Introduction to Algorithms

Topic 1 : Basic Concepts

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2022

Outline of Topics

Why Study Algorithms?

Sorting

Running Time Analysis

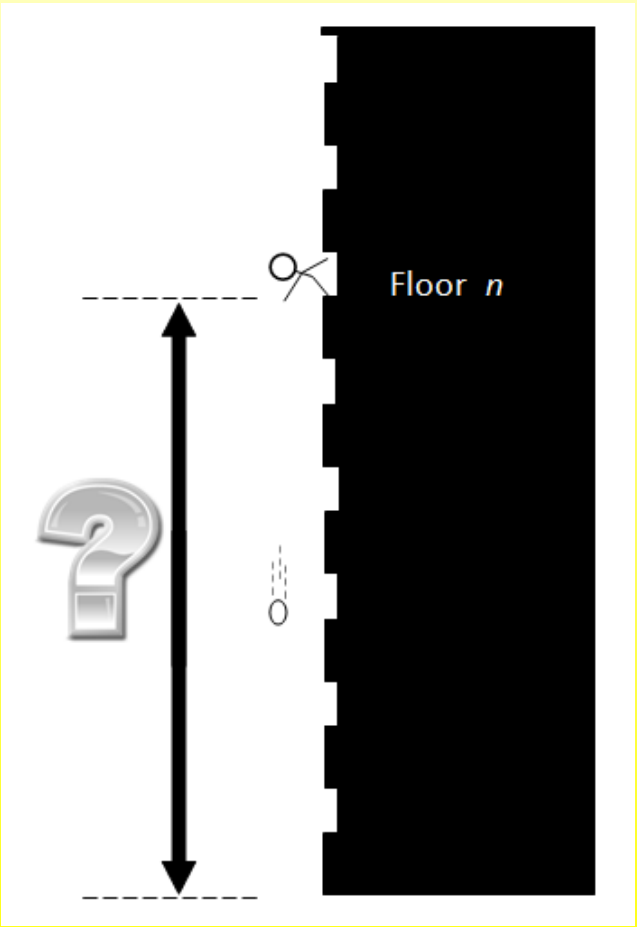
Why study algorithms?

- Compulsory
- To be a smart programmer
- To be more intelligent
- The way to research

  .....

A Google Interview Question

- You are given 2 identical eggs and have access to a 100-story building.
- Eggs may break if dropped from the first floor or may not even break if dropped from 100th floor.
- You need to figure out the highest floor an egg can be dropped without breaking.
- You are allowed to break 2 eggs. How many drops you need to make.



A Perplexing Polynomial Puzzle

- Imagine that you are given a black box . It has a slot where you can put any real number, after you place one, it makes some sound, vibrates a bit, and returns a value.
- Inside the box is a secret polynomial $p(x)$: if you put in x, you get out $p(x)$.
- All the coefficients of the polynomial are natural numbers.
- Your job is to find the exact coefficients of the polynomial.

The Secretary Problem

We have n candidates (perhaps applicants for a job or possible marriage partners). Our goal is choose the very best candidate. The assumptions are

- Candidates can be totally ordered from best to worst with no ties.
- Candidates arrive sequentially in random order.
- We can only determine the relative ranks of the candidates as they arrive. We cannot observe the absolute ranks.
- After each interview we must either immediately accept or reject the applicant. Once a candidate is rejected, she can not be recalled. Once a candidate is accepted, we stopped interviewing.
- The number of candidates n is known.

