Introduction to Algorithms

SARCAR Sayan

Faculty of Library, Information, and Media Science

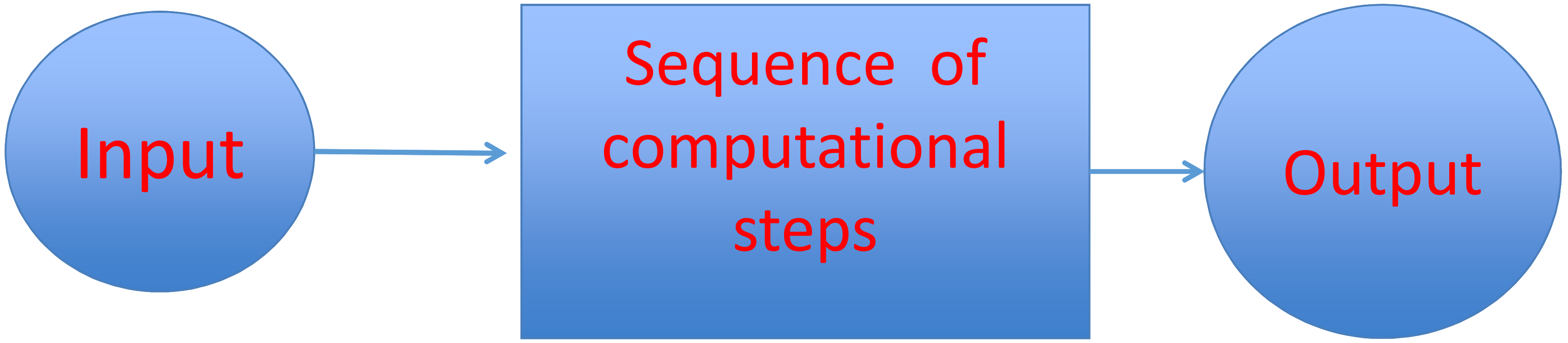
筑波大学
University of Tsukuba

Contents

- Logic and Reasoning
- Formal logic
- First order logic
- Propositional logic
- Quantifier
- Questions

What is Algorithm?

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.



It solves *Computational problems*

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number N
  - Output: Is the number prime?
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically
- Example:
  - Input: A picture in digital format
  - Output: An English description of what the picture shows

筑波大学 University of Tsukuba

Algorithm (many definitions)

An algorithm is an exact specification of how to solve a computational problem
An algorithm must specify every step completely, so a computer can implement it without any further “understanding”
An algorithm must work for all possible inputs of the problem.
Algorithms must be:

- Correct: For each input produce an appropriate output
- Efficient: run as quickly as possible, and use as little memory as possible – more about this later

There can be many different algorithms for each computational problem.

筑波大学
University of Tsukuba

Describing Algorithm

- Algorithms can be implemented in any programming language
- Usually we use “pseudo-code” to describe algorithms

```
Testing whether input N is prime:

For j = 2 .. N-1
  If j|N
    Output "N is composite" and halt
Output "N is prime"
```

筑波大学
University of Tsukuba

Greatest Common Divisor

- The first algorithm “invented” in history was Euclid’s algorithm for finding the greatest common divisor (GCD) of two natural numbers
- **Definition:** The GCD of two natural numbers *x, y* is the largest integer *j* that divides both (without remainder). I.e. $j|x$, $j|y$ and *j* is the largest integer with this property.
- **The GCD Problem:**
  - Input: natural numbers *x, y*
  - Output: *GCD(x,y)* – their GCD

Euclid’s GCD algorithm

```
public static int gcd(int x, int y) {
  while (y!=0) {
    int temp = x%y;
    x = y;
    y = temp;
  }
  return x;
}
```

筑波大学
University of Tsukuba

Euclid's GCD algorithm

```
while (y!=0) {
  int temp = x%y;
  x = y;
  y = temp;
}
```

Example: Computing GCD(48,120)

```
                        temp      x        y
After 0 rounds           --       72       120
After 1 round            72       120      72
After 2 rounds           48       72       48
After 3 rounds           24       48       24
After 4 rounds           0        24       0

                              Output: 24
```

筑波大学
University of Tsukuba

Square Root

- The problem we want to address is to compute the square root of a real number.
- When working with real numbers, we can not have complete precision.
  - The inputs will be given in finite precision
  - The outputs should only be computed approximately
- The square root problem:
  - Input: a positive real number *x*, and a precision requirement ε
  - Output: a real number *r* such that $|r-\sqrt{x}|\leq\varepsilon$

