Algorithm

Homework 2

CHANG LIU

chang.liu@jhu.edu

Mar. 7, 2012

1 Correctness of BUBBLE SORT

```
BUBBLESORT($A$)
1  for $i = 1$ to $A.length - 1$
2      for $j = A.length$ downto $i + 1$
3          if $A[j] < A[j-1]$
4              exchange $A[j]$ with $A[j-1]$
```

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 j^{th} 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 i^{th} 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 ● Best-case run time O(nlogn) ● Average run time O(nlogn)	● Complicated algorithm ● Choosing pivot element, a wrong choice of pivot element may result in slower performance ● Worst-case run time O(n^2)
Insertion Sort	● Simple and easy to use ● Efficient on smaller data and already substantially sorted data ● Low memory requirements ● Best-case run time O(n) ● Average run time O(n^2)	● Inefficient for large lists ● Worst-case run time O(n^2)
Bubble Sort	● Simple algorithm ● Stable sorting algorithm ● Best-case run time O(n^2) ● Average run time O(n^2)	● 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. ● Worst-case run time O(n^2)
Bucket Sort	● Efficient when $n < k$ ● Better performance than bubble sort ● Best-case run time O(n) ● Average run time O(n)	● Requires more memory ● Performance tends to degrades with clustering values, these values will fall into a single bucket and sort slowly ● Worst-case run time O(n^2)
Merge Sort	● Runs faster in the worst case ● Best-case run time O(nlogn) ● Average run time O(nlogn)	● It is a complex algorithm ● It requires at least twice the memory requirements as the other sorting algorithm ● Worst-case run time O(nlogn)

3 Finding the Number of Distinct Array Elements

```
Distinct-Number (A)
for i = 1 to A.length
    Tag = 0
    for j = i+1 to A.length
        if (A[j] == A[i])
            remove A[j] from the array
            tag = 1
    If (tag == 1)
        Remove A[i] from the array
return A.length
```

For each inner loop, we remove the elements if we find they are same to the A[i], which is the key element in the outer loop. Then when finished the inner loop, we remove the A[i] if tag == 1, which means duplicate found in the array with the same value of A[i]. Thus, for each iteration of the whole algorithm, we try to find if there’s duplicated elements has the same value of A[i], if there is, we remove all of them from the array. Thus the final array contains only distinct value from the original array, we can then return the length of the final array as the number of distinct elements in the original array.

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:

When A.length = 1, only 1 element are there in A. Thus after comparison, if the only element equal to v, return 1, else return NIL.

When A.length = n, assume the algorithm holds, it will return i which A[i] = v.

When A.length = n+1, we have 3 cases:
case 1: when A[i] in A[1..n] and A[i] = v, the algorithm will return I;
case 2: when there is no element in A[1..n] equals v, and A[i+1] = v, then return i+1;
case 3: when there is no element in A[1..n] equals v, and A[i+1] != v, then returns NIL.
Thus the algorithm holds for A.length = n + 1.

5 Exercise 2.3-5

```
Binary-Search (A, v, h, t)
1   if h > t
2       return NIL
3   j = $\lfloor A.\text{length}/2 \rfloor$
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 $T(n) = T(n/2) + \Theta(1)$, and we can calculate the $T(n) = \Theta(\lg n)$.

6 Exercise 2.3-6

We can use the binary search instead in improving the overall worst-case running time of insertion sort to $\Theta(\mathrm{nlgn})$. 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 $\Theta(\mathrm{lgn})$ 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$\Theta(\mathrm{n}) * \Theta(\mathrm{lgn}) = \Theta(\mathrm{nlgn})$.

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.

What the insertion sort algorithm actually does is to pick up the element one by one from the original array and put them into the subarray in a sorted order. Suppose that the array A starts out with an inversion (k, j), then k < j and A[K] > A[j]. At the time that the outer for loop of lines 1-8 sets key=A[j], the value that started in A[k] is still left to the A[j]. That is, it is in A[i], where i<=i<j, and so the inversion becomes (i, j). When in the while loop, as the algorithm only moves the elements that are less than key, it moves only elements that correspond to inversions. That is, each iteration of the while loop in the algorithm corresponds to the elimination of one inversion.

d.