Candies for Children

There are n children standing in a line. Each child is assigned a rating value given in the integer array ratings.
You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

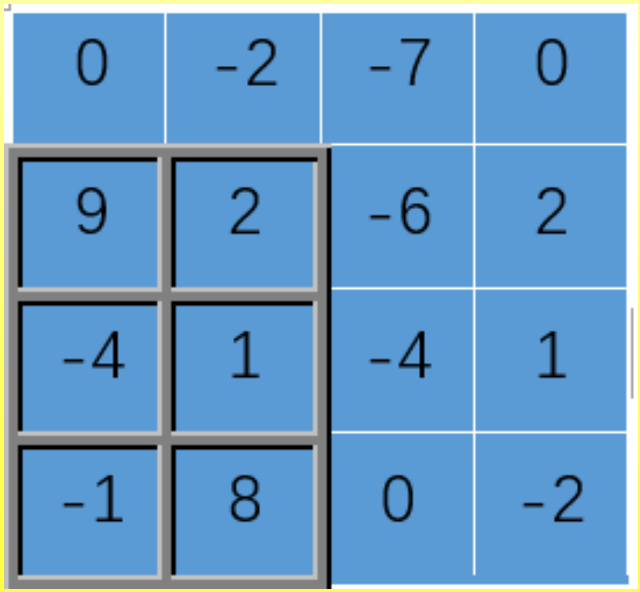
Return the minimum number of candies you need to have to distribute the candies to the children.

Erect the Fence

- You are given an array trees where $trees[i] = [x_i, y_i]$ represents the location of a tree in the garden.
- You are asked to fence the entire garden using the minimum length of rope as it is expensive. The garden is well fenced only if all the trees are enclosed.
- Return the coordinates of trees that are exactly located on the fence perimeter.

Another question

- Suppose you have an $N \times N$ matrix of positive and negative integers.
- Write some code that finds the sub-matrix with the maximum sum of its elements.



Another question

- Suppose you have given N companies, and we want to eventually merge them into one big company.
- How many ways are there to merge? Assume that each time you can merge two companies into one new company.

Another question

- You are given a small sorted list of numbers, and a very very long sorted list of numbers - so long that it had to be put on a disk in different blocks.
- How would you find those short list numbers in the bigger one?

Another question

- You are given with three sorted arrays (in ascending order)
- you are required to find a triplet (one element from each array) such that distance is minimum.
- If $a[i], b[j]$ and $c[k]$ are three elements then distance = $\max(|a[i]-b[j]|, |a[i]-c[k]|, |b[j]-c[k]|)$.
- Please give a solution in O(n) time complexity.

Another question

- Given an array whose elements are sorted, return the index of the first occurrence of a specific integer.
- Do this in sub-linear time.
- Do not just go through each element searching for that element.

Course Goals

- Use O, Ω, and Θ notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.

Course Goals

- Use O, Ω, and Θ notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
- Determine the time complexity of simple algorithms, deduce the recurrence relations that describe the time complexity of recursively defined algorithms, and solve simple recurrence relations.

Course Goals

- Use O, Ω, and Θ notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
- Determine the time complexity of simple algorithms, deduce the recurrence relations that describe the time complexity of recursively defined algorithms, and solve simple recurrence relations.
- Design algorithms using the brute-force, greedy, dynamic programming, divide-and-conquer, branch and bound strategies.

Course Goals

- Use O, Ω, and Θ notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
- Determine the time complexity of simple algorithms, deduce the recurrence relations that describe the time complexity of recursively defined algorithms, and solve simple recurrence relations.
- Design algorithms using the brute-force, greedy, dynamic programming, divide-and-conquer, branch and bound strategies.
- Design algorithms using at least one other algorithmic strategy from the list of topics for this unit.

Course Goals, Cont.

- Use and implement the fundamental abstract data types – specifically including hash tables, binary search trees, and graphs – necessary to solve algorithmic problems efficiently.

Course Goals, Cont.

- Use and implement the fundamental abstract data types – specifically including hash tables, binary search trees, and graphs – necessary to solve algorithmic problems efficiently.
- Solve problems using techniques learned in the design of sequential search, binary search, $O(N \log N)$ sorting algorithms, and fundamental graph algorithms, including depth-first and breadth-first search, single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm.

Course Goals, Cont.

