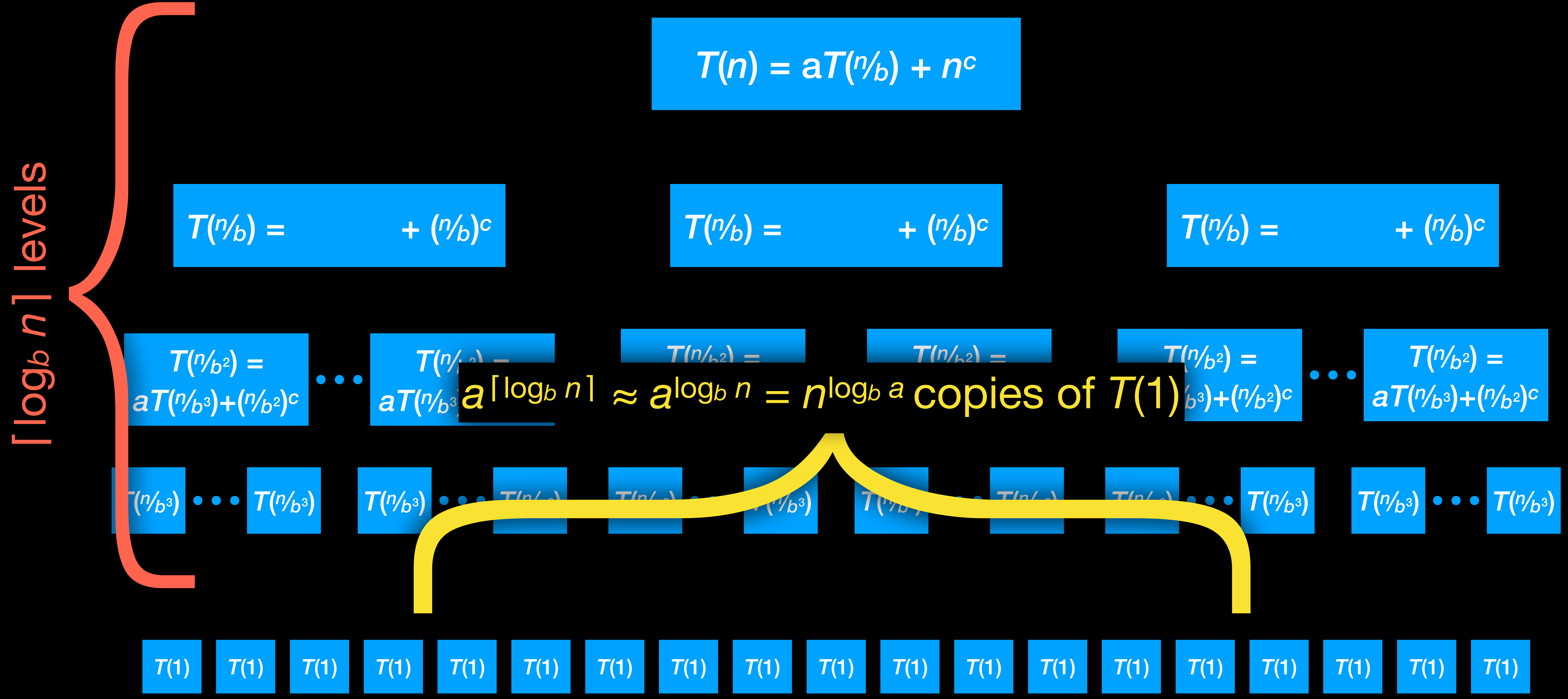
$T(n) = \qquad + n^c$

$T(n/b) = \qquad + (n/b)^c$ $\qquad$ $T(n/b) = \qquad + (n/b)^c$ $\qquad$ $T(n/b) = \qquad + (n/b)^c$

$T(n/b^2) = aT(n/b^3)+(n/b^2)^c$ $\cdots$ $T(n/b^2) = aT(n/b^3)+(n/b^2)^c$ $\qquad$ $T(n/b^2) = aT(n/b^3)+(n/b^2)^c$ $\cdots$ $T(n/b^2) = aT(n/b^3)+(n/b^2)^c$ $\qquad$ $T(n/b^2) = aT(n/b^3)+(n/b^2)^c$ $\cdots$ $T(n/b^2) = aT(n/b^3)+(n/b^2)^c$

a^2 times $T(n/b^2)$ requires time $(a/b^c)^2\, n^c$ + recursive calls

Proof sketch of the master theorem



Proof sketch of the master theorem

- Summing up the work on every level: $T(n) = O(n^{\log_b a}) + n^c \sum_{i=0}^{\lceil \log_b n \rceil} ({}^a\!/\!_{b^c})^i$

- If $n^{\log_b a} > n^c = f(n) \Leftrightarrow \log_b a > c \Leftrightarrow a > b^c \Leftrightarrow {}^a\!/\!_{b^c} > 1$, then $T(n)$ is dominated by the last summand $n^c\, ({}^a\!/\!_{b^c})^{\lceil \log_b n \rceil} \approx n^c\, ({}^a\!/\!_{b^c})^{\log_b n} = n^c\, n^{\log_b a} / n^{\log_b b^c} =$ $n^c\, n^{\log_b a} / n^c = n^{\log_b a}$, so $T(n) = O(n^{\log_b a})$.

- If $n^{\log_b a} = n^c = f(n) \Leftrightarrow \log_b a = c \Leftrightarrow a = b^c \Leftrightarrow {}^a\!/\!_{b^c} = 1$, then every summand is 1, so $T(n) = O(n^{\log_b a}) + n^c \lceil \log_b n \rceil = \Theta(n^{\log_b a} \log n)$.