Square Root Algorithm

```
public static double sqrt(double x,
 double epsilon){
  double low = 0;
  double high = x>1 ? x : 1;
  while (high-low > epsilon) {
    double mid = (high+low)/2;
    if (mid*mid > x)
        high = mid;
    else
        low = mid;
  }
  return low;
}
```

筑波大学
University of Tsukuba

Binary Search Algorithm – sample run

```
while (high-low > epsilon) {
  double mid = (high+low)/2;
  if (mid*mid > x)
    high = mid;
  else
    low = mid;
}
```

Example: Computing sqrt(2) with precision 0.05:

```
                    mid       mid*mid      low        high
After 0 rounds       --         --          0          2
After 1 round         1          1          1          2
After 2 rounds       1.5        2.25        1          1.5
After 3 rounds       1.25       1.56..      1.25       1.5
After 4 rounds       1.37..     1.89..      1.37..     1.5
After 5 rounds       1.43..     2.06..      1.37..     1.43..
After 6 rounds       1.40..     1.97..      1.40..     1.43..


                      Output: 1.40…
```

筑波大学
University of Tsukuba

How fast will your program run?

- The running time of your program will depend upon:
  - The algorithm
  - The input
  - Your implementation of the algorithm in a programming language
  - The compiler you use
  - The OS on your computer
  - Your computer hardware
  - Maybe other things: temperature outside; other programs on your computer; …
- <u>Our Motivation:</u> analyze the running time of an algorithm as a function of only simple parameters of the input.

筑波大学
University of Tsukuba

Basic idea: counting operations

- Each algorithm performs a sequence of basic operations:
  - Arithmetic: (low + high)/2
  - Comparison: if (x > 0) …
  - Assignment: temp = x
  - Branching: while (true) { … }
  - …
- Idea: count the number of basic operations performed on the input.
- Difficulties:
  - Which operations are basic?
  - Not all operations take the same amount of time.
  - Operations take different times with different hardware or compilers

筑波大学 University of Tsukuba

Testing operation times on your system

```java
import java.util.*;
public class PerformanceEvaluation {
  public static void main(String[] args) {
    int i=0;   double d = 1.618;
    SimpleObject o = new SimpleObject();
    final int numLoops = 1000000;
    long startTime = System.currentTimeMillis();;
    for (i=0 ; i<numLoops ; i++){
      // put here a command to be timed
    }
    long endTime = System.currentTimeMillis();
    long duration = endTime - startTime;
    double iterationTime =  (double)duration / numLoops;
    System.out.println("duration: "+duration);
    System.out.println("sec/iter: "+iterationTime);
}}
class SimpleObject {
  private int x=0;
  public void m() { x++; }
}
```

筑波大学
University of Tsukuba

Sample running times of basic Java operations

Operation	Loop Body	nSec/iteration Sys1	nSec/iteration Sys2
Loop Overhead	;	196	10
Double division	d = 1.0 / d;	400	77
Method call	o.m();	372	93
Object Construction	o=new SimpleObject();	1080	110