- Use and implement the fundamental abstract data types – specifically including hash tables, binary search trees, and graphs – necessary to solve algorithmic problems efficiently.
- Solve problems using techniques learned in the design of sequential search, binary search, $O(N \log N)$ sorting algorithms, and fundamental graph algorithms, including depth-first and breadth-first search, single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm.
- Demonstrate the following abilities: to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in simple programming contexts.

Course Goals, Cont.

- Communicate theoretical and experimental analyses of a set of algorithms (i.e. sorting) in a lab report format.

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability
- robustness

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability
- robustness
- user-friendliness

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability
- robustness
- user-friendliness
- programmer time

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability
- robustness
- user-friendliness
- programmer time
- simplicity

Analysis of Algorithms

The theoretical study of computer-program performance and resource usage. What's more important than performance?

- correctness
- functionality
- reliability
- modularity
- maintainability
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility

Why study algorithms and performance?

- Algorithms help us to understand **scalability**.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a **language** for talking about program behavior.
- Performance is the **currency** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

The Problem of Sorting

The Problem of Sorting

Input: a sequence $\mathscr{A} = \langle a_1, a_2, a_3, \cdots, a_{n-1}, a_n \rangle$ of n numbers.
Output: a permutation $\langle a'_1, a'_2, a'_3, \cdots, a'_{n-1}, a'_n \rangle$ of the sequence $\mathscr{A}$ such that

$$a'_1 \leq a'_2 \leq a'_3 \leq \cdots \leq a'_{n-1} \leq a'_n$$

An example:

Input :	8	2	4	9	3	6
Output :	2	3	4	6	8	9

Insertion Sort

The pseudocode for insertion sorting:
INSERTIONSORT(A,n) **Note**[1]

1: **for** $j \leftarrow 2$ to n **do**
2: $\quad key \leftarrow A[j]$
3: $\quad i \leftarrow j-1$
4: $\quad$ **while** $i > 0$ and $A[i] > key$ **do**
5: $\quad\quad A[i+1] \leftarrow A[i]$
6: $\quad\quad i \leftarrow i-1$
7: $\quad A[i+1] = key$

[1] Here A is an array of n elements.

Insertion Sort

The pseudocode for insertion sorting:
INSERTIONSORT(A, n) **Note**[1]

1: **for** $j \leftarrow 2$ to n **do**
2: $\quad key \leftarrow A[j]$
3: $\quad i \leftarrow j-1$
4: $\quad$ **while** $i > 0$ and $A[i] > key$ **do**
5: $\quad\quad A[i+1] \leftarrow A[i]$
6: $\quad\quad i \leftarrow i-1$
7: $\quad A[i+1] = key$

Loop Invariant: $A[1 \cdots j-1]$ is sorted.

[1] Here A is an array of n elements.

