# Algorithm

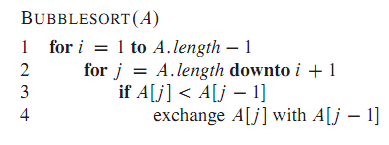
# Homework 2

## CHANG LIU

## chang.liu@jhu.edu

# Mar. 7, 2012

### 1 Correctness of BUBBLE SORT



To prove the correctness of bubble sort, we need first to prove that the loop invariant holds for the loop from line 2 to line 4:

**Statement**: At the start of each iteration of the **for** loop from line 2 – 4, the subarray A[j] stores the smallest element in A[j..n].

**Initialization**: We start by showing that the loop invariant holds before the first loop iteration, when i = 1. The subarray A[n .. n], therefore, consists of just the single element A[j = n], which is the smallest element in the subarray.

**Maintenance**: Before the jth iteration, A[j] stores the small least elements among A[j .. n]. The body of the **for** loop works by moving the smaller element of A[j] and A[j - 1] to the A[j - 1] (line 3 – 4), thus the A[j - 1] then consists the smallest element among A[j-1 .. n]. Decrease j for the next iteration of the for loop then preserves the loop invariant.

**Termination**: The condition that causing the **for** loop to terminate is that j < i+1. Because each loop iteration decrease j by 1, we must have j = i + 1 at the end of the loop. Then we have j – 1 = i, thus A[i] = A[j - 1] stores the smallest elements in the subarray A[i .. n] (A[j .. n]).

Then we prove the loop invariant holds for the outer loop from line 1 to line 4:

**Statement**: At the start of each iteration of the for loop of lines 1 – 4, the subarray A[1 .. i-1] has the smallest (i-1) elements in the original array.

**Initialization**: when i = 1, The subarray A[1 .. 0] is an empty array, so it is trivially sorted.

**Maintenance**: Before the ith iteration, A[1 .. i-1] has the smallest (i-1) elements in the original array. The body of the **for** loop works by moving the smallest element among A[i .. n] to the A[i] (line 2 – 4), thus the A[1 .. i] then consists the smallest (i) elements among the original A. increase i for the next iteration of the for loop then preserves the loop invariant.

**Termination**: The condition that causing the **for** loop to terminate is that i > n-1. Because each loop iteration increase i by 1, we must have i = n-1 at the end of the loop. Thus we have the A[1 .. n-1] contains the smallest (n-1) elements in the original array. Then A[n] automatically becomes the biggest element in the original array. Observing that the subarray A[1 .. n] is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

### 2 Comparing Sorting Algorithm

|  |  |  |
| --- | --- | --- |
| **Sorting Algorithm** | **Advantage** | **Disadvantage** |
| Quick Sort | * Fast algorithm | * Complicated algorithm * Choosing pivot element, a wrong choice of pivot element may result in slower performance |
| Insertion Sort | * Simple and easy to use * Efficient on smaller data and already substantially sorted data * Low memory requirements | * Inefficient for large lists |
| Bubble Sort | * Simple algorithm * Stable sorting algorithm | * Inefficient * Slow in sorting. Although large elements at the beginning of the list are quickly swapped but for small elements towards the end move to the beginning very slowly. |
| Bucket Sort | * Efficient when n < k * Better performance than bubble sort | * Requires more memory * Performance tends to degrades with clustering values, these values will fall into a single bucket and sort slowly |
| Merge Sort | * Runs faster in the worst case | * It is a complex algorithm * It requires at least twice the memory requirements as the other sorting algorithm |

### 3 Finding the Number of Distinct Array Elements

### 4 Exercise 2.1-3

Searching-Value (A, v)

1 for i = 1 to A.length

2 if A[i] = v

3 return i

4 return NIL

Prove the correctness by induction:

### 5 Exercise 2.3-5

Binary-Search (A, v, h, t)

1 if h > t

2 return NIL

3 j =

4 if v == A[j]

5 return j

6 if v < A[j]

7 return Binary-Search(A, v, h, j)

8 if v > A[j]

9 return Binary-Search(A, v, j, t)

As this Binary Search is a recursive version, and the array is sorted, based on the comparison of v to the middle element in the searched range, the search continues with the range halved and terminate when value v is found. Thus the recurrence is T(n) = T(n/2) + (1), and we can calculate the T(n) = .

### 6 Exercise 2.3-6

We can use the binary search instead in improve the overall worst-case running time of insertion sort to This is because what we are doing in the while loop from line 5 to line 7 is to insert the new element into the already sorted subarray, thus we can apply a binary search as stated in the description of Exercise 2.3-5. Thus the inner performance will be after replace the while loop with the recursive expression. Moreover, the outer loop repeat the job for n times, thus we have the total running time of

### 7 Problem 2-4

**a. List the five inversions of the array <2, 3, 8, 6, 1>**

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

**b.**

The array has the most inversions: {n, n-1, n-2, … , 2, 1}

Thus we have inversion (n-1) + (n-2) + … + 1 = n(n-1)/2

**c.**

### 8 Problem 4-1

1. 

a = 2, b = 2, f(n) = n4, thus we have:



Case 3 should apply if we show that the regularity condition holds for f(n).

For sufficient large n, we have that



Thus apply case 3, we have