Sys1: PII, 333MHz, jdk1.1.8, -nojit

Sys2: PIII, 500MHz, jdk1.3.1

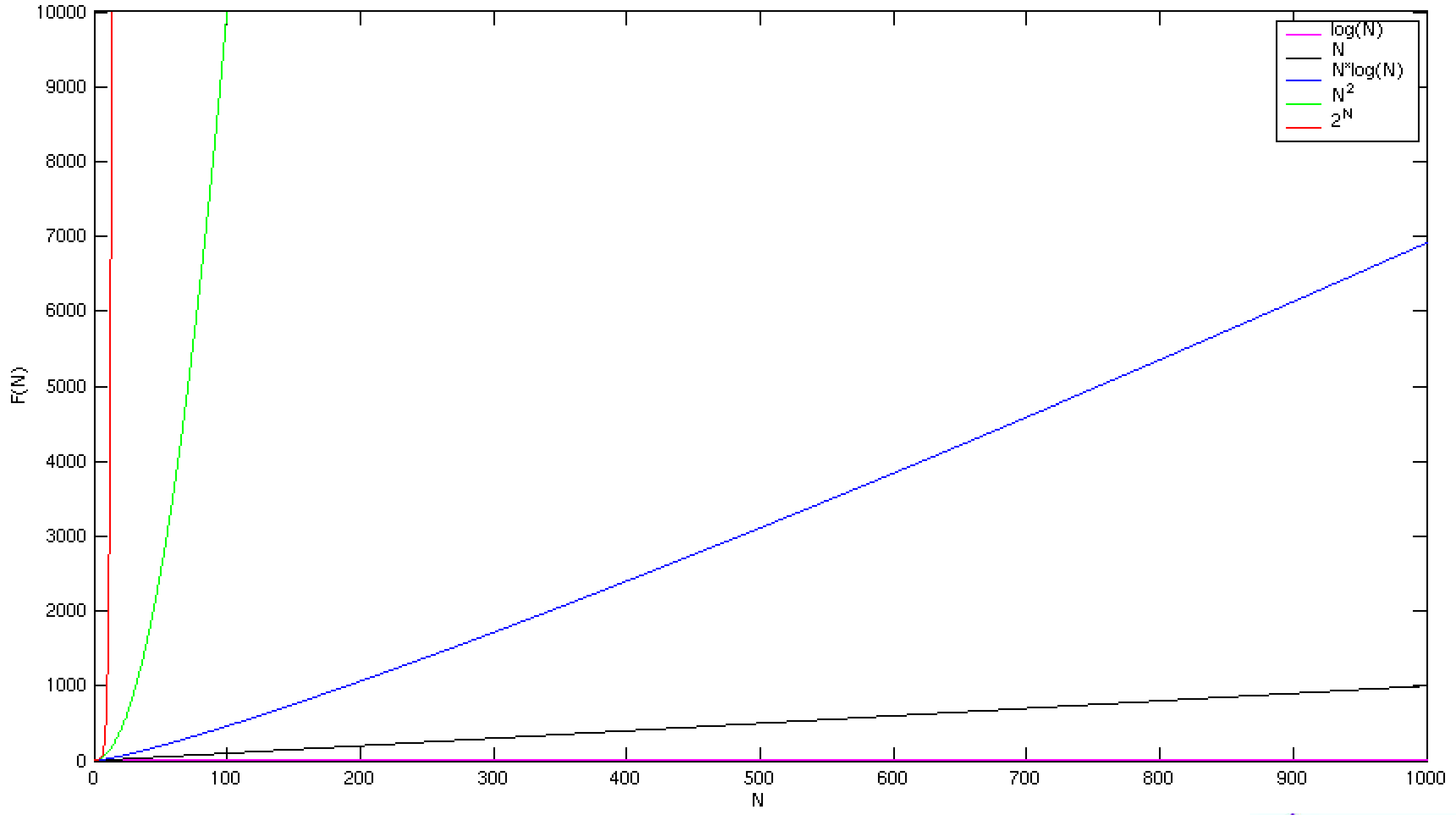
Asymptotic running times

- Operation counts are only problematic in terms of constant factors.
- The general form of the function describing the running time is invariant over hardware, languages or compilers!

```
public static int myMethod(int N){
  int sq = 0;
  for(int j=0; j<N ; j++)
    for(int k=0; k<N ; k++)
      sq++;
  return sq;
}
```

- Running time is “about” N^2 .
- We use “Big-O” notation, and say that the running time is $O(N^2)$

Asymptotic behavior of functions



Mathematical Formalization

- Definition: Let f and g be functions from the natural numbers to the natural numbers. We write $f=O(g)$ if there exists a constant c such that for all n: $f(n) \leq cg(n)$.

$$f=O(g) \quad \Leftrightarrow \quad \exists c \forall n: f(n) \leq cg(n)$$

- This is a mathematically formal way of ignoring constant factors, and looking only at the "shape" of the function.
- $f=O(g)$ should be considered as saying that "f is at most g, up to constant factors".
- We usually will have f be the running time of an algorithm and g a nicely written function. E.g. The running time of the previous algorithm was $O(N^2)$.

筑波大学
University of Tsukuba

Asymptotic analysis of algorithms

- We usually embark on an *asymptotic worst case* analysis of the running time of the algorithm.
- Asymptotic:
  - Formal, exact, depends only on the algorithm
  - Ignores constants
  - Applicable mostly for large input sizes
- Worst Case:
  - Bounds on running time must hold for *all* inputs.
  - Thus the analysis considers the worst-case input.
  - Sometimes the “average” performance can be much better
  - Real-life inputs are rarely “average” in any formal sense