Insertion Sort: An example

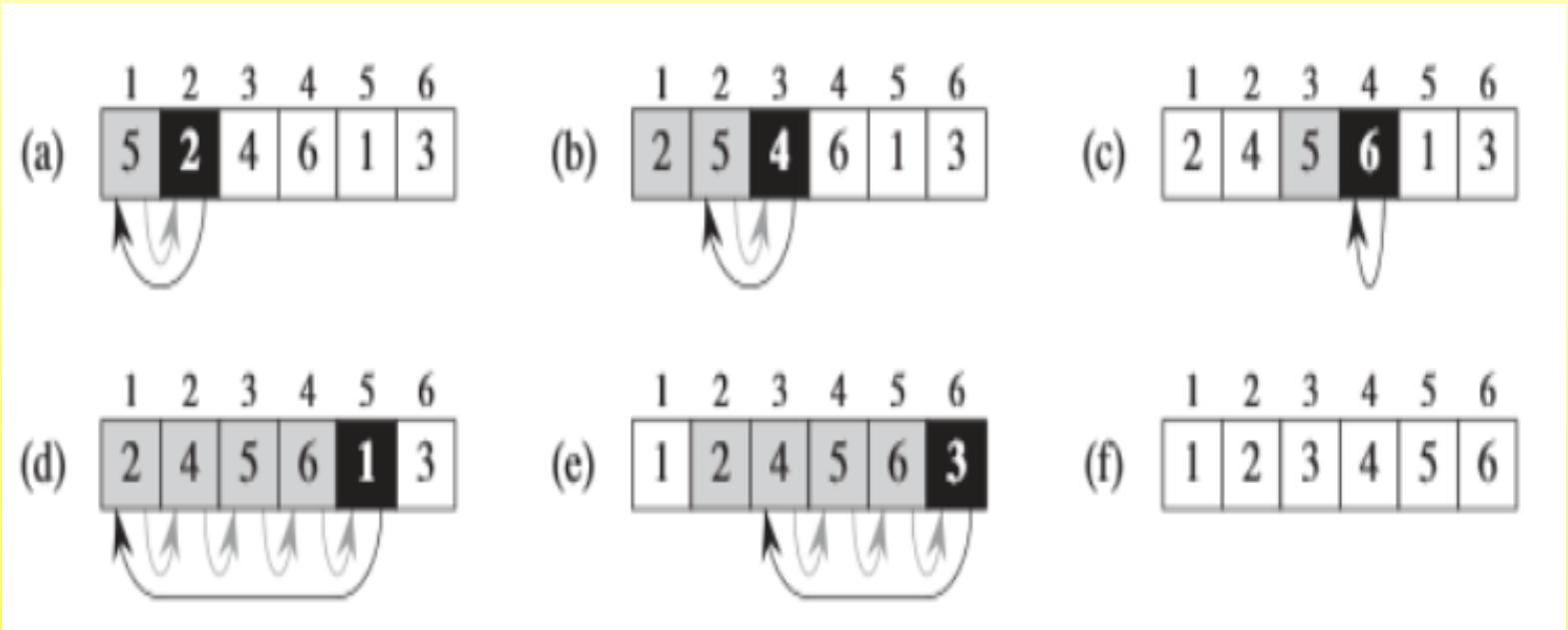


Figure: The operation of INSERTION-SORT on the array A = 5, 2, 4, 6, 1, 3

Running Time Analysis

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the **size** of the input, since short sequences are easier to sort than long ones.
- Generally, we seek **upper bounds** on the running time, because everybody likes a guarantee.

Different Analysis Approaches

- **Worst-case: (usually)**
  $T(n)$ = maximum time of algorithm on any input of size n.

Different Analysis Approaches

- **Worst-case: (usually)**
  $T(n)$ = maximum time of algorithm on any input of size n.
- **Average-case: (sometimes)**
  $T(n)$ = expected time of algorithm over all inputs of size n by assuming that the input follow some distribution.
  Need assumption of statistical distribution of inputs (such as random uniform distribution or Poisson distribution).

Different Analysis Approaches

- **Worst-case: (usually)**
  $T(n)$ = maximum time of algorithm on any input of size n.
- **Average-case: (sometimes)**
  $T(n)$ = expected time of algorithm over all inputs of size n by assuming that the input follow some distribution.
  Need assumption of statistical distribution of inputs (such as random uniform distribution or Poisson distribution).
- **Best-case: (bogus)**
  Cheat with a slow algorithm that works fast on some input.

Different Analysis Approaches

- **Smooth analysis: (newest approach)**
  - For any instance of the problem, analyze the time complexity by assuming that the input can be perturbed with small perturbation.
  - First example: Simplex method that solves linear programming has polynomial time smooth complexity (while its worst case complexity is exponential).

Different Analysis Approaches

- **Smooth analysis: (newest approach)**
  - If the smoothed complexity of an algorithm is low, then it is unlikely that the algorithm will take a long time to solve practical instances whose data are subject to slight noises and imprecisions.
  - Smoothed complexity results are strong probabilistic results, in every large enough neighbourhood of the space of inputs, most inputs are easily solvable.

Different Analysis Approaches

- **Smooth analysis: (newest approach)**
  - Smoothed analysis generalizes both worst-case and average-case analysis and inherits strengths of both.
  - It is intended to be much more general than average-case complexity, while still allowing low complexity bounds to be proven.

