COMP9024 24T1 Data Structures and Algorithms

Week 3 Tutorial
Analysis of Algorithms

[Show with no answers] [Show with all answers]

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?

   [hide answer]

Statement	# primitive operations
for all i=*n* down to 1 **do**	(n+1)+n = 2n+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 $O(n^2)$

   To help understand the number of primitive operations without relying on maths beyond the scope of this course:

   - The outer loop is executed *n* times. This means we'll have:
     - *n*+1 primitive operations to evaluate the loop condition before each iteration. These are the *n* times the condition is true and the body of the loop is executed, plus the 1 time it is false and the loop is terminated.
     - *n* primitive operations to decrement the loop counter at the end of each iteration.
     - Thus, the outer loop accounts for a total of (n+1)+n = 2n+1 primitive operations.
   - As for the inner loop:
     - The first time the outer loop is entered, the inner loop is executed once.
     - The second time the outer loop is entered, the inner loop is executed twice.
     - And so on until the n^{th} time the outer loop is entered, and the inner loop is executed *n* times.
     - This forms a very common arithmetic series, sometimes called the Gauss Summation, for the number of times the inner loop is executed: 1+2+...+n = $\frac{n(n+1)}{2}$, meaning in total the inner loop is executed $\frac{n(n+1)}{2}$ times.
     - For now let's just focus on the $\frac{n(n+1)}{2}$ times the inner loop condition is true and its body is executed.
       - Each of those times we need 2 primitive operations, 1 for the loop condition and 1 for the loop counter.
       - So this totals $\frac{2n(n+1)}{2} = n(n+1)$ primitive operations.
     - But we also need to account for the times when the loop condition fails and the inner loop terminates.
       - This will happen once each time the outer loop is entered, meaning it will happen *n* times.
     - So in total the inner loop will require n(n+1)+n = n(n+2) primitive operations.
   - Finally, we're in the body of the inner loop, where we have a single print statement.

- This will be called each time the inner loop is entered, i.e. $\frac{n(n+1)}{2}$ times.
- Each time we need to index into the array twice.
- This gives us a total of $\frac{2n(n+1)}{2}$ = n(n+1) primitive operations.

- To summarise:
  - The outer loop requires 2n+1 primitive operations.
  - The inner loop requires n(n+2) primitive operations.
  - The print statement requires n(n+1) primitive operations.
  - For a total of $2n^2+5n+1$ primitive operations.

2. (Algorithms and complexity)

Design an algorithm to determine if an integer array, A, contains two elements that sum to a given value, v.

a. Write the algorithm in pseudocode.
b. Determine the worst-case time complexity of your algorithm.
c. Translate your algorithm into a C function that accepts the array, its length, and the sum value, and returns true or false.

[hide answer]

a.

```
hasTwoSum(A, v):
|  Input:  array A[1..n] of integers
|          integer v
|  Output: true if A contains two elements that sum to v, false otherwise
|
|  for all i=1..n-1 do
|  |  for all j=i+1..n do
|  |  |  if A[i] + A[j] = v then
|  |  |     return true
|  |  |  end if
|  |  end for
|  end for
|
|  return false
```

b. The worst case occurs when there are no two elements that sum to the given value. In this case, there are n-1 iterations of the outer loop and (n-1) + (n-2) + ... + 1 = $\frac{n(n-1)}{2}$ iterations of the inner loop. Removing constant factors and lower-order terms gives a worst case time complexity of $O(n^2)$.

c.

```
bool hasTwoSum(int a[], int n, int v) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (a[i] + a[j] == v) {
            return true;
         }
      }
   }
   return false;
}
```

3. (Asymptotic growth)

Arrange the following functions in increasing order according to asymptotic growth:

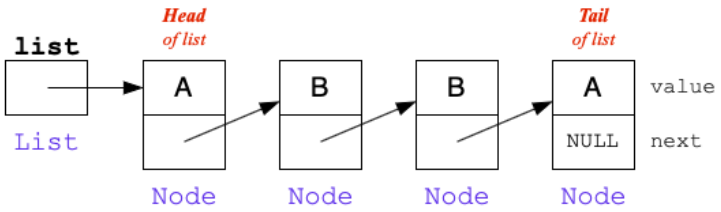
3^n, $\sqrt{4^n}$, $\log^2 n$, $\sqrt{n}$, n^2, $\log n$, $20n$

[hide answer]

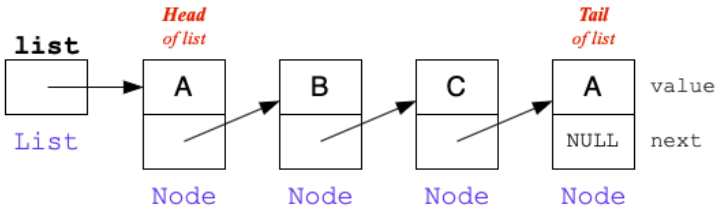
- First the logs: $\log n$, $\log^2 n$
- Second the polynomials: $\sqrt{n}$, $20n$, n^2
- Third the exponentials: $\sqrt{4^n} = 2^n$, 3^n

4. (Linked lists)

A word is a palindrome if its reverse is equal to itself. For example, ABBA is a palindrome, whereas ABCA is not. One way of representing a word in a computer is to have a singly-linked list, where we have a list element for each letter. For example, ABBA is represented by the singly-linked list



and ABCA is represented by the singly-linked list



Design an algorithm that, given a pointer to the head of a singly-linked list which represents a word, outputs true if the word is a palindrome, and false otherwise. Your algorithm should run in linear time and should not use any data structures other than singly-linked lists, i.e. no arrays, stacks, queues, etc.