The running time of Euclid’s GCD Algorithm

- How fast does Euclid’s algorithm terminate?
  - After the first iteration we have that $x > y$. In each iteration, we replace (x, y) with $(y, x\%y)$.
  - In an iteration where $x>1.5y$ then $x\%y < y < 2x/3$.
  - In an iteration where $x \leq 1.5y$ then $x\%y \leq y/2 < 2x/3$.
  - Thus, the value of xy decreases by a factor of at least $2/3$ each iteration (except, maybe, the first one).

```
public static int gcd(int x, int y) {
  while (y!=0) {
    int temp = x%y;
    x = y;
    y = temp;
  }
  return x;
}
```

The running time of Euclid’s Algorithm

- Theorem: Euclid’s GCD algorithm runs it time *O(N),* where N is the input length *(N=log2x + log2y).*
- Proof:
  - Every iteration of the loop (except maybe the first) the value of xy decreases by a factor of at least 2/3. Thus after *k+1* iterations the value of *xy* is at most $(2/3)^k$ the original value.
  - Thus the algorithm must terminate when k satisfies: $xy(2/3)^k < 1$ (for the original values of x, y).
  - Thus the algorithm runs for at most $1+\log_{3/2} xy$ iterations.
  - Each iteration has only a constant *L* number of operations, thus the total number of operations is at most $(1+\log_{3/2} xy)L$
  - Formally, $(1+\log_{3/2} xy)L \leq L(1+2\log_2 x+2\log_2 y) \leq 3LN$
  - Thus the running time is *O(N).*

筑波大学
University of Tsukuba

Running time of Square root algorithm

- The value of *(high-low)* decreases by a factor of exactly *2* each iteration. It starts at max(x,1), and the algorithm terminates when it goes below ε.
- Thus the number of iterations is at most $\log_2(\max(x,1)/\varepsilon)$
- The running time is $O(\log x + \log \varepsilon^{-1})$

```
public static double
  sqrt(double x, double epsilon){
  double low = 0;
  double high = x>1 ? x : 1;
  while (high-low > epsilon) {
    double mid = (high+low)/2;
    if (mid*mid > x)
      high = mid;
    else
      low = mid;
  }
  return low;
}
```

筑波大学 University of Tsukuba

Newton-Raphson Algorithm

```
public static double sqrt(double x, double epsilon){
  double r = 1;
  while ( Math.abs(r - x/r) > epsilon)
    r = (r + x/r)/2;
  return r;
}
```

Newton-Raphson – sample run

```
while ( Math.abs(r - x/r) > epsilon)
  r = (r + x/r)/2;
```

Example: Computing sqrt(2) with precision 0.01:

```
                         r        x/r
After 0 rounds           1         2
After 1 round            1.5       1.33..
After 2 rounds           1.41..    1.41..


                                     Output: 1.41…
```

Analysis of Running Time

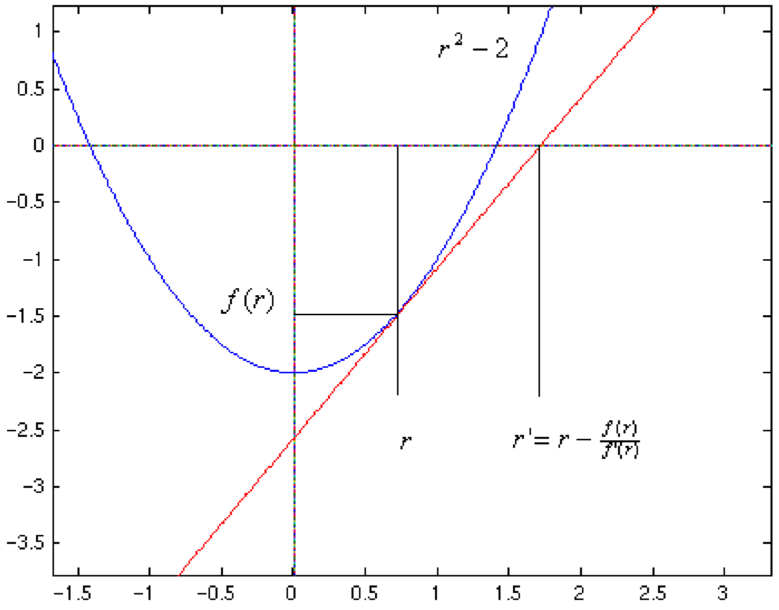
- Correctness is clear since for every *r* the square root of *x* is between and *r* and *x/r.*
- Here we will analyze the running time <u>only for *1<x<2*</u>
- Denote: $r' = (r + x/r)/2$

$$r'^2 - x = (r + x/r)^2/4 - x = \frac{r^4 + 2r^2x + x^2 - 4r^2x}{4r^2} = \frac{(r^2 - x)^2}{4r^2}$$

