
Author: Parth Shah

Title: Chapter 2 Solutions

Notes

Blah blah blah

2.1 Insertion Sort

Problem 2.1-1 Show how insertion sort works on the sequence $A = \{31, 41, 59, 26, 41, 58\}$

Solution 2.1-1

Problem 2.1-2 Rewrite insertion sort to sort in nondecreasing order.

Solution 2.1-2

Problem 2.1-3 Consider searching problem. Given sequence $A =< a_1, a_2, a_3, a_4, ...a_n >$ return index i where value x is located in A. If x is not in A return NIL

Solution 2.1-3

Problem 2.1-4 Write an algorithm to add 2 n-bit integers A and B to create (n+1)-bit integer C.

Solution 2.1-4

2.2 Analyzing Algorithms

Problem 2.2-1 Express function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ notation.

Solution 2.2-1

Problem 2.2-2 Selection sort goes in increasing order from 1 to $n-1$ and finds the i^{th} smallest number and swaps it with $A[i]$. What loop invariant does selection sort maintain? Why does it only need to run for the first $n-1$ elements? Best and worst case run-time?

Solution 2.2-2

Problem 2.2-3 In linear search on average how many elements are checked? Worst case? Asymptotic notation for average and worst case?

Solution 2.2-3 Since the element is equally likely to be found at any index, the average number of elements looked is simply the expected index the element is at. This is $\boxed{(n+1)/2}$. Worst case it looks through all $\boxed{n}$ elements. In asymptotic notation average case is $\boxed{\Theta(n)}$ and worst case is $\boxed{\Theta(n)}$ as well.

Problem 2.2-4 How can we modify any algorithm to have good best case running time?

Solution 2.2-4 Check if input already satisfies the output or some simple condition and then return it. Example in any sorting method check if its already in order and if so return the input. Otherwise sort.

2.3 Designing Algorithms

Problem 2.3-1 Illustrate merge sort operation on $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

Solution 2.3-1

Problem 2.3-2 Rewrite the merge procedure such that it simply copies the remaining array into A once either L or R are completely copied into A.

Solution 2.3-2

Problem 2.3-3 Use induction to show that when n is a power of two the solution to the following recurrence $T(n)$ is $T(n) = n \lg(n)$

$$T(n) = \begin{cases} 2 & n = 2 \\ 2T(n/2) + n & n = 2^k, k > 1 \end{cases}$$

Solution 2.3-3

Problem 2.3-4 Insertion sort can be expressed recursively. Sort $A[1...n-1]$ and then insert $A[n]$ in proper spot. Write the recurrence for this description of insertion sort.

Solution 2.3-4 The recurrence is $T(n) = T(n-1) + \Theta(n)$.

Problem 2.3-5 Write pseudocode for binary search method on a sorted array. Show that worst case is $\Theta(\lg(n))$

Solution 2.3-5

Problem 2.3-6 Insertion sort uses linear scan to find location for each new entry. Can a binary search improve the run time to $\Theta(n \lg(n))$

Solution 2.3-6 No. The key is in swapping. While finding the location to insert it into would be sped up to $O(\lg(n))$ there is no way around the $O(n)$ swaps required to get the new entry into the proper location.

Problem 2.3-7 Describe a $\Theta(n \lg(n))$ solution that given a set S of n integers and a value x finds if a pair of integers in S that sum to x.

Solution 2.3-7 First sort S using merge sort in $\Theta(n \lg(n))$ time. Then for each index i from 1 to n binary search for $x - A[i]$ in sorted array A. If it exists at index j and $j \neq i$ return (i, j). Otherwise return no solution.

Problems

Problem 2-1 Sometimes insertion sort is faster on smaller sequences due to constants involved in the runtime. So often insertion sort is used as a subroutine on the merge sort recursion when the size of the array to be sorted is less than or equal to some size k. Consider a merge sort in which n/k sublists of size k are sorted by insertion sort and then are merged using the standard merge sort procedure.

a) Show that the insertion sort portion runs in $\Theta(nk)$.

