Name: Shuo WU Student ID: 45266 Wustl key: wushuo

CSE 541T HW1

Question 1,

a, linear search:

```
1,      for i =0 to A.length-1
2,              now = A[i]
3,              if v = now
4,                      return i
5,              end if
6,      end for
7,      return NIL
```

Loop invariants:

Initialization:

We start by showing that the loop invariant holds before the first loop iteration, when i=0. The now =A[0] which is the first item. If the *v* = A[0], the algorithm will return 0.
Else, if the array length = 1, it will return NIL, or it will go to the first iteration of the loop. So that shows the loop invariant holds prior to the first iteration of the loop.

Maintenance:

The body of the for loop will go through A[1], A[2]…A[A.length-3], A[A.length-2], A[A.length-1]. If A[x] != *v*, then the algorithm will check if A[x+1] =*v* or not. At any point, if A[x]= *v*(x< A.length-1), the algorithm will return the value of x as index, or it will go to the next iteration of loop. Therefore, the loop variant holds until the algorithm goes to the last position of the array which is A[A.length-1].

Termination:

Finally, if the algorithm finds any x (x means the index, and only the first x) that makes A[x]= *v*, it will terminate and return the value of x. Or after comparing A[A.length-1] with v, if the algorithm go through all the position of the array and doesn't find any x that holds A[x] = *v*, it will return NIL as line 7, we can say there isn't any elements that equals to v in the array A. Hence, the algorithm is correct.

b. selection sort

```
1,              for i =0 to n-1 do :
2,              minIndex = i
3,                      for j =i+1 to n do:
4,                              if A[j] < A[minIndex]:
5,                                      minIndex = j
6,                              end if
7,                      end for
8,                      if list[minIndex] < list[i]:
9,                              exchange A[minIndex] with A[i]
10,             end for
```

Loop invariants:

Initialization:

We start by showing that the loop invariant holds before the first loop iteration, when i=0, we assume the index of minimum element is 0, then we go into the second for loop starts at line 4,

in this for loop, we iterate all the element from index 1(i+1) to the last one to check if the smallest element in the subarray A[1….last one], if the smallest element of subarray smaller than list [i], then exchange the value of these two, that move the smallest element of the subarray to the current index.

Maintain:

After finding the smallest element of the subarray and exchanging the smallest element and current element, the algorithm increase i which makes i= i+1, and then find the smallest element of the subarray from list[i+2] to list[list.length-1], then it compares the smallest element of the subarray with the current element which is A[i+1] to make sure it put the smallest element from subarray into A[i+1], so the algorithm remains true for the next iterations.

Termination:

When the loop terminates, the element in the last position (at the end of the array) is the biggest element of the whole array- list. Which i = list.length-1. Now we have the subarray list[0..list.length-2] sorted and list[list.length-1] which is the biggest element of the array list. Since it doesn't been selected from the previous iterations. Therefore, we conclude that the entire array is sorted. Hence the algorithm is correct

c. bubble-sort

```
1,      for i = 0 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]
```

Loop invariants:

Initialization:

For the first iteration, i=0, then we go into the second loop in lines through 4 find the smallest element of the whole array A, it kinds of like make the smaller of two elements stand out, and then move the smaller element to first position and the bigger one to the later position, after first iteration, we get the smallest element of the whole array.

Maintain:

The for loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of A[i..n] is at most j. This is clearly true prior to the first iteration because the position of any element is at most A.length. To see that each iteration maintains the loop invariant, suppose that j = k and the position of the smallest element of A[i..n] is at most k. Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can't be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.

Termination:

The for loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Prior to the first iteration i = 1, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fix i and suppose that A[1..i − 1] contains the i − 1 smallest

elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of A[i..n] is in position i. Since the i − 1 smallest elements of A are already in A[1..i − 1], A[i] must be the i th smallest element of A. Therefore A[1..i] contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, A[1..n] contains the n elements of A in sorted order as desired.

d. Horner's rule

1, y=0

2, for i= n downto 0

3 $y=a_i + x*y$

Initialization:

When i=n, (the first loop iteration). The value of y, therefore, equals to a_n, which is in fact the last element of polynomial.

For example if n=0, from the algorithm, we will get that $y = a_0$

And the polynomial gives us that $y = a_0$, therefore, they have the same result.

Showing that the loop invariant holds prior to the first iteration of the loop.

Maintain:

The for loop then makes y equals $y = a_i + x*(a_{i+1}+x*y') =$

$$= a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = a_i + x\sum_{k=1}^{n-i} a_{k+i}x^{k-1} = \sum_{k=0}^{n-i} a_{k+i}x^k$$

which shows that the loop invariant holds prior to terminate.

Termination:

At termination, i=0, so is summing up to n-1, therefore, the last iteration give us the correct final output as shown before. We can see that the algorithm evaluated to the function. This is the value of the polynomial evaluated at x.

Question 2,

a.

Pseudo-code:

```
def selection_sort(A, index):
    if index >= len(A):
        print(A)
        return
    minIndex = index
    for i in range (index+1,len(A)):
        if A[minIndex] > A[i]:
            minIndex = i
    if minIndex !=index:
        exchange A[index] with A[minIndex]
    return selection_sort(A, index+1)
```

T(n) = T(n-1)+ O(n)

The running time for worst-case = $\Theta(n^2)$

b.