- Thus $\varepsilon_n < \varepsilon_{n-1}^2$, where $\varepsilon_n = r^2 - x$ after *n* loops
- At the beginning $\varepsilon_0 < 1$, and $\varepsilon_1 < 1/4$
- In general we have that $\varepsilon_n < 2^{-2^n}$
- At the end it suffices that $\varepsilon_n \le \varepsilon$, since $| r - \sqrt{x} | \le | r^2 - x |$
- Thus the algorithm terminates when $n = \log\log \varepsilon^{-1}$

In General…

- The Newton-Raphson method can be used to find the roots of any *differentiable* function *f*.
- In our case, to find √2, we solved $f(r) = r^2 - 2 = 0$
- *So,* $r' = r - \frac{f(r)}{f'(r)} = r - \frac{r^2 - 2}{2r} = \frac{r + 2/r}{2}$



Example: Sorting problem

•Input: A sequence of n numbers:

•Output: A permutation (reordering) of the input sequence such that

Ex. Input: sequence 31, 41, 59, 26, 41, 58
Output: sequence 26, 31, 41, 41, 58, 59

筑波大学 University of Tsukuba

Correct Algorithms

- An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one.

- *Incorrect algorithms can sometimes be useful, if their error rate can be controlled. (An example of this when we study algorithms for finding large prime numbers.)*

What kinds of problems are solved by algorithms?

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and our goal is to determine the shortest route from one intersection to another.
- We are given a sequence A1, A2, ..., An of n matrices, and we wish to determine their product A1. A2. … . An.
- We are given an equation ax ≡ b (mod n), where a, b, and n are integers, and we wish to find all the integers x, modulo n, that satisfy the equation.
- We are given n points in the plane, and we wish to find the convex hull of these points. The convex hull is the smallest convex polygon containing the points.

筑波大学 University of Tsukuba

Data structures

- A data structure is a way to store and organize data in order to facilitate access and modifications.
- No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them:
  - Table, Stacks and Queues, Linked lists
  - Representing rooted trees
  - Hash tables
  - Binary Search Trees
  - Red-black trees, …

Hard problems

- There are some problems for which no efficient solution is known, which are known as NP-complete:
  - it is unknown whether or not efficient algorithms exist for NP-complete problems.
  - the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.
  - a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

Choosing algorithms

Ex: Fibonacci sequence is defined as follows.

$$F(0) = 0, F(1) = 1, \text{ and}$$
$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1.$$

Write an algorithm to computer F(n).

Algorithms 1 and 2 for Fibonacci

```
function fib1(n){
  **if** n < 2 **then return** n;
  **else return** fib1(n-1) + fib1(n-2);
}

function fib2(n){
  i= 1; j = 0;
  **for** k = 1 **to** n **do** { j = i+j; i = j- i;}
  **return** j;
}
```

Algorithm 3 for Fibonacci

```
function  fib3(n){
  i = 1; j = 0; k = 0; h = 1;
  while n>0 do {
              if (n odd) then {  t = jh;
                                 j = ih + jk +t;
                                 i = ik +t;}
              t = h^2;
              h = 2kh+t;
              k = k^2+t;
              n = n div 2;}
  return j;
}
```

筑波大学
University of Tsukuba

Example of running times for Fibonacci

n	10	20	30	50	100	10000	1 000 000	100000000
fib1	8 ms	1 s	2 min	21 days				
fib2	1/6 ms	1/3 ms	½ ms	¾ ms	3/2 ms	150 ms	15 s	25 min
fib3	1/3 ms	2/5 ms	½ ms	½ ms	½ ms	1 ms	3/2 ms	2 ms