Machine-independent time

What is insertion sort's worst-case time?
It depends on the speed of our computer:

- relative speed (on the same machine),
- absolute speed (on different machines).

BIG IDEA:
Ignore machine-dependent constants.
Look at growth of $T(n)$ as $n \to \infty$.

Machine-independent time

What is insertion sort's worst-case time?
It depends on the speed of our computer:

- relative speed (on the same machine),
- absolute speed (on different machines).

BIG IDEA:
Ignore machine-dependent constants.
Look at growth of $T(n)$ as $n \to \infty$.

Asymptotic Analysis

Further Assumptions

Constant time operations

- plus, minus, times, division
- load, store, copy and so on
- control operations: branch, subroutine call and so on.

Θ-Notation

Mathematics:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1 \textit{ and } c_2 \textit{ and}$$

$$\textit{integer } n_0 \textit{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \textit{ for all } n \geq n_0\}$$

Engineering:

- Drop low-order terms; ignore leading constants
- Example: $3n^3 + 90n^2 - 5n + 2006 = \Theta(n^3)$.

O-Notation

Mathematics:

$$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c_2 \textit{ and}$$

$$\textit{integer } n_0 \textit{ such that } f(n) \leq c_2 g(n) \textit{ for all } n \geq n_0\}$$

Engineering:

- Drop low-order terms; ignore leading constants
- Example: $3n^3 + 90n^2 - 5n + 2006 = O(n^3)$, or $O(n^4)$.

Ω-Notation

Mathematics:

$$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1 \textit{ and}$$

$$\textit{integer } n_0 \textit{ such that } 0 \leq c_1 g(n) \leq f(n), \textit{for all } n \geq n_0\}$$

Engineering:

- Drop low-order terms; ignore leading constants
- Example: $3n^3 + 90n^2 - 5n + 2006 = \Omega(n^3)$, or $\Omega(n^2)$.

Asymptotic performance

When n is large enough, a $\Theta(n \log n)$ algorithm always beats a $\Theta(n^2)$ algorithm.

- We should not always ignore the asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Insertion sort analysis

INSERTIONSORT(A,n) **Note**[2]

for $j \leftarrow 2$ to n **do**

$\Longrightarrow$ Loop n-1 times

$key \leftarrow A[j]$

$i \leftarrow j-1$

while $i > 0$ and $A[i] > key$ **do**

$\Longrightarrow$ Loop $j/2$ times on average, Loop j times in worst case.

$A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] = key$

[2]Here A is an array of n elements.

Insertion sort analysis

Worst Case: input reversely sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Insertion sort analysis

Worst Case: input reversely sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Average-case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Insertion sort analysis

Worst Case: input reversely sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Average-case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is Insertion sorting a fast sorting algorithm?

Insertion sort analysis

Worst Case: input reversely sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Average-case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is Insertion sorting a fast sorting algorithm?

- Moderately so, for small n.

Insertion sort analysis

Worst Case: input reversely sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Average-case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is Insertion sorting a fast sorting algorithm?

- Moderately so, for small n.
- **Not at all** for large n.

More on Average-case Analysis for Insertion Sort

Expected Value: The expected value of a random variable X on a probability space (S,p) is the sum

$$E(X) = \sum_{s \in S} X(s)p(s),$$

where $X(s)$ is the value of variable X in state s and $p(s)$ is the probability that state s happens.
Here we assume that the space S is composed of discrete events.

Expected Value Example

For example, an American roulette wheel has 38 equally possible outcomes. A winning bet placed on a single number pays 35-to-1 (this means that you are paid 35 times your bet and your bet is returned, so you get 36 times your bet). So considering all 38 possible outcomes, the expected value of the profit resulting from a \$1 bet on a single number is:

$$\left(-\$1 \times \frac{37}{38}\right) + \left(\$35 \times \frac{1}{38}\right),$$

which is about -\$0.0526. Therefore one expects, on average, to lose over five cents for every dollar bet, and the expected value of a one dollar bet is \$0.9474.

