EE602: ALGORITHMS

HOMEWORK #1 **Due: 9/07**

HUALIANG Li

(1) Exercises 2.1-3

Pseudocode:

```
Search(A[], v):
        For i = 0 to A.length-1
                If A[i] == v
                        Return i
        Return NIL
```

Proof:

Initialization:

Initially the subarray is the empty array, the pseudocode will return NIL, which is trivial.

Maintenance:

Before each iteration, we know that A[0…i-1] does not contain v. We compare A[i] with v, if they are equal, we return the index. Otherwise we continue to the next iteration. So, this step reserves the invariant.

Termination:

The loop terminates when i > A.length -1. At this point, we know that A[0…i] does not contain v. Thus, we return NIL, which is correct.

(2) Exercises 2.2-2

Pseudocode:

```
Selection_Sort(A[]):
    For i = 0 to A.length -2
            Min index = i
            For j = i + 1 to A.length – 1
                    If A[j] < A[min index]
                            Min index = j
            swap A[i] and A[min index]
```

Loop invariant:

1. At the beginning of each iteration of the outer for loop, A[0…i-2] is a sorted non-increasing array which contains the most smallest i-1 elements of the original array. 2. At eh beginning of each iteration of the inner for loop, A[min index] is the smallest element in the subarray A[i…j-1].

Why only n – 1 elements:

In the final iteration of the outer loop, the algorithm will be left with two elements, which will store the smaller one in A[n-2], the larger one will be in A[n-1]. There is no need to do one more step. Otherwise, we will have to do a redundant step which needs us to sort a single element.

Notation:

The best case is when the inner for loop if statement never been invoked. So, the number of operation is:

$$(n - 1)((n-2)/2)$$

The worst case is when the inner for loop if statement happens to be invoked every time. So, the number of operation is:

$$(n-1)(n-2)$$

Both of the cases is (n^2).

(3) Exercises 2.3-5

```
Binary_Search(A, v):
    Low = 1
    Hight = A.length
    While low <= high:
            mid = (hight + low)/2
            if mid == v
                    return mid
            else if A[mid] < v
                    low = mid + 1
            else high = mid – 1
    return NIL
```

It is easy to see that T(n+1) = T(n/2) + c. It is trivial to see that the worst running time is when the algorithm exhaust all the iteration and find the target at the last one. So, it is easy to guess that running time is (lgn).

(4) Exercises 2.3-4. To what, asymptotically, does the recurrence evaluate?

The recurrence is:

T(n) = theta(1), if n=1; T(n) = T(n-1) + C(n-1), if n > 1. Where C(n) is the time to insert an element into a sorted array of n elements.

The recurrence at iteration i evaluates to a sorted array0…i-1.

(5) Exercises 2.3-6

No. Even if it finds the position in logarithmic time, it still needs to shift all elements after it to the right, which is linear in the worst case. It will perform the same number of swaps, although it would reduce the number of comparisons.

(6) A graph T is said to be a tree if T is connected and has no cycles. Show by induction that if T is a tree with n vertices, then T has exactly n-1 edges.

Caution: When proving the induction step (if true for a tree of n nodes then true for a tree of n+1nodes), make sure to argue that your proof will work for an *arbitrary* tree of n+1 nodes.

Proof:
Base Case: Let n = 1, this is a trivial tree with 1-1 = 0 edge. So, it is true for n = 1.
Induction step: Let n = j, and assuming it is true for k. That is, every arbitrary tree with k vertices has exactly k-1 edges. Let T be a tree with k+1 vertices, and let v be a leaf of tree T. So, if we delete leaf v from tree T, it will become a tree T' with k vertices, therefor it has k-1 edges. Leat v must have a degree of 1, because it is a leaf, so, tree T must have one more edge than tree T', which is k edges.

(7) Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of Θ notation, prove that max(f(n); g(n)) = is Θ (f(n) + g(n)).

Proof:
Let max denote max(f(n), g(n)),
We need to find positive constants c1, c2,
such that 0<=c1(f(n) + g(n)) <= max <= c2(f(n) + g(n)) for all n >= n0.
so, it is easy to see that max >= f(n) and max >= g(n). Therefore:
max + max >= f(n) + g(n)
so, max >= (1/2)*(f(n) + g(n)) that is we find c1 = 1/2;
since f(n) and g(n) are nonnegative functions, we have max <= f(n) + g(n)
that is, we find c2 = 1.
Therefore, according to the basic definition of Θ notation, we have
max(f(n); g(n)) = is Θ (f(n) + g(n)).

(8) For each pair of functions (f(n); g(n)) below, you must state and prove the strongest possible relationship between them, in terms of say whether f(n) = O(g(n)), or f(n) = Ω(g(n)), or f(n) = Θ(g(n)), or none of these. You must supply a clear justification or proof for each pair.

(a) 2n and n!;

2n =O(n!). Because n! = n*n-1*… 3 * 2 * 1,

so n! = 2*n *(n-1 *n-2 *… 4 * 3 *1) >= 2n when n > 1. Therefore, 2n has upper bound n!.

(b) $6^{\lg n}$ and $5^{\lg n}$;

$6^{\lg n} = \Omega(5^{\lg n})$. Because 6^m > 5^m when m > 0. If n > 1, lgn > 0. Therefore, $6^{\lg n} > 5^{\lg n}$ when n> 1.

(c) $n^{1/2}$ and $n^{\sin n}$;

none of these. Because they will intersected with each other oscillately as n grows.

(d) $(n!)^n$ and $(2^2)^n$

$(n!)^n = \Omega((2^2)^n)$. Because n! > 2^2 when n >2, so, $(n!)^n > (2^2)^n$ when n>2.

(9) Problem 2.4

Five inversions:
<8, 6>, <2, 1>, <3, 1>, <8, 1>, <6, 1>
Array with most inversions:
That would be an array sorted in decreasing order, which is <n, n-1, n-2, …, 1>.
N = (n-1) + (n-2) + (n-3) + … + 2 + 1 = ((n-1) + 1) * (n – 1) / 2 = n(n-1)/2
Relation with insertion sort:
The running time of insertion sort is same as the number of the inversions. This is because in the insertion sort, we are basically moving elements that are in a situation of inversion, which is exact number of the inversion. Therefore, it is same.
Algorithm that count inversion in (nlgn):

```
MERGE-SORT(A, p, r):
 if p < r
    inversions = 0
    q = (p + r) / 2
    inversions += merge_sort(A, p, q)
    inversions += merge_sort(A, q + 1, r)
    inversions += merge(A, p, q, r)
    return inversions
 else
    return 0

MERGE(A, p, q, r)
 n1 = q - p + 1
 n2 = r - q
 let L[1..n1] and R[1..n2] be new arrays
 for i = 1 to n1
    L[i] = A[p + i - 1]
 for j = 1 to n2
    R[j] = A[q + j]
 i = 1
 j = 1
 for k = p to r
```

```
    if i > n_l
        A[k] = R[j]
        j = j + 1
    else if j > n_2
        A[k] = L[i]
        i = i + 1
    else if L[i] ⩽ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
        inversions += n_l - i
return inversions
```