Solution

b) Show that the merge of the n/k sublists can be done in $\Theta(n \lg(n/k))$

Solution

c) The new algorithm runs in $\Theta(nk + n \lg(n/k))$. What value of k gives the same runtime as merge sort.

Solution

d) How should you choose k in practice.

Solution Try multiple values of k on multiple values of n. In fact you can binary search and test on values of k. To speed up the search set a small constant lower bound and an upper bound on order $\lg(n)$. Usually a good k may be the size of a block in your cache.

Problem 2-2

```
1: procedure BUBBLESORT(A)
2:     for i = 1 to A.length do
3:         for j = A.length downto i + 1 do
4:             if A[j] < A[j − 1] then
5:                 swap A[j] and A[j − 1]
```

a) To prove Bubblesort is correct we need to show it terminates and that when it does it is in nondecreasing order. What else do we need to show in order to say that bubblesort actually sorts.

Solution We also need to show that values are not somehow changed. The output has the same values as the input.

b) State a loop invariant for lines 2-4 and show it holds.

Solution The invariant is that the minimum $A[k]$ seen is at index j. Prove by induction. Initially only 1 element is seen and therefore it is minimum. It is at index j. If this holds true for $j = k + 1$ we can show it is true for $j = k$. If $A[k]$ is less than $A[k + 1]$ order remains otherwise they are swapped. Since we are taking the minimum of the two and $A[k + 1]$ is the minimum of all previously seen, the invariant is clearly maintained.

c) State a loop invariant for the entire bubble sort.

Solution At the end of each iteration of the for loop all elements $A[1]$ through $A[i]$ are sorted. Prove by induction. This is clearly true in the beginning as 0 elements are sorted. Given the first $i - 1$ elements are sorted we can show after the next iteration i elements will be sorted. Based on the solution to part b, we know that the minimum of all elements in indices i to n will be placed into $A[i]$. Since the elements $A[1]$ to $A[i - 1]$ is less than $A[i]$ the first i elements are now sorted. Therefore the invariant is maintained.

d) Give worst case runtime for bubble sort.

Solution Worst case requires $n - i$ swaps for each iteration of the outer for loop. Summing over all i this gives $n(n - 1)/2$ and hence asymptotic time of $\boxed{\Theta(n^2)}$.

Problem 2-3 Horner's rule is a way to quickly evaluate a polynomial. Instead of evaluating all x^k, Horner's rule recursively multiplies by x and adds a coefficient. By Horner's rule, $P(x) = a_0 + x(a_1 + x(a_2 + ... + xa_n)...)$. Algorithmically it looks as follows:

```
1: y = 0
2: for i = n downto 0 do
3:     y = a_i + x · y
```

a) In terms of Θ-notation what is the runtime.

Solution

b) Write pseudocode for naive implementation of polynomial evaluation. What is the runtime?

Solution

c) Assume invariant is that of component up to degree k is evaluated correct. Prove invariant holds.

Solution

d) Conclude the method is correct.

Solution

Problem 2-4 Inversion are as such. In an array A for indices i and j such that $i < j$, if $A[i] > A[j]$ there is an inversion.

a) List the five inversions of $\{2, 3, 8, 6, 1\}$

Solution (2,1) (3,1) (8,1) (6,1) (8,6)

b) What array from set $\{1, 2, ..., n\}$

Solution $\{n, n-1, ..., 2, 1\}$. This has $\sum_{i=1}^{n-1} i$ inversions which is equal to $n(n-1)/2$.

c) What is the relationship between runtime of insertion sort and the number of inversions.

Solution The runtime is equivalent to the number of inversions. For each inversion insertion sort will need to make one swap between those two elements.

d) Give a method to calculate the number of inversions in $\Theta(n \lg(n))$ time.

Solution Use merge sort except modify the merge method. Keep a counter. Every time any merge method takes an element from R before L add the remaining size of R to the counter. This counter is global and at the end the counter is the number of inversions.