Linearity of Expectation

Theorem: Let X_1 and X_2 be random variables (need to be the same probability space (S,p)). Then $E(X_1+X_2)=E(X_1)+E(X_2)$.
Example: When two *fair dice* are rolled, here are both calculations for the expected total value:

$$E(X_1+X_2)=E(X_1)+E(X_2)=\frac{7}{2}+\frac{7}{2}=7$$

$$E(X_1+X_2)=\sum_{j=1}^{6}\sum_{i=1}^{6}(i+j)\cdot\frac{1}{36}=7$$

Notice here $E(X)=\sum_{i=1}^{6} i\cdot\frac{1}{6}=7/2$.

Average Case Computational Complexity

Compute the expected value of the random variable that counts how many operations are executed by the algorithm.
X_i is the random variable equal to the number of comparisons used to insert a_i into the proper position after the first $i-1$ elements have been sorted. Clearly, we have $X_1 = 0$, and

$$1 \leq X_i \leq i-1$$

Let X denote the total number of comparisons used to sort the array. Then

$$X = X_1 + X_2 + X_3 + \cdots + X_n$$

$$E(X) = E(X_1) + E(X_2) + E(X_3) + \cdots + E(X_n)$$

For random data, it is equally likely the i^{th} element could go in any sorted position from 1 to $i-1$. Thus,

$$p(k\ comparisons) = 1/(i-1)$$

Then

$$E(X_i) = \sum_{k=1}^{i-1} k \cdot p(k\ comparisons) = \sum_{k=1}^{i-1} \frac{k}{i-1} = i/2$$

Then

$$E(X) = \sum_{i=2}^{n} E(X_i) = \sum_{i=2}^{n} \frac{i}{2} = \frac{(n-1)(n+2)}{4}$$

What do we mean exactly by random data in our previous analysis?

What do we mean exactly by random data in our previous analysis?

$a_1, a_2, \cdots, a_i \cdots, a_{n-1}, a_n$ is a random input if each of the possible $n!$ permutations is equally possible.

Merge Sort Idea

- divide and conquer (and combine) approach, recursive algorithm

Merge Sort Idea

- divide and conquer (and combine) approach, recursive algorithm
- basic step, you can merge two sorted lists of total length n in $\Theta(n)$ linear time

Merge Sort Idea

- divide and conquer (and combine) approach, recursive algorithm
- basic step, you can merge two sorted lists of total length n in $\Theta(n)$ linear time
- second key idea, a list of length one element is sorted

Merge Sort Idea

- divide and conquer (and combine) approach, recursive algorithm
- basic step, you can merge two sorted lists of total length n in $\Theta(n)$ linear time
- second key idea, a list of length one element is sorted

We will see many algorithms that are essentially divide and conquer. The key steps of these algorithms are
(1) [Divide]: divide the problem into smaller sub-problems
(2) [Conquer]: solve each sub-problems
(3) [Merge]: merge the solutions from sub-problems to form a solution for the problem.

Merge Sort

MERGESORT(A, p, r)

1: **if** $p < r$ **then**
2: MERGESORT$(A, p, \lfloor \frac{p+r}{2} \rfloor)$
3: MERGESORT$(A, \lfloor \frac{p+r}{2} \rfloor$+1,r)
4: **Merge** the two sorted lists.

Merge Sort

MERGESORT(A, p, r)

1: **if** $p < r$ **then**
2: MERGESORT$(A, p, \lfloor \frac{p+r}{2} \rfloor)$
3: MERGESORT$(A, \lfloor \frac{p+r}{2} \rfloor$+1,r)
4: **Merge** the two sorted lists.

It is a recursive algorithm. The **Key subroutine** is **Merge**.

