COMP9024 20T0 Data Structures and Algorithms

Week 01b Problem Set
Analysis of Algorithms

1. (Counting primitive operations)

   The following algorithm

   - takes a sorted array A[1..*n*] of characters
   - and outputs, in reverse order, all 2-letter words vω such that v≤ω.

```
for all i=n down to 1 do
   for all j=n down to i do
      print "A[i]A[j]"
   end for
end for
```

   Count the number of primitive operations (evaluating an expression, indexing into an array). What is the time complexity of this algorithm in big-Oh notation?

   Answer:

Statement	# primitive operations
for all i=*n* down to 1 **do**	n+(n+1)
for all j=*n* down to *i* **do**	3+5+…+(2n+1) = n(n+2)
print "A[*i*]A[*j*]"	(1+2+…+n)·2 = n(n+1)
end for	
end for	

   Total: $2n^2+5n+1$, which is $O(n^2)$

2. (Big-Oh Notation)

   a. Show that $\sum_{i=1}^{n} i^2 \in O(n^3)$

   b. Show that $\sum_{i=1}^{n} \log i \in O(n \log n)$

   c. Show that $\sum_{i=1}^{n} \frac{i}{2^i} \in O(1)$

   Answer:

   a. $1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$, which is in $O(n^3)$.

   b. $\sum_{i=1}^{n} \log i \le \sum_{i=1}^{n} \log n = n \cdot \log n$, which is in $O(n \log n)$

   c. Let $S = \sum_{i=1}^{n} \frac{i}{2^i}$. Then $S = \sum_{i=1}^{n} \frac{1}{2^i} + \sum_{i=1}^{n} \frac{i-1}{2^i} = \sum_{i=1}^{n} \frac{1}{2^i} + \sum_{i=1}^{n-1} \frac{i}{2^{i+1}} < 1 + \frac{1}{2}S$. Therefore, $S < 2$. Consequently, $\sum_{i=1}^{n} \frac{i}{2^i}$ is in $O(1)$.

   Note: it is easy to see that $\sum_{i=1}^{n} \frac{1}{2^i} < 1$ from the fact that $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$.

3. (Algorithms and complexity)

   Develop an algorithm to determine if a character array of length *n* encodes a *palindrome*, that is, which reads the same forward and backward. For example, "racecar" is a palindrome.

   a. Write the algorithm in pseudocode.

   b. Analyse the time complexity of your algorithm.

   c. Implement your algorithm in C. Your program should prompt the user to input a string and check whether it is a palindrome. Examples of the program executing are

```
prompt$ ./palindrome
Enter a word: racecar
yes
prompt$ ./palindrome
Enter a word: reviewer
no
```

   Hint: You may use the standard library function `strlen(char[])`, defined in `<string.h>`, which computes the length of a string (without counting its terminating `'\0'`-character).

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `palindrome.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ 9024 dryrun palindrome
```

Answer:

```
isPalindrome(A):
   Input  array A[0..n-1] of chars
   Output true if A palindrome, false otherwise

   i=0, j=n-1
   while i<j do
      if A[i]≠A[j] then
         return false
      end if
      i=i+1, j=j-1
   end while
   return true
```

Time complexity analysis: There are at most $\left\lfloor \frac{n}{2} \right\rfloor$ iterations of the loop, and the operations inside the loop are O(1). Hence, this algorithm takes O(*n*) time.

```
#include <stdio.h>
#include <string.h>

int isPalindrome(char A[], int len) {
   int i = 0;
   int j = len-1;

   while (i < j) {
      if (A[i] != A[j])
         return 0;
      i++;
      j--;
   }
   return 1;
}

int main(void) {
   char str[BUFSIZ];

   printf("Enter a word: ");
   scanf("%s", str);
   if (isPalindrome(str, strlen(str))) {
      printf("yes\n");
   } else {
      printf("no\n");
   }
   return 0;
}
```

4. (Algorithms and complexity)

   Let $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ be a polynomial of degree n. Design an O(*n*)-time algorithm for computing the function $p(x)$.

   Hint: Assume that the coefficients a_i are stored in an array A[0..n].

   Answer:

   Rewriting $p(x)$ as $(\ldots((a_n x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \ldots + a_1) \cdot x + a_0$ leads to the following algorithm:

