(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 $O(n^2)$

(Big-Oh Notation)

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

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

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

Answer:

a. $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$, 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 $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$.

(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?

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
```