Merge Two Sorted Arrays

MERGE(A, B, C) Note[3]

```
1:  i ← 1, j ← 1, k ← 1, DONE ← FALSE
2:  while not DONE do
3:      while j ≤ n_B and A[i] ≥ B[j] do
4:          C[k] ← B[j], k++, j++
5:      DONE ← (j > n_B)
6:      if not DONE then
7:          C[k] ← A[i], k++, i++
8:          DONE ← (i > n_A)
9:  while i ≤ n_A do
10:     C[k] ← A[i], k++, i++
11: while j ≤ n_B do
12:     C[k] ← B[j], k++, j++
```

[3] Here A and B are sorted arrays of n_A and n_B sizes, C stores the final sorted result

Sentinel Card

To avoid having to check whether either input pile is empty in each basic step, we can place on the bottom of each pile a sentinel card, which contains a special value ∞ that we use to simplify our code.

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A, p, r)

1: **if** $p < r$ **then**
2: MERGESORT$(A, p, \lfloor \frac{p+r}{2} \rfloor)$
3: MERGESORT$(A, \lfloor \frac{p+r}{2} \rfloor$+1,r)
4: **Merge** the two sorted lists.

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A,p,r) $\Longrightarrow T(n)$

1: **if** $p < r$ **then**
2: MERGESORT$(A,p,\lfloor\frac{p+r}{2}\rfloor)$
3: MERGESORT$(A,\lfloor\frac{p+r}{2}\rfloor$+1,r)
4: **Merge** the two sorted lists.

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A,p,r) $\Longrightarrow T(n)$

1: **if** $p < r$ **then**
2: MERGESORT$(A,p,\lfloor\frac{p+r}{2}\rfloor)$ $\Longrightarrow T(n/2)$
3: MERGESORT$(A,\lfloor\frac{p+r}{2}\rfloor$+1,r)
4: **Merge** the two sorted lists.

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A, p, r) $\Longrightarrow T(n)$

1: **if** $p < r$ **then**
2: MERGESORT$(A, p, \lfloor \frac{p+r}{2} \rfloor)$ $\Longrightarrow T(n/2)$
3: MERGESORT$(A, \lfloor \frac{p+r}{2} \rfloor$+1,r) $\Longrightarrow T(n/2)$
4: **Merge** the two sorted lists.

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A, p, r) $\Longrightarrow T(n)$

1: **if** $p < r$ **then**
2: MERGESORT$(A, p, \lfloor \frac{p+r}{2} \rfloor)$ $\Longrightarrow T(n/2)$
3: MERGESORT$(A, \lfloor \frac{p+r}{2} \rfloor$+1,r) $\Longrightarrow T(n/2)$
4: **Merge** the two sorted lists. $\Longrightarrow c \cdot n = \Theta(n)$

Merge Sort Time Complexity Analysis

Assume that the input array has size n. Let $T(n)$ be the time to sort A using merge sort.

MERGESORT(A,p,r) $\Longrightarrow T(n)$

1: **if** $p < r$ **then**
2: MERGESORT$(A,p,\lfloor\frac{p+r}{2}\rfloor)$ $\Longrightarrow T(n/2)$
3: MERGESORT$(A,\lfloor\frac{p+r}{2}\rfloor$+1,r) $\Longrightarrow T(n/2)$
4: **Merge** the two sorted lists. $\Longrightarrow c\cdot n = \Theta(n)$

Thus, we have

$$T(n) = 2T(n/2) + c\cdot n$$

Recurrence for merge sort

Thus we have the following recurrence relations for $T(n)$:

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

- **Sloppiness**: The actual formula should be $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c \cdot n$. But it turns out not to matter asymptotically.
- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- We will provide several ways to find a good upper bound on $T(n)$ (see CLRS for more details).

Time complexity of merge sort

Theorem: The asymptotic time complexity for merge sorting is $\Theta(n \log n)$.

Approaches:

- Recursion tree
- Substitution method (guess, verify, solve)
- Master Theorem

Want More Information?

For more information on the course, visit http://202.38.86.171/. All handouts and important information will be posted there. Please check it for new information.