Solutions to
"Introduction to Algorithms, 3rd edition"

Jian Li
(yinyanghu)

June 9, 2014

Acknowledgements

Contents

Part I

Foundations

Chapter 1

The Role of Algorithms in Computing

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\log(n)$	2^{10^6}	$2^{10^6 \cdot 60}$	$2^{10^6 \cdot 60 \cdot 60}$	$2^{10^6 \cdot 60 \cdot 60 \cdot 24}$	$2^{10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 30}$	$2^{10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365}$	$2^{10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365 \cdot 100}$
$\sqrt{N}$	$(10^6)^2$	$(10^6 \cdot 60)^2$	$(10^6 \cdot 60 \cdot 60)^2$	$(10^6 \cdot 60 \cdot 60 \cdot 24)^2$	$(10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 30)^2$	$(10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365)^2$	$(10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365 \cdot 100)^2$
n	10^6	$10^6 \cdot 60$	$10^6 \cdot 60 \cdot 60$	$10^6 \cdot 60 \cdot 60 \cdot 24$	$10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 30$	$10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365$	$10^6 \cdot 60 \cdot 60 \cdot 24 \cdot 365 \cdot 100$
$n\log(n)$	$62,746$	$2.8 \cdot 10^6$	$1.33 \cdot 10^8$	$2.75 \cdot 10^9$	$7.18 \cdot 10^{10}$	$7.97 \cdot 10^{11}$	$6.86 \cdot 10^{13}$
n^2	$1,000$	$7,746$	$60,000$	$293,939$	$1.6 \cdot 10^6$	$5.6 \cdot 10^6$	$5.6 \cdot 10^7$
n^3	100	391	$1,533$	$4,421$	$13,737$	$31,594$	$146,646$
2^n	20	26	32	36	41	45	51
$n!$	$(9,10)$	$(11,12)$	$(12,13)$	$(13,14)$	$(15,16)$	$(16,17)$	$(17,18)$

Table 1.1: Solution to Problem 1.1

1.1 Comparison of running times

Table 1.1 shows the solution. We assume the base of $\log(n)$ is 2. And we also assume that there are 30 days in a month and 365 days in a year.

Note Thanks to Valery Cherepanov(Qumeric) who reported an error in the previous edition of solution.

Chapter 2

Getting Started

2.1 Insertion sort on small arrays in merge sort

2.1.1 a

The insertion sort can sort each sublist with length k in $\Theta(k^2)$ worst-case time. So sorting all n/k sublists could be completed in $\Theta(k^2 \cdot n/k) = \Theta(nk)$ worst-case time.

2.1.2 b

Naive We could easily find a naive method. Let us try to think n/k sublists as n/k sorted queues. We scan all head elements of n/k queues, and find the smallest element, then pop it from the queue. The running time of each scan is $\Theta(n/k)$. And we need pop all n elements from n/k queues. So this naive method costs $n \cdot \Theta(n/k) = \Theta(n^2/k)$ time.

Heap Sort *If you do not know what the Heap Sort is, you could temporarily skip this method before you read* ***Chapter 6: Heapsort****.*

Similarly, we could use a min-heap to maintain all head elements. There are at most n/k elements in the heap, so each *INSERT* and *EXTRACT-MIN* operation takes $\mathcal{O}(\log(n/k))$ worst-case time. And every element enters and leaves the heap just once. Therefore, the overall worst-case running time is $n \cdot \mathcal{O}(\log(n/k)) = \mathcal{O}(n \log(n/k))$.

Merge Sort We could use the same procedure in Merge Sort, except the base case is a sublist with k elements instead. We get the recurrence

$$T(m) = \begin{cases} \Theta(1) & \text{if } m \leq k \\ 2T(m/2) + \Theta(m) & \text{otherwise} \end{cases}$$

Draw a **recursion tree**, and get the result

$$\begin{aligned} T(n) &= 1/2 \cdot n/k \cdot 2k + 1/4 \cdot n/k \cdot 4k + \cdots + n \\ &= n \log(n/k) \end{aligned}$$

Therefore, the worst-case running time is $\Theta(n \log(n/k))$.

2.1.3 c

The largest value of k is $\Theta(\log(n))$. The running time is $\Theta(nk+n\log(n/k)) = \Theta(n\log(n) + n\log(n/\log(n))) = \Theta(n\log(n))$, which has the same running time as standard merge sort.

2.1.4 d

Since k is the length of the sublist, we should choose the largest k that Insertion Sort can sort faster than Merge Sort on the list with length k.

In practice, Timsort, a hybrid sorting algorithm, use the exactly same idea with some complicated techniques.

2.2 Correctness of bubblesort

2.2.1 a

We also need to prove that A' is a permutation of A.

2.2.2 b

Lines 2-4 maintain the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2-4, $A[j]$ is the smallest element of $A[j..A.length]$. Moreover, $A[j..A.length]$ is a permutation of the initial $A[j..A.length]$.

Initialization Prior to the first iteration of the loop, we have $j = A.length$, so that the subarray $A[j..A.length]$ have only one element, $A[A.length]$. Trivially, $A[A.length]$ is the smallest element as well as a permutation of itself.