- If $n^{\log_b a} < n^c = f(n) \Leftrightarrow \log_b a < c \Leftrightarrow a < b^c \Leftrightarrow {}^a\!/\!_{b^c} < 1$
then the sum is a geometric series and is bounded by $1 / (1 - {}^a\!/\!_{b^c})$,
so $T(n) = O(n^{\log_b a} + n^c / (1 - {}^a\!/\!_{b^c})) = O(n^c) = O(f(n))$.

Proof sketch of the master theorem

In general, we cannot assume $f(n) = n^c$.
Still, we can say that:

- If $f(n) = O(n^{\log_b a - \varepsilon})$,
  then there exists c_1 such that $f(n) \le c_1\, n^{\log_b a - \varepsilon}$ for large n,
  so there exists c_1' such that $T(n) = aT(n/b) + f(n) \le c_1'\, T'(n)$ for large n,
  where $T'(n) = aT'(n/b) + n^{\log_b a - \varepsilon}$.
  So $T'(n) = O(n^{\log_b a})$ and therefore $T(n) = O(n^{\log_b a})$.

- If $f(n) = \Theta(n^{\log_b a})$,
  then there exist c_1 and c_2 such that $c_1\, n^{\log_b a} \le f(n) \le c_2\, n^{\log_b a}$ for large n,
  similarly we can construct $T'(n) = aT'(n/b) + n^{\log_b a}$
  and get $T(n) = \Theta(T'(n)) = \Theta(n^{\log_b a} \log n)$.

Proof sketch of the master theorem

In general, we cannot assume $f(n) = n^c$.
Still, we can say that:

- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) \leq cf(n)$ for large n, for some constant $c < 1$, then the sum of the work on every level becomes

$$T(n) = n^{\log_b a}T(1) + \sum_{i=0}^{\lceil \log b\ n \rceil} a^i f(n/b^i) \quad \leq \quad n^{\log_b a}T(1) + \sum_{i=0}^{\lceil \log b\ n \rceil} c^i f(n)$$

$$\leq n^{\log_b a}T(1) + f(n) / (1–c) = \Theta(f(n))$$

(For small $n' = n/b^i$, it may be that $af(n'/b) \not\leq cf(n')$,
but the terms for small n' do not contribute much to $T(n)$.)

Content Overview

Growth of Functions / Divide and Conquer

- divide and conquer = solve a problem by recursively solving smaller problems of the same kind.
- Also: calculate the speed of a recursive algorithm.

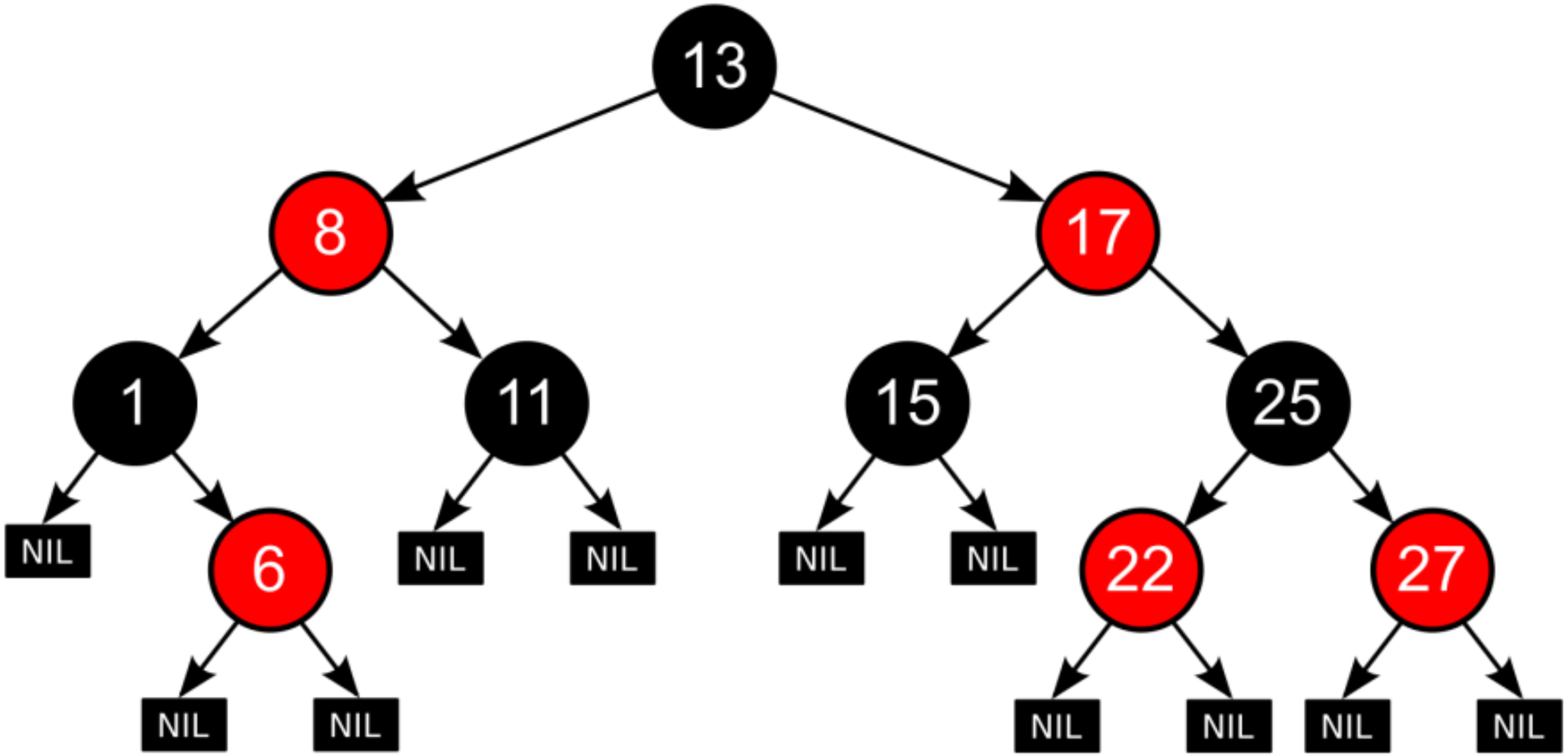
Sorting

- first large-scale use of tabulating machines (end 19th century): count census data, sort punched cards.
- census offices needed faster data analysis because population grew in many countries