```
p(A,x):
   Input  coefficients A[0..n], value x
   Output A[n]·x^n+A[n-1]·x^(n-1)+...+A[1]·x+A[0]

   p=A[n]
   for all i=n-1 down to 0 do
      p=p·x+A[i]
   end for
```

   This is obviously O(n).

5. (Algorithms and complexity)

   A vector V is called *sparse* if most of its elements are 0. In order to store sparse vectors efficiently, we can use an array L to store only its non-zero elements. Specifically, for each non-zero element *V[i]*, we store an index-value pair *(i,V[i])* in L.

   For example, the 8-dimensional vector *V*=(2.3,0,0,0,-5.61,0,0,1.8) can be stored in an array L of size 3, namely L[0]=(0,2.3), L[1]=(4,-5.61) and L[2]=(7,1.8). We call L the *compact form* of V.

   Describe an efficient algorithm for adding two sparse vectors V_1 and V_2 of equal dimension but given in compact form. The result should be in compact form too, of course. What is the time complexity of your algorithm depending on the sizes *m* and *n* of the compact forms of V_1 and V_2, respectively?

   Hint: The sum of two vectors V_1 and V_2 is defined as usual, e.g. (2.3,-0.1,0,0,1.7,0,0,0) + (0,3.14,0,0,-1.7,0,0,-1.8) = (2.3,3.04,0,0,0,0,0,-1.8).

   Answer:

In the algorithm below, `L[i].x` denotes the first component (the index) of a pair L[i], and `L[i].v` denotes its second component (the value).

```
VectorSum(L1,L2):
   Input  compact forms L1 of length m and L2 of length n
   Output compact form of L1+L2

   i=0, j=0, k=0
   while i≤m-1 and j≤n-1 do
      if L1[i].x=L2[j].x then       // found index with entries in both vectors
         if L1[i].v≠-L2[j].v then   // only add if the values don't add up to 0
            L3[k]=(L1[i].x, L1[i].v+L2[j].v)
            i=i+1, j=j+1, k=k+1
         end if
      else if L1[i].x<L2[j].x then  // copy an entry from L1
         L3[k]=(L1[i].x,L1[i].v)
         i=i+1, k=k+1
      else
         L3[k]=(L2[j].x,L2[j].v)    // copy an entry from L2
         j=j+1, k=k+1
      end if
   end while
   while i<m-1 do                  // copy the remaining pairs of L1, if any
      L3[k]=(L1[i].x,L1[i].v)
      i=i+1, k=k+1
   end while
   while j<n-1 do                  // copy the remaining pairs of L2, if any
      L3[k]=(L2[j].x,L2[j].v)
      j=j+1, k=k+1
   end while
```

Time complexity analysis: Adding a pair to L_3 takes O(1) time. At most $m+n$ pairs from L_1 and L_2 are added. Therefore, the time complexity is $O(m+n)$.

6. (Ordered linked lists)

A particularly useful kind of linked list is one that is sorted. Give an algorithm for inserting an element at the right place into a linked list whose elements are sorted in ascending order.

Example: Given the linked list

```
L = 17 26 54 77 93
```

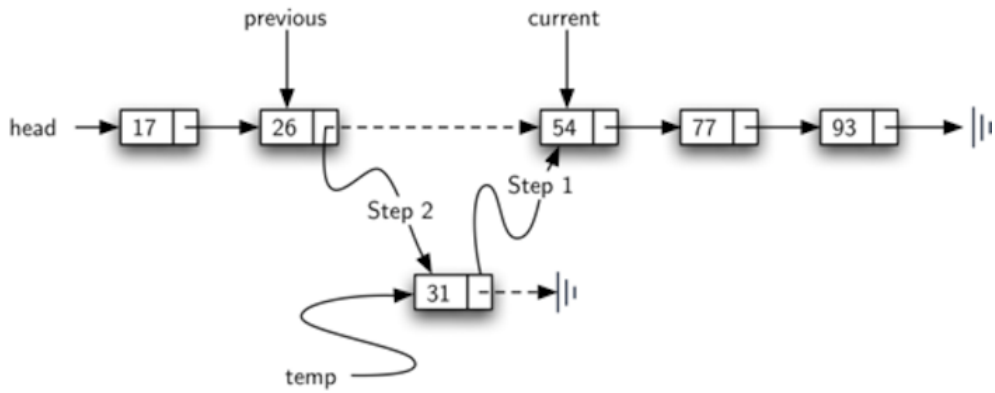
the function `insertOrderedLL(L,31)` should return the list

```
L = 17 26 31 54 77 93
```

Hint: Develop the algorithm with the help of a diagram that illustrates how to use pointers in order to
- find the right place for the new element and
- link the new element to its predecessor and its successor.

Answer:

The following diagram illustrates the use of pointers to the previous, the current and the new list element, and how to link them:




