CLRS Notes

MIT 6.046J

Author: Haopeng Li

Date: July 28, 2022

ElegantLaTeX Program

Contents

Chapter 1 Analysis of Algorithms

Introduction

- ❑ *Insertion sort*
- ❑ *Asymptotic analysis*
- ❑ *Merge sort*
- ❑ *Recurrences*

Definition 1.1 (Algorithms)

The theoretical study of computer-program performance and resource usage. ♣

Note *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!*

1.1 The problem of sorting

Problem 1.1(The problem of sorting)

- Input:sequence $< a_1, a_2, \cdots, a_n >$ of numbers.
- Output: permutation $< a'_1, a'_2, \cdots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

1.2 Insertion Sort

```
Insertion-Sort(A,n)
 for j <- 2 to n
   do key <- A[j]
     i <- j - 1
     while i > 0 and A[i] > key
       do A[i+1] <- A[i]
       i<- i-1
     A[i+1] = key
```

1.3 Running time

- 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.

1.3.1 Kinds of Analysis

Definition 1.2 (Worst-Case(usually))

$T(n)$ = maximum time of algorithm on any input of size n. ♣

Definition 1.3 (Average-Case(Sometimes))

- *$T(n)$ = expected time of algorithm over all inputs of size n.*
- *Need assumption of statistical distribution of inputs.* ♣

Definition 1.4 (Best-case: (bogus))

Cheat with a slow algorithm that works fast on some input. ♣

Note *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).

Note *BIG IDEA:*

1. Ignore machine-dependent constants.
2. look at the growth of $T(n)$ as $n \to \infty$
3. "Asymptotic Analysis"

1.3.2 Θ-Notation

Definition 1.5 (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* $n \ge n_0\}$ ♣

Note *Engineering:Drop low-order terms; Ignore leading constants.*

Example 1.1

$$3n^3 + 90n^2 - 5n + 6046 = \Theta\left(n^3\right)$$

1.3.3 Asymptotic performance

Note *When n gets large enough, a $\Theta\left(n^2\right)$ algorithm always beats a $\Theta\left(n^3\right)$ algorithm.*

Note

- *We shouldn't ignore 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.*

1.3.4 Insertion sort analysis

Worst case

Input reverse sorted

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

Average case

All permutations equally likely.

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

Note *Is insertion sort a fast sorting algorithm?*

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

1.4 Merge Sort

```
Merge-Sort A[1..n]
  1. If n = 1,done
  2. Recurisively sort A[1...⌈n/2⌉] and A[⌈n/2⌉ + 1...n]
  3. Merge the 2 sorted lists.
```

Note *Key subroutine: MERGE*

1.4.1 Analyzing Merge Sort

Time $= \Theta(n)$ to merge a total of n elements (linear time).

$T(n)$	MERGE-SORT $A[1 \ldots n]$
$\Theta(1)$	1. If $n = 1,$ done.
$2T(n/2)$	2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.
$\Theta(n)$	3. "Merge" the 2 sorted lists

Note ***Sloppiness**: $2T(n/2)$ should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.*

1.4.2 Recurrence for merge sort

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

Note

- *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.*
- *CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.*

1.4.3 Recursion tree

Example 1.2 Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

1.4.4 Conclusions

- $\Theta(n \lg n)$ grows more slowly than $\Theta\left(n^2\right)$
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.

Chapter 2 Asymptotic Notation & Recurrences

Introduction

- $O-, \Omega-$, *and* $\Theta-$ *notation*
- *Substitution method*
- *Iterating the recurrence*
- *Recursion tree*
- *Master method*

2.1 Asymptotic notation

2.1.1 $O-$notation (upper bounds)

Definition 2.1 ($O-$notation (upper bounds))

We write $f(n) = O(g(n))$ *if there exist constants* $c > 0, n_0 > 0$ *such that* $0 \leq f(n) \leq \mathrm{c}g(n)$ *for all* $n \geq n_0$ ♣

Example 2.1

$$2n^2 = O\left(n^3\right) \quad (c = 1, n_0 = 2)$$

Note *Notice that in this equation,* $2n^2$ *are functions not values and the eqal sign is just "one-way" equality. Actually, it can be denoted more precisely:*

$$2n^2 \in O\left(n^3\right)$$

Note *Convention: A set in a formula representsan anonymous function in the set.*

Example 2.2

$$n^2 + O(n) = O\left(n^2\right)$$

means for any $f(n) \in O(n)$,$n^2 + f(n) = h(n)$ for some $h(n) \in O(n^2)$

2.1.2 $\Omega-$ notation(lower bounds)