Sorting

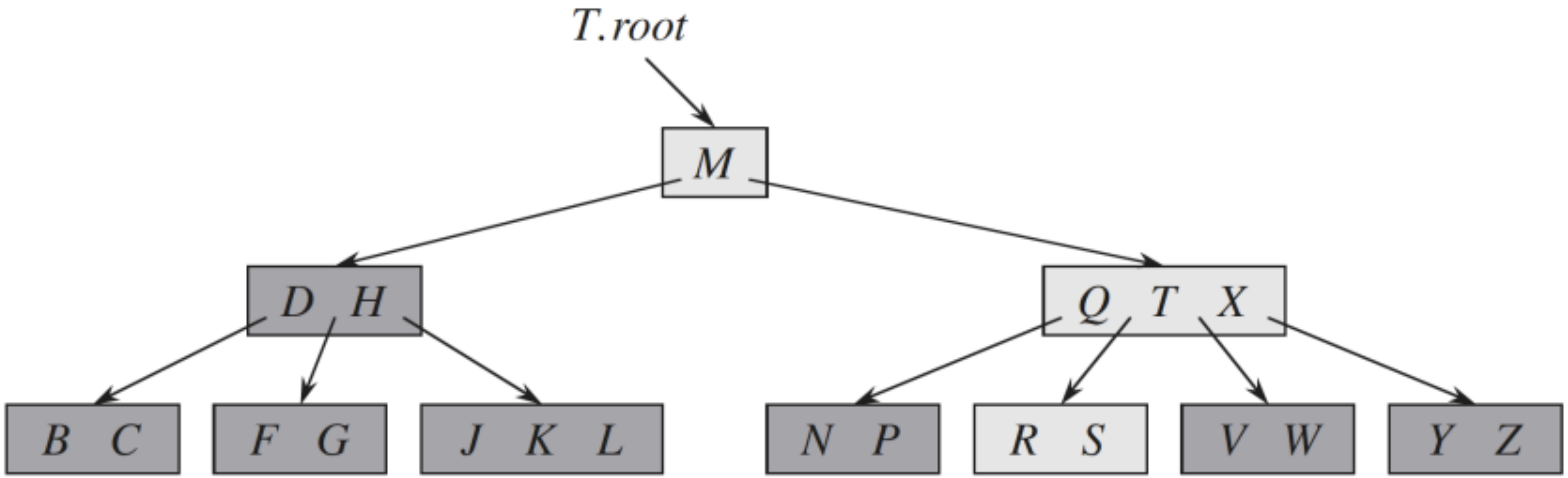
- multiple “fast” sorting algorithms: merge sort, heapsort, quicksort
- even faster algorithms for special situations: counting sort, bucket sort
- selection of the *n*th-smallest element

Red-black trees



B-trees

- B-tree where each node below the root has 2 or 3 entries.