```
insertOrderedLL(L,d):
   Input  ordered linked list L, value d
   Output L with d added in the right place

   current=L, previous=NULL
   while current≠NULL and current.value<d do
      previous=current
      current=current.next
   end while
   temp=makeNode(d)
   if previous≠NULL then
      temp.next=current      // Step 1
      previous.next=temp     // Step 2
   else
```

```
      temp.next=L            // add new element at the beginning
      L=temp
   end if
   return L
```

7. (Advanced linked list processing)

   Describe an algorithm to split a linked list in two halves and output the result. If the list has an odd number of elements, then the first list should contain one more element than the second.

   Note that:
   - your algorithm should be 'in-place' (so you are not permitted to create a second linked list or use some other data structure such as an array);
   - you should not traverse the list more than once (e.g. to count the number of elements and then restart from the beginning).

   An example of the result of the algorihtm could be

```
Linked list: 17 26 31 54 77 93 98
First half:  17 26 31 54
Second half: 77 93 98
```

   Answer:

   The following solution uses a "slow" and a "fast" pointer to traverse the list. The fast pointer always jumps 2 elements ahead. At any time, if `slow` points to the i^{th} element, then `fast` points to the $2{\cdot}i^{th}$ element. Hence, when the fast pointer reaches the end of the list, the slow pointer points to the last element of the first half.

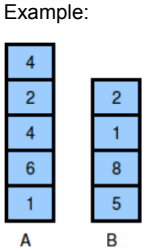
```
SplitList(L):
   Input linked list L

   slow=L, fast=L.next
   while fast≠NULL and fast.next≠NULL do
      slow=slow.next
      fast=fast.next.next
   end while
   List2=slow.next           // this becomes head of second half
   slow.next=NULL            // cut off at end of first half
   print "First half:", showLL(L)
   print "Second half:", showLL(List2)
```

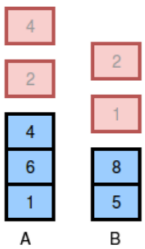
8. **Challenge Exercise**

   Suppose that you are given two stacks of non-negeative integers A and B and a target threshold $k \geq 0$. Your task is to determine the maximum number of elements that you can pop from A and B so that the sum of these elements does not exceed k.

   Example:



   Maximum number of elements that can be popped without exceeding k = 10 is 4:



   If k = 7, then the answer would be 3 (the top element of A and the top two elements of B).

   a. Write an algorithm (in pseudocode) to determine this maximum for any given stacks A and B and threshold k. As usual, the only operations you can perform on the stacks are pop() and push(). You *are* permitted to use a third "helper" stack but no other aggregate data structure.

   b. Determine the time complexity of your algorithm depending on the sizes m and n of input stacks A and B.

   Hints:
   - A so-called greedy algorithm would simply take the smaller of the two elements currently on top of the stacks and continue to do so as long as you haven't exceeded the threshold. This won't work in general for this problem.
   - Your algorithm only needs to determine the number of elements that can maximally be popped without exceeding the given k. You do not have to return the numbers themselves nor their sum. Also you do not need to restore the contents of the two stacks; they can be left in any state you wish.

   Answer:

```
MaxElementsToPop(A,B,k):
   Input  stacks A and B, target threshold k≥0
   Output maximum number of elements that can be popped from A and B
          so that their sum does not exceed k

   sum=0, count=0, create empty stack C
   while sum≤k and stack A not empty do   // Phase 1: Determine how many elements can be popped just from A
      v=pop(A), push(v,C)                 //          and push those onto the helper stack C
      sum=sum+v, count=count+1
   end while
   if sum>k then                          // exceeded k?
      sum=sum-pop(C), count=count-1       //   then subtract last element that's been popped off A
   end if
   best=count                             // best you can do with elements from A only
```

```
while stack B not empty do
   sum=sum+pop(B), count=count+1       // Phase 2: add one element from B at a time
   while sum>k and stack C not empty do   // and whenever threshold is exceeded:
      sum=sum-pop(C), count=count-1       //   subtract more elements originally from A (now in C)
   end while                              //   to get back below threshold
   if sum≤k and count>best then
      best=count                       // update each time you got a better score
   end if
end while
return best
```

Time complexity analysis: In phase 1, the worst case is when all *m* elements in stack A need to be visited, for a maximum of *m*+1 pop and *m* push operations. In phase 2, the worst case is when all elements have to be taken from both B and C, for a maximum of *m*+*n* pop operations. Assuming that push() and pop() take constant time, the overall complexity is O(m+n). This makes it a linear-time solution.