O-notation is an upper-bound notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

Definition 2.2 ($\Omega-$ notation(lower bounds))

$\Omega(g(n)) = \{f(n)$: *there exist constants* $c > 0, n_0 > 0$ *such that* $0 \leq \mathrm{c}g(n) \leq f(n)$ *for all* $n \geq n_0\}$ ♣

Example 2.3

$$\sqrt{n} = \Omega(\lg n) \quad (c = 1, n_0 = 16)$$

2.1.3 $\Theta-$notation(tight bounds)

Definition 2.3 ($\Theta-$notation(tight bounds))

$$\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n))$$

♣

Example 2.4

$$\frac{1}{2}n^2 - 2n = \Theta\left(n^2\right)$$

2.1.4 $o-$notation and ω-notation

O-notation and Ω-notation are like $\leq$ and $\geq$. o-notation and ω-notation are like $<$ and $>$.

Definition 2.4 ($o-$notation)

$o(g(n)) = \{f(n)$: *for any constant* $c > 0$ *,there is a constant* $n_0 > 0$ *such that* $0 \leq f(n) < cg(n)$ *for all* $n \geq n_0\}$

♣

Example 2.5

$$2n^2 = o\left(n^3\right) \quad (n_0 = 2/c)$$

Definition 2.5 (ω-notation)

$o(g(n)) = \{f(n)$: *for any constant* $c > 0$ *, there is a constant* $n_0 > 0$ *Such that* $0 \leq f(n) < \mathrm{cg}(n)$ *for all* $n \geq n_0\}$

♣

Example 2.6

$$\sqrt{n} = \omega(\lg n) \quad (n_0 = 1 + 1/c)$$

2.2 Solving recurrences

The analysis of merge sort from Lecture 1 required us to solve a recurrence.Recurrences are like solving integrals, differential equations, etc. Learn a few tricks.

2.2.1 Substitution method

Definition 2.6 (Substitution method)

The most general method:

1. *Guess the form of the solution.*
2. *Verify by induction.*
3. *Solve for constants.*

♣

Example 2.7

$$T(n) = 4T(n/2) + n$$

- Assume that $T(1) = \Theta(1)$.
- Guess $O\left(n^3\right)$. (Prove O and Ω separately.)
- Assume that $T(k) \leq ck^3$ for $k < n$
- Prove $T(n) \leq cn^3$ by induction.

Solution

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4c(n/2)^3 + n \\ &= (c/2)n^3 + n \\ &= cn^3 - \left((c/2)n^3 - n\right) \leftarrow \textit{ desired - residual} \\ &\le cn^3 \longleftarrow \textit{ desired} \end{aligned}$$

whenever $(c/2)n^3 - n \ge 0$ *, for example,if* $c \ge 2$ *and* $n \ge 1$

Note *We must also handle the initial conditions, that is, ground the induction with base cases.*

1. **Base**: $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
2. For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!

Note *A tighter upper bound? We shall prove that* $T(n) = O\left(n^2\right)$

Assume that $T(k) \le ck^2$ for $k < n$:

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\le 4c(n/2)^2 + n \\ &= cn^2 + n \\ &= O\left(n^2\right) \end{aligned}$$

Wrong! We must prove the I.H.($T(k) \le ck^2$)

$$= cn^2 - (-n)$$

for no choice of $c > 0$. Lose!

Note *IDEA: Strengthen the inductive hypothesis.(Subtract a low-order term.)*

Inductive hypothesis: $T(k) \le c_1k^2 - c_2k$ *for* $k < n$*.*

Solution

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4\left(c_1(n/2)^2 - c_2(n/2)\right) + n \\ &= c_1n^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\le c_1n^2 - c_2n \textit{ if } c_2 \ge 1. \end{aligned}$$

Pick c_1 *big enough to handle the initial conditions.*

2.2.2 Recursion-tree method

- A recursion tree models the costs(time)ofa recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (...)
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method.

Example 2.8 Solve

$$T(n) = T(n/4) + T(n/2) + n^2$$

2.2.3 The master method

The master method applies to recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive.