```
Count-Inversions(A, p, r)
Inversions = 0
If p < r
    q = j = $\lfloor (p + r)/2 \rfloor$
    inversions = inversions + Count-Inversions(A, p, q)
    inversions = inversions + Count-Inversions(A, q+1, r)
    inversions = inversions + Merge-Inversions(A, p, q, r)
return inversions
Merge-Inversions(A, p, q, r)
$n_1$ = q – p + 1
$n_2$ = r – q
let L[1 .. $n_1$+1] and R[1 .. $n_2$+1] be the new arrays
for i = 1 to $n_1$
    L[i] = A[p+i-1]
For j = 1 to $n_2$
    R[j] = A[q+j]
L[$n_1$+1] = $\infty$
R[$n_2$+1] = $\infty$
i = 1
j = 1
inversions = 0
counted = FALSE
for k = p to r
    if counted == FALSE and R[j] < L[i]
```

```
        inversions = inversions + $n_1$ – 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 invresions
```

The initial call is Count-Inversions(A, 1, n)

In MERGE-INVERSIONS, the boolean variable counted indicates whether we have counted the merge-inversions involving R[j]. We count them the first time that both R[j] is exposed and a value greater then R[j] becomes exposed in the L array. We set counted to FALSE upon each time that a new value becomes exposed in R. We don't have to worry about merge-inversions involving the sentinel ∞ in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last for loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.

8 Problem 4-1

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

a = 2, b = 2, f(n) = n^4, thus we have:

$n^{\log_b^a} = n^{\log_2^2} = \Theta(n)$

$\rightarrow f(n) = \mathrm{O}(n^{\log_2^2 + 3})$

Case 3 should apply if we show that the regularity condition holds for f(n).
For sufficient large n, we have that

$af(n/b) = 2f(n/2) = 2(n/2)^4 \le \dfrac{1}{2^2} n^4 = cn^4$

$\rightarrow c = \dfrac{1}{2^2}$

Thus apply case 3, we have

$T(n) = \Theta(f(n)) = \Theta(n^4)$

b. $T(n) = T(7n/10) + n$

a = 1, b = 10/7, thus we have:

$n^{\log_b^a} = n^{\log_{1.43}^1} = \Theta(n^0) = \Theta(1)$

$\rightarrow f(n) = \mathrm{O}(n^{0+1}) \rightarrow \varepsilon = 1$

Case 3 should apply if we show that the regularity condition holds for f(n).
For sufficient large n, we have that

$af(n/b) = f(7n/10) = \dfrac{7}{10} n \le \dfrac{8}{10} n = cf(n)$

$\rightarrow c = \dfrac{8}{10}$

Thus apply case 3, we have

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

c. $T(n) = 16T(n/4) + n^2$

a = 16, b = 4, f(n) = n^2, thus we have:

$n^{\log_b^a} = n^{\log_4^{16}} = \Theta(n^2)$

$\rightarrow f(n^2) = \mathrm{O}(n^{\log_4^{16}})$

Thus case 2 apply, $f(n) = \Theta(n^2 \lg n)$

d. $T(n) = 7T(n/3) + n^2$

a = 7, b = 3, f(n) = n^2, thus we have:

$n^{\log_b^a} = n^{\log_3^7} = \Theta(n^{1.77})$

$\rightarrow f(n^2) = \mathrm{O}(n^{\log_3^7 + 0.23}) \rightarrow \varepsilon \approx 0.23$

Thus case 3 should apply if we can show that the regularity condition holds for f(n).
For sufficient large n, we have that

$$af(n/b) = 7f(n/3) = 7(n/3)^2 \leq \frac{7}{9}n^2 = cf(n)$$

$$\rightarrow c = \frac{7}{9}$$

Thus apply case 3, we have

$T(n) = \Theta(f(n)) = \Theta(n^2)$

e. $T(n) = 7T(n/2) + n^2$

a = 7, b = 2, f(n) = n^2, thus we have:

$n^{\log_b^a} = n^{\log_2^7} = \Theta(n^{2.81})$

$\rightarrow f(n^2) = \mathrm{O}(n^{\log_2^7 - 0.81}) \rightarrow \varepsilon \approx -0.81$

Thus case 1 apply, we have

$T(n) = \Theta(f(n)) = \Theta(n^{\log_b^a}) = \Theta(n^{\log_2^7})$

f. $T(n) = 2T(n/4) + \sqrt{n}$

a = 2, b = 4, f(n) = $\sqrt{n}$, thus we have:

$n^{\log_b^a} = n^{\log_4^2} = \Theta(n^{0.5})$

$\rightarrow f(n^{0.5}) = \mathrm{O}(n^{\log_4^2})$

Thus case 2 apply, we have

$T(n) = \Theta(n^{\log_b^a} \lg n) = \Theta(n^{0.5} \lg n)$

g. $T(n) = T(n-2) + n^2$

$T(n) = T(n-2) + n^2$
$= n^2 + (n-2)^2 + (n-4)^2 + \cdots + T(1)$

Assume that T(1) = 0, we have

$T(n) = T(n-2) + n^2$
$= n^2 + (n-2)^2 + (n-4)^2 + \cdots + T(1)$
$= 0^2 + 2^2 + 4^2 + \cdots + n^2$
$= \frac{3}{2} n(n+1)(2n+1)$
$\Rightarrow \Theta(n^3)$