[hide answer]

Let L be the linked list representing our word; the idea of the algorithm is to create a new linked list R, which contains the word of L reversed, and then compare the two words. We create the linked list R by traversing the list L and for each element in L, inserting this element at the head of R. At the end of this procedure, the list R will be equal to the reverse of L.

```
Palindrome(L)
|  Input  linked list L representing a word
|  Output true if L is a palindrome, and false otherwise
|
|  p=L
|  R=NULL
|  while p≠NULL do
|     R=insertLL(R,p.value)
|     p=p.next
|  end while
|  p=L
|  q=R
|  while p≠NULL do
|     if p.value≠q.value then
|        return false
|     end if
|     p=p.next
|     q=q.next
|  end while
|  return true
```

In the worst case, both `while` loops will be executed n times, where n is the length of the word. Since each loop consists of a constant number of operations, each of constant time, the worst case time complexity is O(n).

5. (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
```

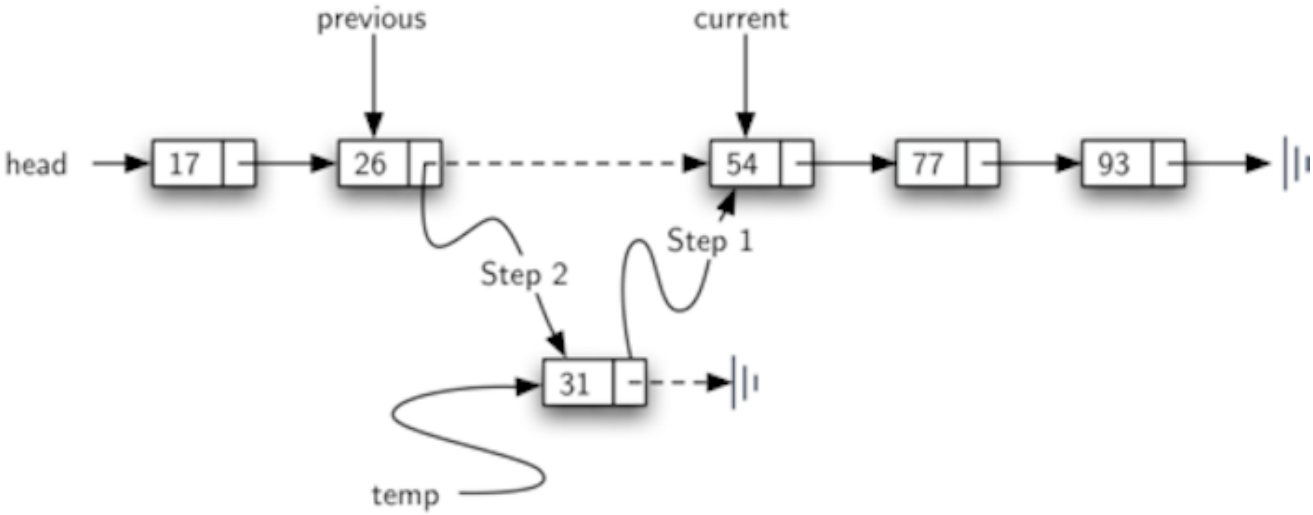
the function `insertOrderedLL(L,31)` should return the list

```
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.

[hide 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
```

6. (Doubly-linked lists)

In the lecture we have considered **singly-linked** lists, where each element contains a pointer `p.next` to the next element.

In a **doubly-linked list**, every element p contains two pointers:

- a pointer `p.next` to the next element and
- a pointer `p.prev` to the previous element.

For the first element, the value of `p.prev` is NULL. Newly created elements are always assumed to have been initialised with `new.next=new.prev=NULL` .

To maintain a doubly-linked list itself, we need to store two pointers:

- a pointer `head` to the first element and

- a pointer `tail` to the last element.

This is similar to the front and rear pointers used when implementing a queue by a singly-linked list.

a. Let `elem` be a pointer to an element in the list. Describe an algorithm in pseudocode to delete the element at address `elem`. How many pointers in total need to be redirected (i.e. their values changed)?

b. Let `elem` be a pointer to an existing element in the list, and let `new` be a pointer to a newly created element. Describe an algorithm in pseudocode to insert `new` directly after `elem`. How many pointers in total need to be redirected?

[hide answer]

a. Two pointers need to be redirected when deleting an element:

```
Delete(head, tail, elem):
|  Input linked list with head, tail
|        list element elem
|
|  if elem=head then
|     head = elem.next
|  else
|     elem.prev.next = elem.next
|  end if
|  if elem=tail then
|     tail = elem.prev
|  else
|     elem.next.prev = elem.prev
|  end if
```

b. A maximum of four pointers need to be redirected when adding an element:

```
Insert(head, tail, elem, new):
|  Input linked list with head, tail
|        existing list element elem
|        new list element
|
|  new.prev = elem
|  if elem=tail then
|     tail = new
|  else
|     new.next = elem.next
|     elem.next.prev = new
|  end if
|  elem.next = new
```

Challenge Exercises

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

[hide answer]

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

b. $\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \cdot \log n$, 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 $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$.

8. (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].

[hide 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).

9. (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
```

[hide 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)
```