Theorem 2.1 (Three common cases)

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ *for some constant* $\varepsilon > 0$
   - $f(n)$ *grows polynomially slower than* $n^{\log_b a}$ *(by an* n^{ε} *factor).*

   Solution:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

2. $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$ *for some constant* $k \geq 0$
   - $f(n)$ *and* $n^{\log_b a}$ *grow at similar rates.*

   Solution:

$$T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$$

3. $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ *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:

$$T(n) = \Theta(f(n))$$

Example 2.9

$$T(n) = 4T(n/2) + n^3$$

Solution

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3$$

CASE 3*:* $f(n) = \Omega\left(n^{2+\varepsilon}\right)$ *for* $\varepsilon = 1$ *and* $4(n/2)^3 \leq cn^3$ *(reg. cond.) for* $c = 1/2$.

$$\therefore T(n) = \Theta\left(n^3\right)$$

Example 2.10

$$T(n) = 4T(n/2) + n^2/\lg n$$

Solution

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n$$

Master method does not apply. *In particular, for every constant* $\varepsilon > 0$*, we have* $n^{\varepsilon} = \omega(\lg n)$.

2.2.3.1 Idea of master theorem

Chapter 3 Divide and Conquer

Introduction

- *Binary search*
- *Powering a number*
- *Fibonacci numbers*
- *Matrix multiplication*
- *Strassen's algorithm*
- *VLSI tree layout*

3.1 The divide-and-conquer design paradigm

1. Divide the problem (instance) into subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine subproblem solutions.

Note *Merge Sort*

1. *Divide: Trivial.*
2. *Conguer:Recursively sort 2 subarrays.*
3. *Combine:Linear-time merge.*

$$T(n) = 2T(n/2) + \Theta(n)$$

Using Master theorem :

Merge sort: $a = 2, b = 2 \Rightarrow n^{\log_b a} = n^{\log_2 2} = n \Rightarrow$ *CASE 2* $(k = 0) \Rightarrow T(n) = \Theta(n \lg n)$.

3.2 Binary search

Find an element in a sorted array:

1. Divide: Check middle element
2. Conquer: Recursively search 1 subarray.
3. Combine: Trivial.

$$T(n) = 1T(n/2) + \Theta(1)$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{ CASE } 2(k = 0)$$

$$\Rightarrow T(n) = \Theta(\lg n)$$

3.3 Powering a number

Problem 3.1 Compute a^n, where $n \in \mathbb{N}$.

Solution

Naive algorithm: $\Theta(n)$

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \textit{if } n \textit{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \textit{if } n \textit{ is odd} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n)$$

3.4 Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Note ***Naive recursive algorithm:*** *$\Omega(\phi^n)$ (exponential time), where $\phi = (1+\sqrt{5})/2$ is the golden ratio.*

Note

- ***Bottom-up:***
  - *Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.*
  - *Running time: $\Theta(n)$.*
- ***Naive recursive squaring:***

  $F_n = \phi^n/\sqrt{5}$ rounded to the nearest integer.
  - *Recursive squaring: $\Theta(\lg n)$ time.*
  - *This method is unreliable, since floating-point arithmetic is prone to round-off errors.*

Theorem 3.1 (Recursive squaring)

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

♡

Note *Algorithm: Recursive squaring.*

$$\textit{Time } = \Theta(\lg n)$$

Proof

- Base (n = 1):

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$$

- Inductive step ($n \geq 2$):

$$\begin{aligned} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} &= \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \end{aligned}$$

3.5 Matrix multiplication

Definition 3.1 (Matrix multiplication)

$$\left.\begin{array}{ll} \textit{Input:} & A = [a_{ij}]\,, B = [b_{ij}]\,. \\ \textit{Output:} & C = [c_{ij}] = A \cdot B. \end{array}\right\} i, j = 1, 2, \ldots, n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

♣

3.5.1 Standard algorithm

```
for i <- 1 to n
    do for j <- 1 to n
        do c_ij < -0
            for k <- 1 to n
                do c_ij <- c_ij + a_ik · b_kj
```

Running time $= \Theta\left(n^3\right)$

3.5.2 Divide-and-conquer algorithm

Note *$n \times n$ matrix = 2 × 2 matrix of $(n/2) \times (n/2)$ submatrices:*

$$\left[\begin{array}{c:c} r & s \\ \hdashline t & u \end{array}\right] = \left[\begin{array}{c:c} a & b \\ \hdashline c & d \end{array}\right] \cdot \left[\begin{array}{c:c} e & f \\ \hdashline g & h \end{array}\right]$$
$$C = A \cdot B$$

$$\left.\begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{array}\right\} 8 \textit{ mults of } (n/2) \times (n/2) \textit{ submatrices}$$

$$T(n) = 8T(n/2) + \Theta\left(n^2\right)$$

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{ CASE 1} \Rightarrow T(n) = \Theta\left(n^3\right)$$

No better than the ordinary algorithm.