筑波大学
University of Tsukuba

Insertion sort

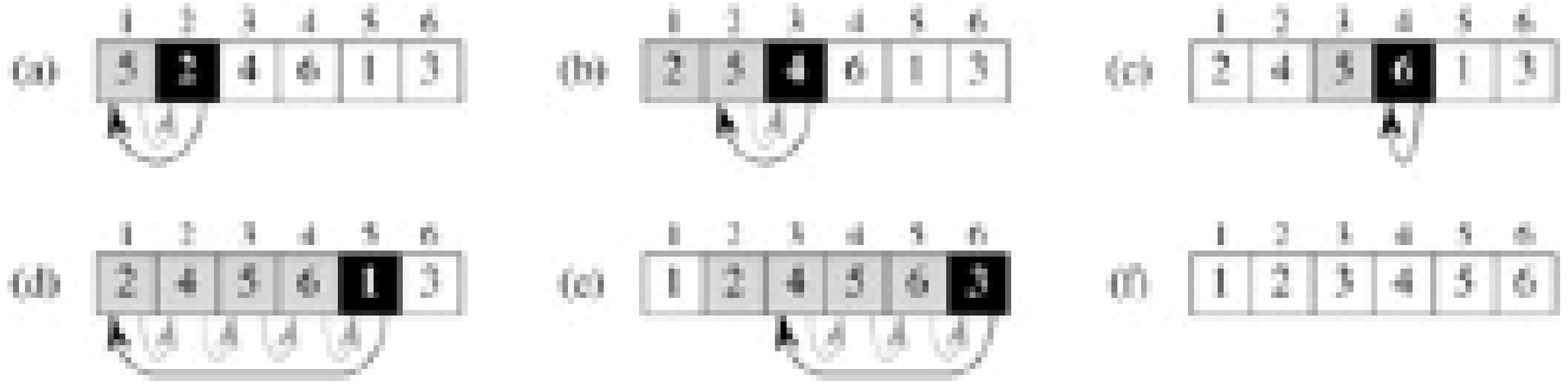
Efficient algorithm for sorting a small number of elements:

- We start with an empty left hand and the cards face down on the table.
- We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

```
INSERTION-SORT(A)
1. for j←2 to length[A]
2.     do key ← A[j]
3.         Insert A[j] into the sorted sequence A[1.. j - 1].
4.         i←j-1
5.         while i>0 and A[i]>key
6.             do A[i + 1] ← A[i]
7.                i←i-1
8.         A[i+1]←key
```

筑波大学
University of Tsukuba

Example



Proof of the correctness of Insertion sort

- We use loop invariants to help us understand why an algorithm is correct.
- We must show three things about a loop invariant:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Analyzing algorithms

- Analyzing an algorithm: for an input size,
  - measure memory (space)
  - measure computational time (running time).
- Input size: depends on the problem:
  - Sorting: number of items in the input; array size,… O(n)
  - Big integer (multiplying, …): number of bits to represent the input in binary notation O(log n)
  - Two number: input of a graph can be O(n,m), number of vertices and number of edges.
- Running time:
  - A constant amount of time is required to execute each line
  - each execution of the ith line takes time c_i , where c_i is a constant.

Analyzing of Insertion sort

- For each j = 2, 3, . . . , n, where n = length[A], we let t_j be the number of times the while loop test in line 5 is executed for that value of j.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

Best case and worst case

- Best case: the array is already sorted

  T(n) = c1n + c2(n - 1) + c4(n - 1) + c5(n - 1) + c8(n - 1)

  = (c1 +c2 +c4 +c5 +c8)n-(c2+c4 +c5 +c8) = a n + b

- Worst case: the array is in reverse sorted order

  T(n) = a n*n + b n + c

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) .
\end{aligned}
$$

Worst-case and average-case analysis

- worst-case running time: the longest running time for any input of size n:
  - upper bound on the running time for any input
  - for some algorithms, the worst case occurs fairly often
  - the "average case" is often roughly as bad as the worst case.
- average-case or expected running time:
  - technique of probabilistic analysis
  - assume that all inputs of a given size are equally likely
  - Difficult to analyze.

Designing algorithms

- The divide-and-conquer approach:
  - Divide the problem into a number of subproblems.
  - Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - Combine the solutions to the subproblems into the solution for the original problem.

- Recursive structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.