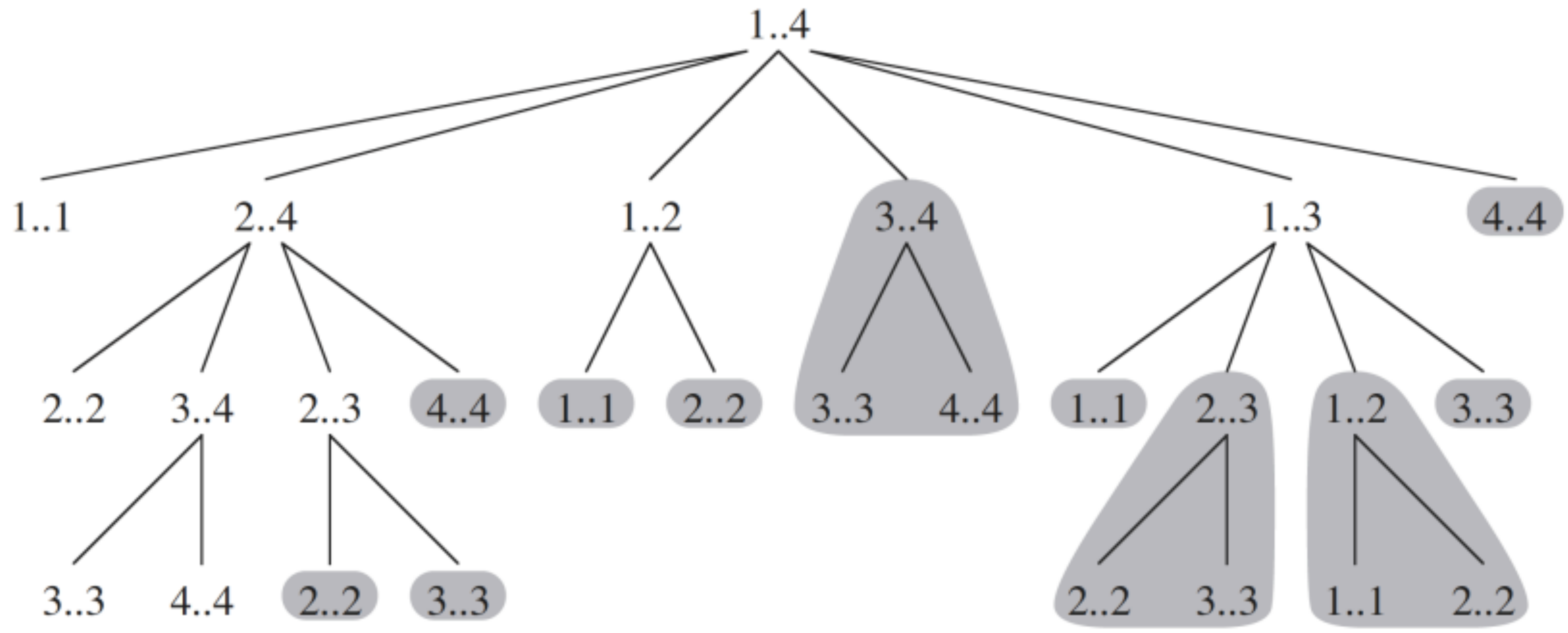
Dynamic programming

Matrix-chain multiplication: Given A_1, A_2, …, A_n, how to compute $A_1A_2\ldots A_n$ using minimum number of scalar multiplications

A_1	A_2	A_3
2×10	10×5	5×20

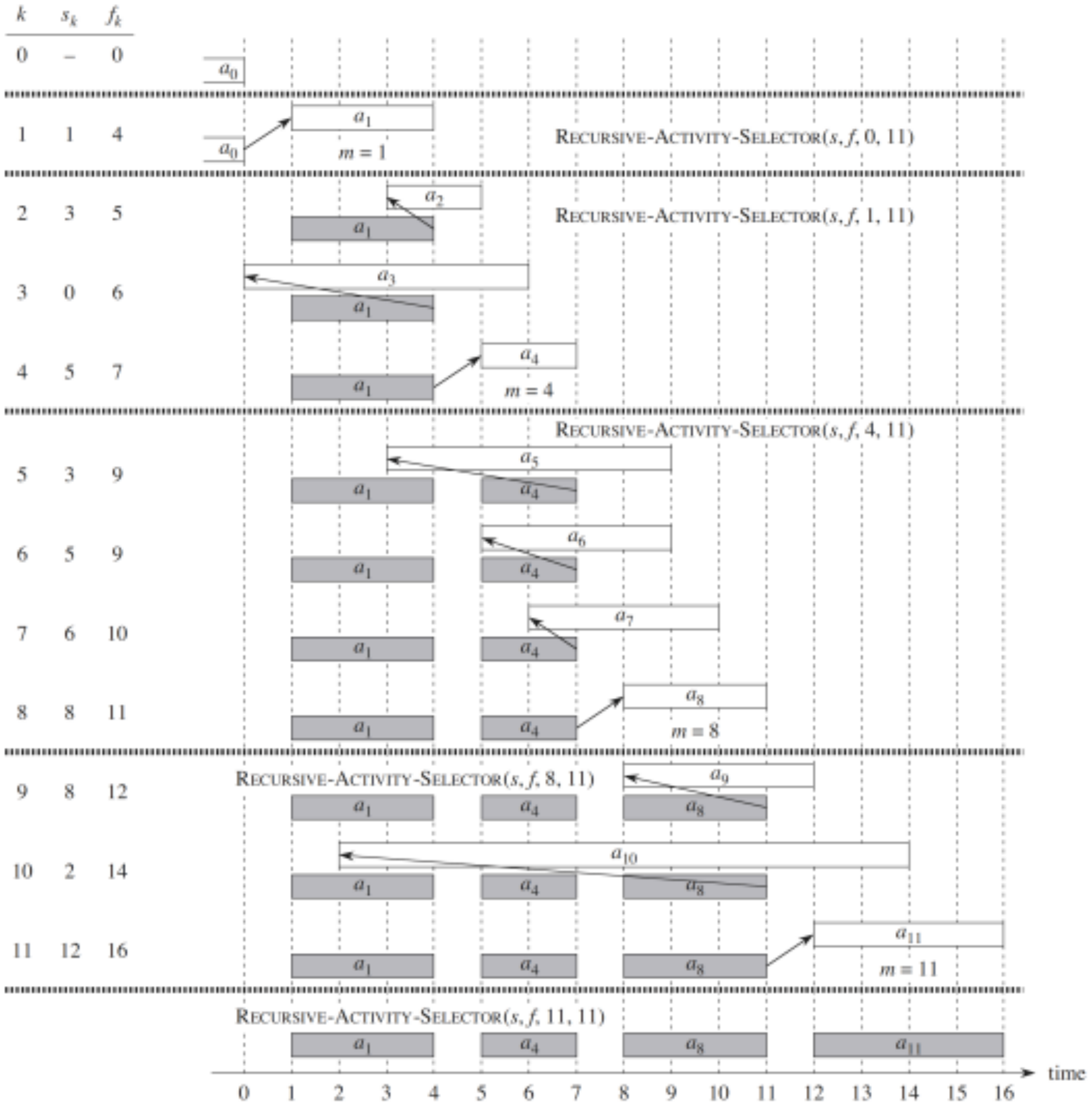
$(A_1A_2)A_3$: $2{\times}10{\times}5 + 2{\times}5{\times}20 = 300$