3.6 Strassen's Algorithm

3.6.1 Strassen's idea

Multiply 2×2 matrices with only 7 recursive mults.

$$\begin{aligned} P_1 &= a \cdot (f - h) & r &= P_5 + P_4 - P_2 + P_6 \\ P_2 &= (a + b) \cdot h & s &= P_1 + P_2 \\ P_3 &= (c + d) \cdot e & t &= P_3 + P_4 \\ P_4 &= d \cdot (g - e) & u &= P_5 + P_1 - P_3 - P_7 \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

Note *7 mults, 18 adds/subs.* ***No reliance on commutativity of mult!***

3.6.2 Strassen's algorithm

1. Divide: Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. Conquer: Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
3. Combine: Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

$$T(n) = 7T(n/2) + \Theta\left(n^2\right)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{ CASE } 1 \Rightarrow T(n) = \Theta\left(n^{\lg 7}\right)$$

Note *The number* 2.81 *may not seem much smaller than 3 , but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for* $n \geq 32$ *or so.*

Note *Best to date (of theoretical interest only):* $\Theta\left(n^{2.376\cdots}\right)$

3.7 VLSI layout

Problem 3.2 Embed a complete binary tree with n leaves in a grid using minimal area.

3.8 Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- The divide-and-conquer strategy often leads to efficient algorithms.

Chapter 4 Quicksort

Introduction

- *Divide and conquer*
- *Partitioning*
- *Worst-case analysis*
- *Intuition*
- *Randomized quicksort*
- *Analysis*

4.1 Quicksort

Note

- *Proposed by C.A.R.Hoare in 1962.*
- *Divide-and-conquer algorithm.*
- *Sorts"in place"(like insertion sort,but not like merge sort).*
- *Very practical (with tuning).*

4.1.1 Divide and conquer

Quicksort an $n-$element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.
2. **Conquer:**Recursively sort the two subarrays.
3. **Combine:**Trivial.

Key: *Linear-time Partitioning subroutine.*

4.1.2 Partitioning subroutine

```
PARTITION(A,p,q) //A[p..q]
    x <- A[p]   //pivot = A[p]
    i <- p
    for J <- p + 1 to q
        do if A[j] ≤ x
            then i <- i + 1
                exchange A[i] <-> A[j]
exchange A[p] <-> A[i]
return i
```

Note *Running time* $= O(n)$ *for* n *elements.*

Example 4.1 Example of partitioning

4.1.3 Pseudocode for 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)
```

4.2 Analysis of quicksort

1. Assume all input elements are distinct.
2. In practice,there are better partitioning algorithms for when duplicate input elements may exist.
3. Let $T(n)$ =worst-case running time on an array of n elements.

4.2.1 Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta\left(n^2\right) \qquad \text{(arithmetic series)} \end{aligned}$$

4.2.1.1 Worst-case recursion tree

4.2.2 Best-case analysis

If we're lucky,PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \text{(same as merge sort)} \end{aligned}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

4.2.3 More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ...

$$\begin{aligned} L(n) &= 2U(n/2) + \Theta(n) \qquad \text{lucky} \\ U(n) &= L(n-1) + \Theta(n) \qquad \text{unlucky} \end{aligned}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1)+\Theta(n/2))+\Theta(n) \\ &= 2L(n/2-1)+\Theta(n) \\ &= \Theta(n \lg n) \text{ Lucky!} \end{aligned}$$

How can we make sure we are usually lucky?

4.3 Randomized quicksort

Note ***IDEA:****Partition around a random element.*

- *Running time is independent of the input order.*
- *No assumptions need to be made about the input distribution.*
- *No specific input elicits the worst-case behavior.*
- *The worst case is determined only by the output of a random-number generator.*

4.4 Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

$$\begin{aligned} T(n) &= \begin{cases} T(0)+T(n-1)+\Theta(n) \text{ if } 0 : n-1 \text{ split} \\ T(1)+T(n-2)+\Theta(n) \text{ if } 1 : n-2 \text{ split} \\ \vdots \\ T(n-1)+T(0)+\Theta(n) \text{ if } n-1 : 0 \text{ split} \end{cases} \\ &= \sum_{k=0}^{n-1} X_k(T(k)+T(n-k-1)+\Theta(n)) \end{aligned}$$

Calculating expectation

$$\begin{aligned} E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k)+T(n-k-1)+\Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E\left[X_k(T(k)+T(n-k-1)+\Theta(n))\right] \\ &= \sum_{k=0}^{n-1} E\left[X_k\right] \cdot E[T(k)+T(n-k-1)+\Theta(n)] \end{aligned}$$

Note *Independence of X_k from other random choices.*

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Proof $E[T(n)] \leq an \lg n$ for constant $a > 0$.Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

Use fact:

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$$

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\ &\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n) \\ &= an \lg n - \left(\frac{an}{4} - \Theta(n)\right) \end{aligned}$$

Express as **desired – residual.**

if a is chosen large enough so that an/4 dominates the $\Theta(n)$.

4.5 Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.

Chapter 5 Sorting Lower Bounds

Introduction

- ❏ *Decision trees*
- ❏ *Counting sort*
- ❏ *Radix sort*
- ❏ *Appendix: Punched cards*

5.1 How fast can we sort?

All the sorting algorithms we have seen so far are **comparison sorts** :only use comparisons to determine the relative order of elements.

- **E.g.** insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.

5.2 Decision Tree

5.2.1 Decision-tree Example

Definition 5.1 (Decision-tree model)

A decision tree can model the execution of any comparison sort:

- *One tree for each input size n.*
- *View the algorithm as splitting whenever it compares two elements.*
- *The tree contains the comparisons along all possible instruction traces.*
- *The running time of the algorithm = the length of the path taken.*
- *Worst-case running time = height of tree.*

♣

5.2.2 Lower bound for decisiontree sorting

Theorem 5.1

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

♡

Proof The tree must contain $\geq n$! leaves, since there are n ! possible permutations. A height- h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

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

Corollary 5.1

Heapsort and merge sort are ***asymptotically optimal comparison sorting algorithms****.*

5.3 Sorting in linear time

5.3.1 Counting sort

No comparisons between elements.

- Input: $A[1 \dots n]$, where $A[j] \in \{1, 2, \dots, k\}$.
- Output: $B[1.. \ n]$, sorted.
- Auxiliary storage: $C[1 \dots k]$.