筑波大学
University of Tsukuba

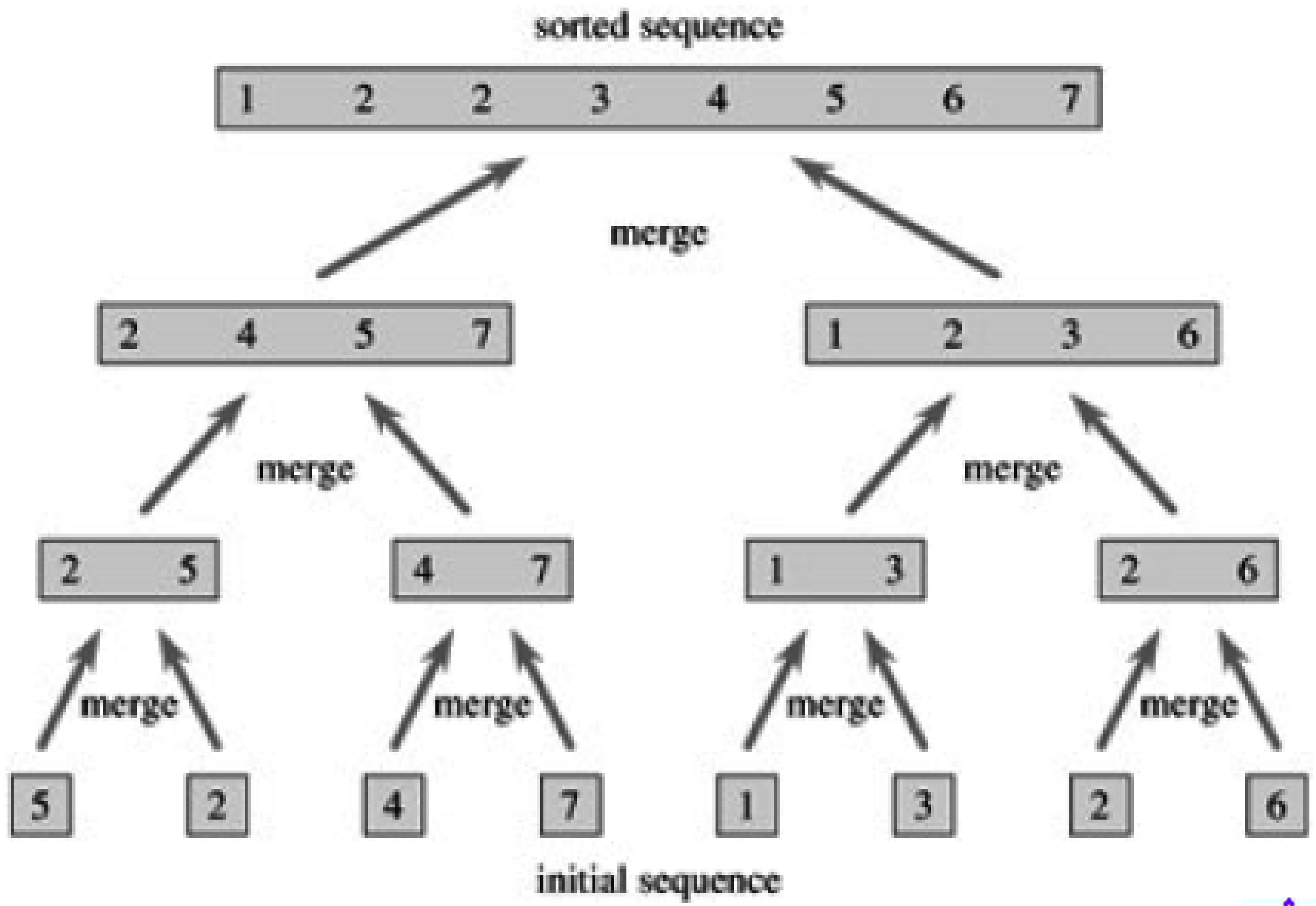
Merge sort algorithm

- Divide: Divide the n-elements sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

```
MERGE-SORT(A, p, r)
1.if p < r
2.     then q← ⌊(p+r)/2⌋
3.     MERGE-SORT(A, p, q)
4.     MERGE-SORT(A, q + 1, r)
5.     MERGE(A, p, q, r)
```

筑波大学
University of Tsukuba

Example


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5 2 4 7 1 3 2 6
initial sequence

Analyzing divide-and-conquer algorithms

- Divide: $D(n) = \Theta(1)$.
- Conquer: solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- Combine: the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \theta(n) & \text{if } n > 1 \end{cases}$$

Growth of Functions

- Asymptotic notation
  - The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency.
  - For input sizes large enough, we make only the order of growth of the running time relevant, so we study the asymptotic efficiency of algorithms.

Asymptotic notations

- g(n) is an **asymptotically tight bound for f(n):**

Θ(g(n)) = {f(n) : there exist positive constants c1, c2, and N such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n) for all n ≥ N}.

- **asymptotic upper bound:**

O(g(n)) = {f(n): there exist positive constants c and N such that

0 ≤ f(n) ≤ cg(n) for all n ≥ N}.

- **asymptotic lower bound:**

Ω(g(n)) = {f(n): there exist positive constants c and N such that

0 ≤ cg(n) ≤ f(n) for all n ≥ N}.