$A_1(A_2A_3)$: $10{\times}5{\times}20 + 2{\times}10{\times}20 = 1400$



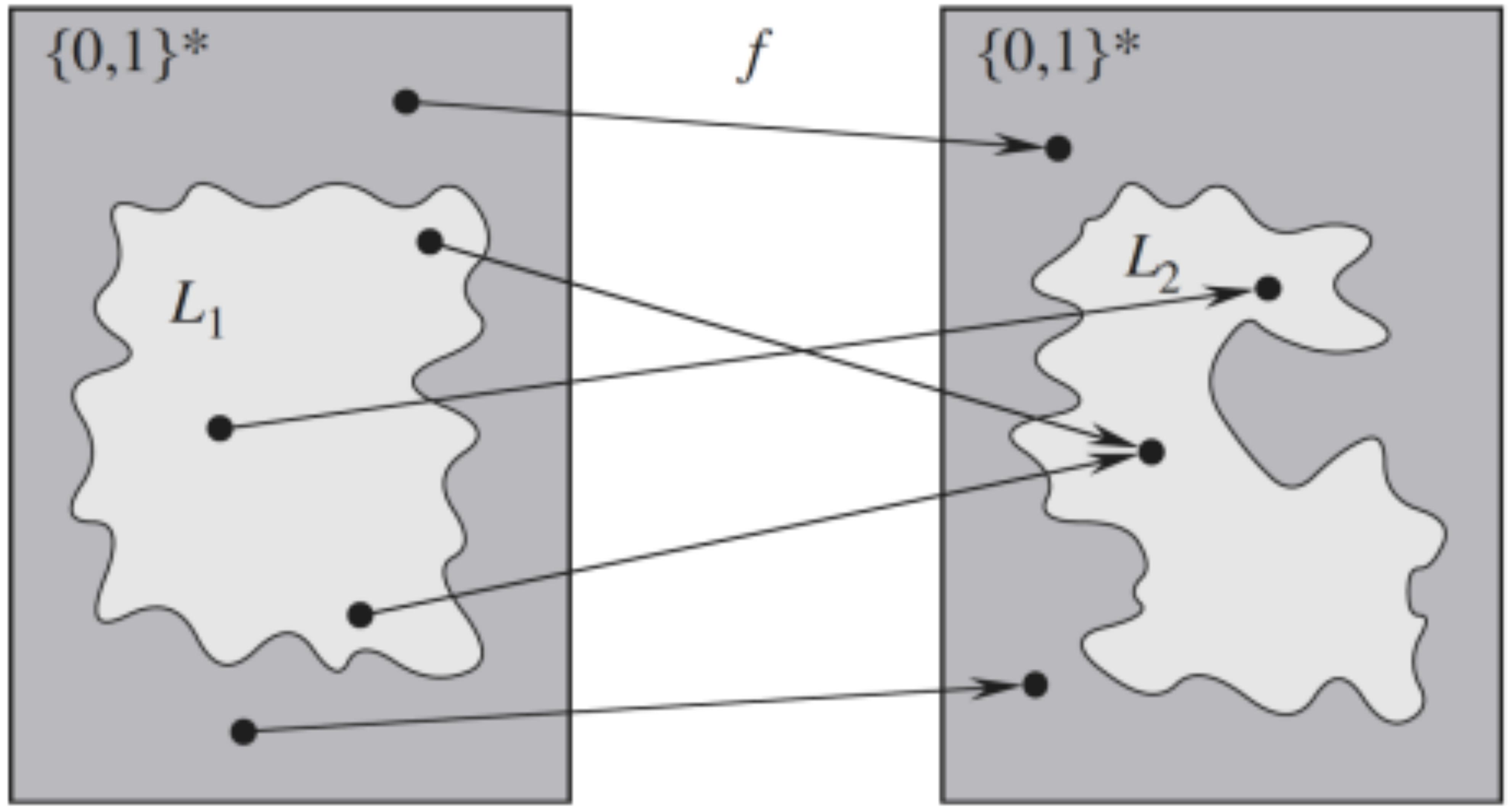
Greedy algorithms

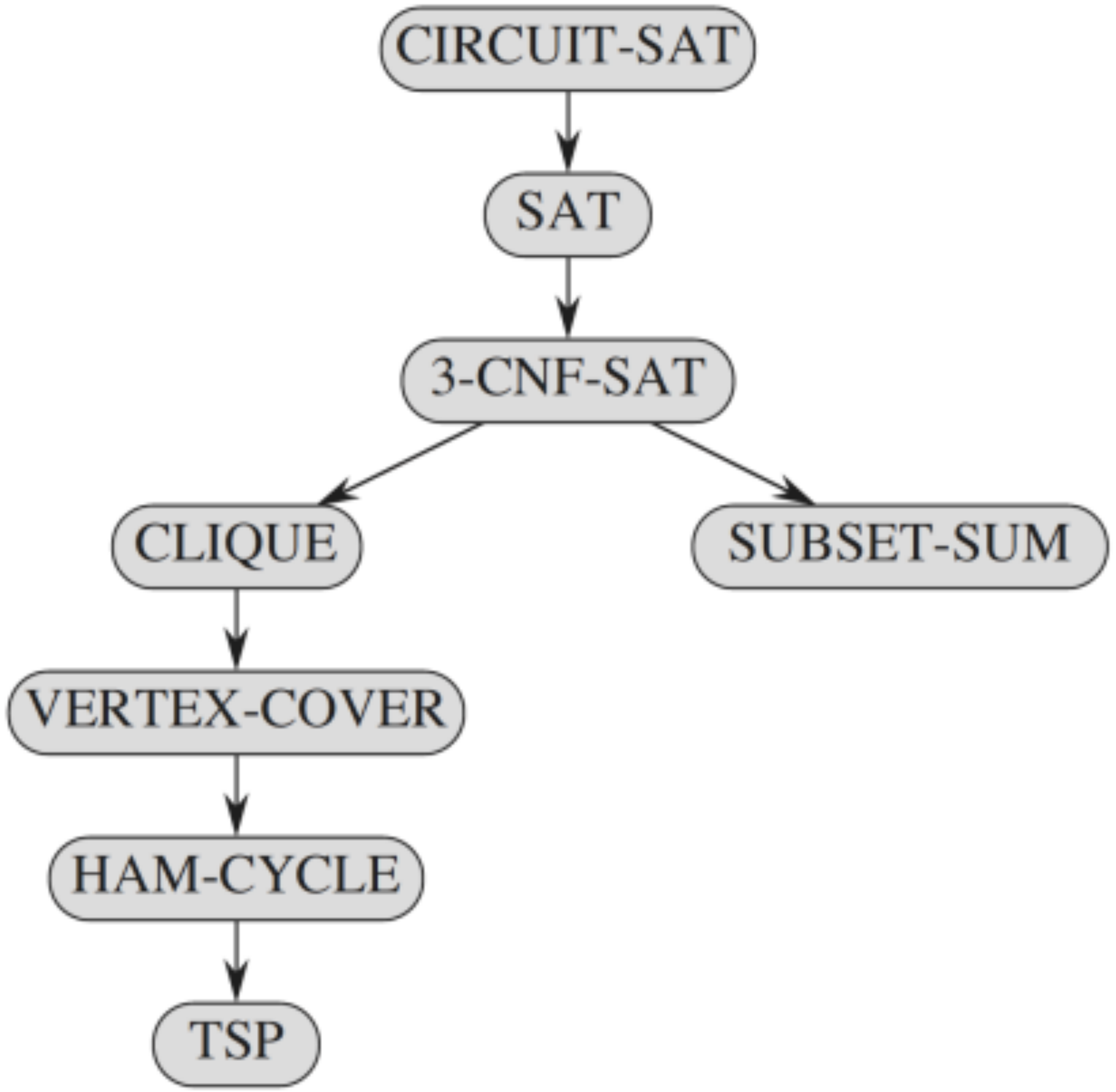
Activity-selection problems: Given a collection of n tasks (s_i, f_i) with start and finish time, select a maximum-size subset of mutually compatible tasks



