

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 vw such that $v \leq w$.

```
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	$3+5+\dots+(2n+1) = n(n+2)$
print "A[i]A[j]"	$(1+2+\dots+n) \cdot 2 = n(n+1)$
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. $1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$, 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 $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$.

3. (Algorithms and complexity)

Let $p(x) = a_nx^n+a_{n-1}x^{n-1}+\dots+a_1x+a_0$ 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 $((\dots((a_nx+a_{n-1}) \cdot x+a_{n-2}) \cdot x+\dots+a_1) \cdot x+a_0)$ 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 $O(n)$.

4. (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

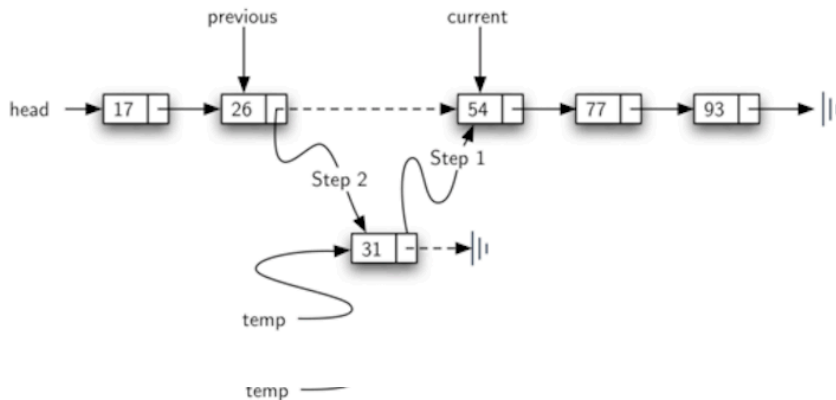
`L = 17 26 31 54 77 93`

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
```

5. (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 algorithm 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 $2i^{\text{th}}$ element. Hence, when the fast pointer reaches the end of the list, the slow pointer points to the last element of the first half.

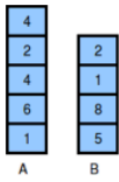
```
SplitList(L):
  Input linked list L

  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)
```

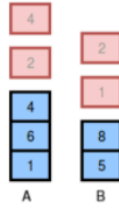
6. Challenge Exercise

Suppose that you are given two stacks of non-negative 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).

- 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.
- 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.