Pseudo-code:

```
def binary_search(A, target, start_index, end_index):
    if end_index < start_index:
        return -1
    else :
        mid = (end_index+start_index)/2
        if A[mid] == target:
            return mid
        elif A[mid] > target:
            return binary_search(A, target,start_index, mid-1)
        else:
            return binary_search(A ,target, mid+1, end_index)
```

T(n) = T(n/2)+ O(1)

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

Question 3,

Select any two elements from set A[], to make x1+x2 = target,
If exist, return true,
Else, return false.
Pseudo, we need to sort the set A first,
We use merge sort to sort the elements in set A
Using the pseudo code in textbook
Merge–sort (A,p,r)
If p<r
q = [(p+r)/2]
merge-sort (A,p,q)
merge-sort (A,q+1,r)
merge(A,p,q,r)
the running time for merge sort is $\Theta(nlg(n))$
after sorting the set A, we use the following algorithm to tell whether or not there exist two elements in A whose sum is X

```
int i = 0
int j = n
while i<j do:
        if A[i]+A[j] == X:
                return true
        if A[i]+A[j] <x:
                i++;
                continue
        if A[i]+A[j]>x:
                j--;
                continue
end while
return false
```

the time complexity for the above code is $\Theta(n)$, cause in worst case, we only need to iterate all the element in array A, and spend constant time on each element to tell whether or not the sum of them is equals to the target value. So the total running time for these two part (first part is merge sort, the second part is above pseudo code) is dominated by the sort part which is $\Theta(nlg(n))$.
Generally, the first step is merge sort time is $t1 = \Theta(nlg(n))$
The second step is find the sum, using two "pointer", the first one is in the start of the array, the second one is in the end of the array, running time is $t2 = \Theta(n)$
So $T(n)= t1 + t2= \Theta(nlg(n))$

Question 4,

We know that, cut a list with n elements into n/k sub-lists of length k
a, the insertion sort can sort a list of length k in $\Theta(k^2)$, so for n/k sub-lists, the total time will be $\Theta(k^2*(n/k))= \Theta(nk)$

b, since the depth of recursion tree for normal merge sort is Log(n),
so the depth of recursion tree for this problem is Log(n)- Log(k), because the recursion tree stops at the sub-list at size k. therefore, the depth of recursion tree is log(n/k)
and each level time consuming is cn, so the total time is log(n/k)*cn which is equals to $\Theta(n\log(n/k))$.

c, if nlog(n/k)+nk = n log(n) in terms of Θ notation
then we need to make sure
nlog(n/k) = Θ(nlogn) (if nlog(n/k) >nk) ~1
or
nk= Θ(nlogn) if nlog(n/k) <nk ~2
so for 1,
k is a constant, not a function of n
for 2,
k =Θ(logn), they have the same asymptotic.
So k =Θ(logn), or we can say k $\in O$ (logn)

d, in practice, we can choose k depends on value c1 and c2 which are the coefficients of nk and nlog(n/k) hidden by the asymptotic notation. We can build a mathematic model to analysis it by doing calculus 1 of k, and find the peak value which makes (nk+ nlog(n/k))' = 0, then we can choose the k value, again the values of coefficients are important for choosing k value.

Name: Shuo WU Student ID: 45266 Wustl key: wushuo

Question 5,

I use divide and conquer algorithm to solve this problem.

For integer A with n digit, we divide it into two integers: a_L (A-left) and a_R (A-right) each with n/2 digits

Same as integer B, b_L and b_R

$$\begin{array}{rr} & a_L \quad a_R \\ X & b_L \quad b_R \\ \hline & a_Lb_R \quad a_Rb_R \\ + a_Lb_L & a_Rb_L \\ \hline a_Lb_L & (a_Lb_R + a_Rb_L) \quad a_Rb_R \end{array}$$

so more formally, we can get a*b by doing this:

$ab = (a_L\ 10^{n/2} + a_R)\ (b_L\ 10^{n/2} + b_R)$

$= a_L\ b_L\ 10^n + a_L\ b_R\ 10^{n/2} + a_R\ b_L\ 10^{n/2} + a_R\ b_R$

$= a_L\ b_L\ 10^n + (a_L\ b_R + a_R\ b_L)\ 10^{n/2} + a_R\ b_R$

$x_1 = a_L\ b_L$

$x_2 = a_R\ b_R$

$x_3 = a_L\ b_R$

$x_4 = a_R\ b_L$

Pseudo-code:

```
multiply(A, B)
{
int n = A.length();
if(n==1)  {
          int C =A.value() * B.value();
          return  C;
          }
else
          {
          aR = last n/2 digits of A; //right n/2 digits
          aL = first remaining (n/2 for even digit, n/2+1 for odd digits) digits of A;
          bR = last n/2 digits of B;
          bL = first remaining (n/2 for even digit, n/2+1 for odd digits) digits of B;
          X1 = multiply(aL, bL);
          X2 = multiply(aR,bR);
          X3 = multiply(aL, bR,);
          X4 = multiply(aR, bL);

          return   x1*pow(10,n)+(x3+x4)*pow(10,n/2)+x2;
          }
```

```
}

new_array (A,B,n){
        int answer= multiply(A,B);
        array C[] =new array [];
        for (int i=0; i<2n;i++){
        C[i] = answer.GetFirstDigit();
        answer = (int)(answer/10);
        }
        return C;
}
```

For multiplying two n-digit integers, we can recursively multiply four pair of n/2-digit integers.
The total time is T(n) = T1+T2
T1 is 4T(n/2)
T2 is $\Theta(n)$
So $T(n) = \Theta(n^2+n)= \Theta(n^2)$
So for big-O notation, the running time is $T(n)= O(n^2)$