```
for i <- 1 to k
    do C[i] <- 0
for j <- 1 to n
    do C[A[j]] <- C[A[]] + 1 . //C[i] = |{key = i}|
for i <- 2 to k
    do C[i] <- C[i] +C[i-1]
for j <- n downto 1
    do B[C[A[j]]] <- A[j]
       C[A[j]] <- C[A[j]] - 1
```

Note ***Analysis***

$$\Theta(k) \begin{cases} \textit{for } i \leftarrow 1 \textit{ to } k \\ \textit{do } C[i] \leftarrow 0 \end{cases}$$

$$\Theta(n) \begin{cases} \textit{for } j \leftarrow 1 \textit{ to } n \\ \textit{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{cases}$$

$$\Theta(k) \begin{cases} \textit{for } i \leftarrow 2 \textit{ to } k \\ \textit{do } C[i] \leftarrow C[i] + C[i-1] \end{cases}$$

$$\Theta(n) \begin{cases} \textit{for } j \leftarrow n \textit{ downto } 1 \\ \textit{do } B[C[A[j]]] \leftarrow \mathrm{A}[j] \\ C[A[j]] \leftarrow C[A[j]] - 1 \end{cases}$$

5.3.1.1 Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Solution

- *Comparison sorting takes $\Omega(n \lg n)$ time.*
- *Counting sort is not a comparison sort.*
- *In fact, not a single comparison between elements occurs!*

Definition 5.2 (Stable sorting)

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

Exercise 5.1 What other sorts have this property?

5.3.2 Radix sort

Note

- *Origin: Herman Hollerith's card-sorting machine for the 1890 U.S. Census.*
- *Digit-by-digit sort.*
- *Hollerith's original (bad) idea: sort on most-significant digit first.*
- *Good idea: Sort on least-significant digit first with auxiliary stable sort.*

5.3.2.1 Operation of radix sort

5.3.2.2 Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.

5.3.2.3 Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base- 2^r digits.

Example 5.1 32-bit word $r=8 \Rightarrow b/r=4$ passes of counting sort on base- 2^8 digits; or $r=16 \Rightarrow b/r=2$ passes of counting sort on base-216 digits.

How many passes should we make?

Note ***Recall:*** *Counting sort takes $\Theta(n+k)$ time to sort n numbers in the range from 0 to $k-1$.*

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta\left(n+2^r\right)$ time. Since there are b/r passes, we have

$$T(n,b)=\Theta\left(\frac{b}{r}\left(n+2^r\right)\right).$$

Choose r to minimize $T(n,b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.Minimize $T(n,b)$ by differentiating and setting to 0 .Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.Choosing $r=\lg n$ implies $T(n,b)=\Theta(bn/\lg n)$.
- For numbers in the range from 0 to n^d-1, we have $b=d\lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.

5.4 Colclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example 5.2

1. At most 3 passes when sorting ≥ 2000 numbers.
2. Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Note ***Downside:*** *Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.*