Maintenance To see that each iteration maintains the loop invariant, we assume that $A[j]$ is the smallest element of $A[j..A.length]$. For next iteration(decrementing j), if $A[j-1] < A[j]$, i.e. $A[j-1]$ is the smallest element of $A[j-1..A.length]$, we have done and skip lines 3-4. Otherwise, lines 3-4 perform the exchange action to maintain the loop invariant. Also, it is still a valid permuation, since we only exchange two adjacent elements.

Termination At termination, $j = i$. By the loop invariant, $A[i]$ is the smallest element of $A[i..A.length]$ and $A[i..A.length]$ is a permutation of the initial $A[i..A.length]$.

2.2.3 c

Lines 1-4 maintain the following loop invariant:

> At the start of each iteration of the **for** loop lines 1-4, the subarray $A[1..i-1]$ contains the smallest $i-1$ elements of the initial array $A[1..A.length]$. And this subarray is sorted, i.e. $A[1] \leq A[2] \leq \cdots \leq A[i-1]$.

Initialization Initially, $i = 1$, i.e. $A[1..i-1]$ is empty. The loop invariant trivially holds.

Maintenance By loop invariant, $A[1..i-1]$ contains the smallest $i-1$ elements and it is sorted. And lines 2-4 perform the action to move the smallest element of the subarray $A[i..A.length]$ into $A[i]$. So incrementing i reestablishes the loop invariant for the next iteration.

Termination At termination, $i = A.length$. By the loop invariant, the subarray $A[1..A.length-1]$ contains the smallest $A.length-1$ elements. Also, this subarray is sorted. So the element $A[A.length]$ must be the largest element and the array $A[1..A.length]$ is sorted.

2.2.4 d

The worst-case running time of Bubble Sort is $\Theta(n^2)$, which is the same as Insertion Sort.

2.3 Correctness of Horner's rule

2.3.1 a

The running time is $\Theta(n)$.

2.3.2 b

Naive-Polynomial-Evaluation shows the pseudocode of naive polynomial-evaluation algorithm. The running time is $\Theta(n^2)$.

```
NAIVE-POLYNOMIAL-EVALUATION(P(x), x)
1  y = 0
2  for i = 0 to n
3        t = 1
4        for j = 1 to i
5              t = t · x
6        y = y + t · a_i
7  return y
```

2.3.3 c

Initialization Prior to the first iteration of the loop, we have $i = n$, so that $\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = \sum_{k=0}^{-1} a_{k+n+1} = 0$ consistent with $k = 0$. So loop invariant holds.

Maintenance By loop invariant, we have $y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$. Then lines 2-3 perform that

$$
\begin{aligned}
y' &= a_i + x \cdot y \\
&= a_i + x \cdot (\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k) \\
&= a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^{k+1} \\
&= \sum_{k=0}^{n-i} a_{k+i}x^k
\end{aligned}
$$

So decrementing i reestablishes the loop invariant for the next iteration.

Termination At termination, $i = -1$. By loop invariant, we get the result $y = \sum_{k=0}^{n} a_k x^k$.

2.3.4 d

The given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \cdots, a_n$, i.e.

$$y = \sum_{k=0}^{n} a_k x^k = P(x)$$

2.4 Inversions

2.4.1 a

(1, 5), (2, 5), (3, 5), (4, 5), (3, 4)

2.4.2 b

Array $\langle n, n-1, n-2, \cdots, 1\rangle$ has $\binom{n}{2} = n(n-1)/2$ inversions.

2.4.3 c

The running time of Insertion Sort and the number of inversions in the input array are exactly same, since each move action in Insertion Sort eliminates exact one inversion.

2.4.4 d

We could modifiy the Merge Sort algorithm to count the number of inversions in the array. The key point is that if we find $L[i] > R[j]$, then each element of $L[i..]$(represent the subarray from $L[i]$) would be as an inversion with $R[j]$, since array L is sorted.

COUNTING-INVERSIONS and *INTER-INVERSIONS* shows the pseudocode of this algorithm.

COUNTING-INVERSIONS($A, left, right$)
$inversions = 0$
if $left < right$
    $mid = \lfloor (left + right)/2 \rfloor$
    $inversions = inversions +$ COUNTING-INVERSIONS($A, left, mid$)
    $inversions = inversions +$ COUNTING-INVERSIONS($A, mid + 1, right$)
    $inversions = inversions +$ INTER-INVERSIONS($A, left, mid, right$)
return *inversions*

```
INTER-INVERSIONS(A, left, mid, right)
n1 = mid - left + 1
n2 = right - mid
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[left + i - 1]
for i = 1 to n2
    R[i] = A[mid + i]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
inversions = 0
counted = FALSE
for k = left to right
    if counted = FALSE and L[i] > R[j]
        inversions = inversions + n1 - i + 1
        counted = TRUE
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
        counted = FALSE
return inversions
```

We can call *COUNTING-INVERSIONS(A, 1, n)* to get the number of inversions in the array A. The worst-case running time is the same as Merge Sort, i.e. $\Theta(n \log(n))$.

Part II

Sorting and Order Statistics

Part III

Data Structures

List of Figures

List of Tables