Asymptotic notations

- o(g(n)) = {f(n) : for any positive constant $c > 0$, there exists a constant $N > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge N$}.

- $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

筑波大学 University of Tsukuba

Asymptotic notations

- $f(n) = O(g(n)) \approx a \leq b$,
- $f(n) = \Omega(g(n)) \approx a \geq b$,
- $f(n) = \Theta(g(n)) \approx a = b$,
- $f(n) = o(g(n)) \approx a < b$,
- $f(n) = \omega(g(n)) \approx a > b$.

Example

- Order the following functions by O and θ

$$f_1(n) = n; \quad f_2(n) = 2^n; \quad f_3(n) = nlog_2(n);$$
$$f_(n) = n + n^3 + 7n^2; \quad f_5(n) = n^2 + log_2(n);$$
$$f_6(n) = n^2; \quad f_7(n) = 2^{2n}; \quad f_8(n) = n^5;$$
$$f_9(n) = \sqrt{n} + log_2(n); \quad f_{10}(n) = ln(2n);$$
$$f_{11}(n) = ln(n); \quad f_{12}(n) = 3^n + n^2;$$
$$f_{13}(n) = log_2(n)$$

Recurrences

- The substitution method
- The recursion method
- The master method

The substitution method

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

- Ex: $T(n) = 2\, T(n/2) + n$.
  1. We guess that $T(n) = O(n \lg n)$
  2. $T(n) \le 2(c\, n/2 \lg(n/2)) + n \le cn \lg(n/2) + n$

     $= cn \lg n - cn \lg 2 + n = cn \lg n - cn + n$

     $\le cn \lg n$

筑波大学 University of Tsukuba

Recursion method

Sum all the per-level costs to determine the total cost of all levels of the recursion.

Ex: $T(n) = 3T(n/4)+n$

$T(n) = n+ 3\ T(n/4)$

$= n + 3(n/4 + 3T(n/16))$

$= n + 3\ n/4 + 3\ (n/16 + 3T(n/64))$

$\leq n + 3n/4 + 9n/16 + \ldots$

$= O(n^2)$

筑波大学
University of Tsukuba

The master method

Master theorem: Let a ≥ 1 and b > 1 be constants, let f(n) be a

function, and let T (n) be defined by

$$\mathbf{T(n) = aT(n/b) + f(n)}$$

Then T (n) can be bounded asymptotically as follows.

1. If for some constant ε > 0, then
2. If then
3. If for some constant ε > 0, and

   if a f (n/b) ≤ c(n) for constant c <1 and all sufficiently large n,

   then *T (n) = Θ(f (n))*.

Using the master method

1. $T(n) = 9T(n/3) + n.$
2. $T(n) = T(2n/3) + 1$
3. $T(n) = 3T(n/4) + n \lg n$
4. $T(n) = 2T(n/2) + n \lg n$

Exercices

Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size *n, insertion sort runs in 8n2 steps, while merge sort runs in 64nlog(n) steps. For which values of n does insertion sort beat merge sort?*

Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

Exercises 2.3-7. Describe a Θ(*n lg n)-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.*

Exercises

Explain why the statement, "The running time of algorithm A is at least O(n2)," is meaningless.

Prove that the running time of an algorithm is Θ(g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is Ω(g(n)).

Problems

Inversions Let A[1,.., n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A.

a. List the five inversions of the array: 2, 3, 8, 6, 1.

b. What array with elements from the set {1, 2, . . . , n} has the most inversions? Howmany does it have?

c. What is the relationship between the running time of insertion sort and the number ofinversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on *n* elements in Θ(n lg n) worst-case time. (Hint: Modify merge sort.)

Problems

Recurrence examples

Give asymptotic upper and lower bounds for T(n) in each of the following recurrences. T(n) is constant for $n \le 2$. Make your bounds as tight as possible, and justify your answers.

a. T(n) = 2T(n/2) + n^3.

b. T(n) = T(9n/10) + n.

c. T(n) = 16T(n/4) + n^2.

d. T (n) = 7T(n/3) + n^2.

e. T(n) = 7T(n/2) + n2.

f. T(n) = 2T(n/4) + sqrt(n)

g. T(n)=T(n-1)+n.

h. T(n) = T(sqrt(n)) +1

筑波大学
University of Tsukuba

Q & A

Please write any feedback regarding class to
sayans@slis.tsukuba.ac.jp
Sub: Informatics class feedback

筑波大学
University of Tsukuba