NP completeness

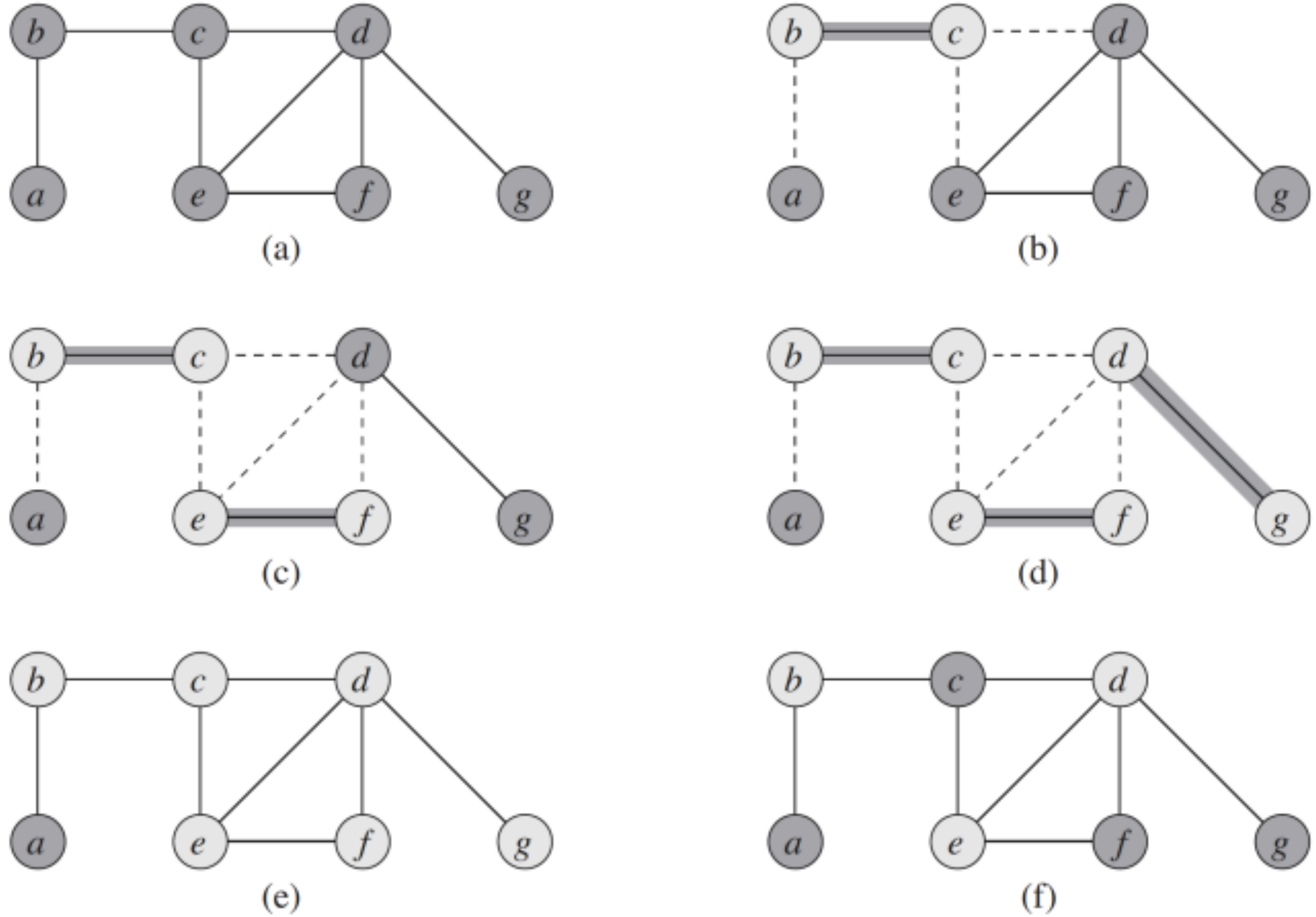
Polynomial-time reduction:





Approximation algorithms

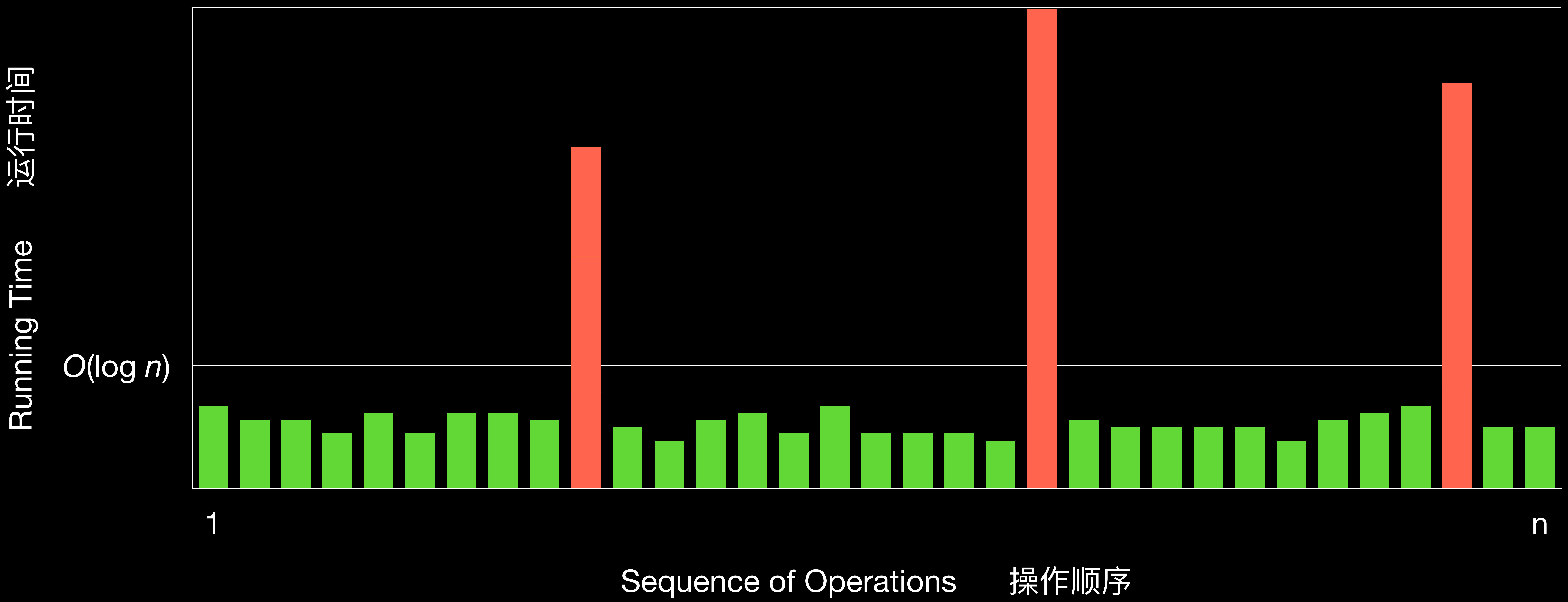
- Approximated vertex cover:



Amortized Analysis

- Timing analysis of a sequence of operations.
  If some operations in the sequence are slow,
  but we know that only a small number of operations can be slow,
  then we can give a better bound of the total runtime
  than just "length of sequence × time of slowest operation".

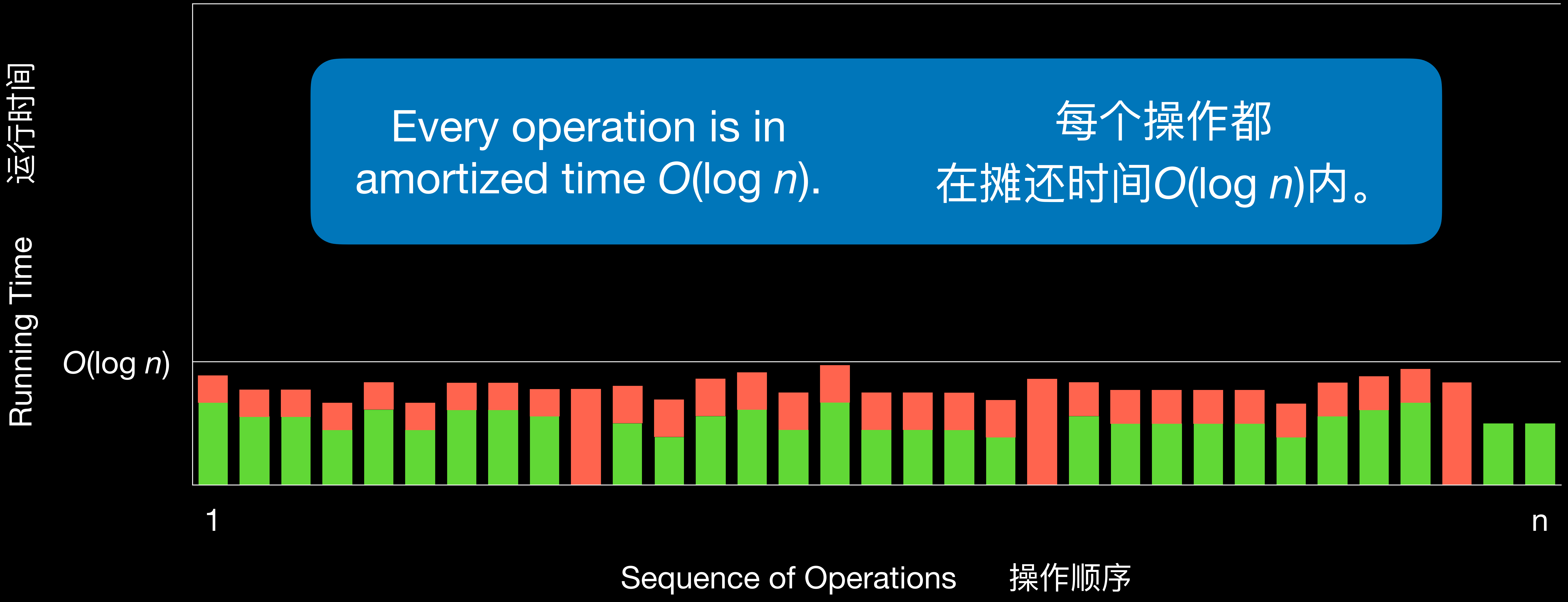
Scapegoat Tree 替罪羊树



Scapegoat Tree 替罪羊树

Every operation is in amortized time $O(\log n)$.

每个操作都在摊还时间$O(\log n)$内。



Graph algorithms

- Breadth-first / depth-first search: visit every vertex of a graph
- Shortest paths
- Network flow

Weighted Graphs 权重的图

Shortest Path 最短路径

Minimum Spanning Tree 最小生成树

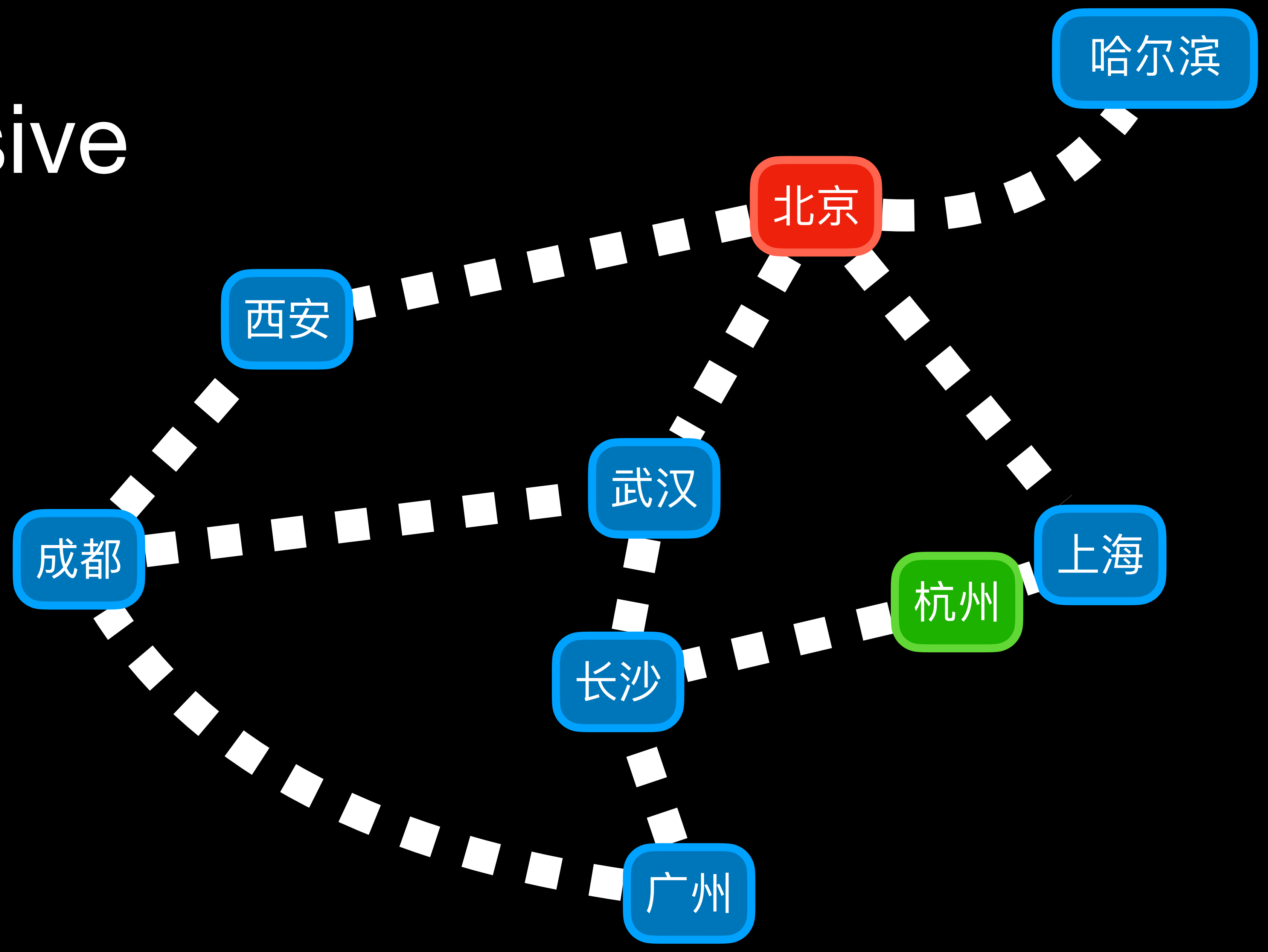
weight = length or cost of the edge

How far / how expensive is the trip from Beijing to Hangzhou?

What is the cheapest connected network?

Maximum Flow 最大流

capacity = width of the edge



How many people can travel each day from Beijing to Guangzhou?

Linear programming

- General method to solve (linear) optimization problems
- example problem: china production plant

Example: Production Planning

- A company offers two products: decorated china and white china.

- If it produces only decorated china, it would need eight employees, and it could produce 2000 pieces per week, which sell at a profit of ¥5/piece (¥1250/employee). However, the company only has six employees.

- If it produces only white china, four employees would be enough, it can produce 3500 pieces per week, and the profit is ¥2/piece (¥1750/employee).

- How many employees should work on which product?

Summary

- Algorithm := sequence of instructions that transform input into output
  把输入转换成输出的计算步骤的序列

- Big-O Notation: describe asymptotic rate of growth of functions
  大O记号：描写函数的渐近的增长速度

- Divide and Conquer: a method to construct algorithms
  divide a problem into smaller problems and solve every one recursively

- Recurrence 递归式: describe runtime of a divide-and-conquer algorithm