

## 1. (Big-Oh Notation)

- Show that  $\sum_{i=1}^n i^2$  is  $O(n^3)$
- Show that if  $p(n)$  is any polynomial in  $n$ , then  $\log p(n)$  is  $O(\log n)$
- Show that  $\sum_{i=1}^n \log i$  is  $O(n \log n)$
- Show that  $\sum_{i=1}^n \frac{i}{2^i}$  is  $O(1)$

**Answer:**

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

b. Let  $p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$ . Then  $p(n) < \max \{|a_k|, |a_{k-1}|, \dots, |a_0|\} \cdot (k+1) \cdot n^k$ . Hence,  $\log p(n) < k \cdot \log \max \{|a_k|, |a_{k-1}|, \dots, |a_0|\} \cdot (k+1) \cdot n$ .

Since  $k$  and  $a_k, \dots, a_0$  are constants, it follows that  $\log p(n)$  is  $O(\log n)$ .

c.  $\sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \cdot \log n$ , which is  $O(n \log n)$

d. 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  $O(1)$ .

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

## 2. (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
<b>for all</b> i=n down to 1 <b>do</b>	$n+(n+1)$
<b>for all</b> j=n down to i <b>do</b>	$3+5+\dots+(2n+1) = n(n+2)$
print "A[i]A[j]"	$(1+2+\dots+n) \cdot 2 = n(n+1)$
<b>end for</b>	
<b>end for</b>	

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

## 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.

- Write the algorithm in pseudocode.
- Analyse the time complexity of your algorithm.
- Implement your algorithm in C. Your program should accept a single command line argument and check whether it is a palindrome. Examples of the program executing are

```
prompt$ ./palindrome racecar
yes
prompt$ ./palindrome 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 the problem set and week. It expects to find a program named `palindrome.c` in the current directory. You can use `dryrun` as follows:

```
prompt$ ~cs9024/bin/dryrun prob05
```

**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(int argc, char *argv[]) {
  if (argc == 2) {
    if (isPalindrome(argv[1], strlen(argv[1])))
      printf("yes\n");
    else
      printf("no\n");
  }
  return 0;
}
```

#### 4. (Algorithms and complexity)

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

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

**Answer:**

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

```
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 a list  $L$  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 a list  $L$  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.  $(2.3, -0.1, 0.0, 1.7, 0.0, 0) + (0.3, 1.4, 0.0, -1.7, 0.0, -1.8) = (2.3, 3.04, 0.0, 0.0, 0.0, -1.8)$ .

**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( $L_1, L_2$ ):
  Input compact forms  $L_1$  of length  $m$  and  $L_2$  of length  $n$ 
  Output compact form of  $L_1 + L_2$ 

   $i=0, j=0, k=0$ 
  while  $i \leq m-1$  and  $j \leq n-1$  do
    if  $L_1[i].x = L_2[j].x$  then           // found index with entries in both vectors
      if  $L_1[i].v \neq -L_2[j].v$  then       // only add if the values don't add up to 0
         $L_3[k] = (L_1[i].x, L_1[i].v + L_2[j].v)$ 
         $i=i+1, j=j+1, k=k+1$ 
      end if
    else if  $L_1[i].x < L_2[j].x$  then       // copy an entry from  $L_1$ 
       $L_3[k] = (L_1[i].x, L_1[i].v)$ 
       $i=i+1, k=k+1$ 
    else
       $L_3[k] = (L_2[j].x, L_2[j].v)$        // copy an entry from  $L_2$ 
       $j=j+1, k=k+1$ 
    end if
  end while
  while  $i < m-1$  do                       // copy the remaining pairs of  $L_1$ , if any
     $L_3[k] = (L_1[i].x, L_1[i].v)$ 
     $i=i+1, k=k+1$ 
  end while
  while  $j < n-1$  do                       // copy the remaining pairs of  $L_2$ , if any
     $L_3[k] = (L_2[j].x, L_2[j].v)$ 
     $j=j+1, k=k+1$ 
  end while

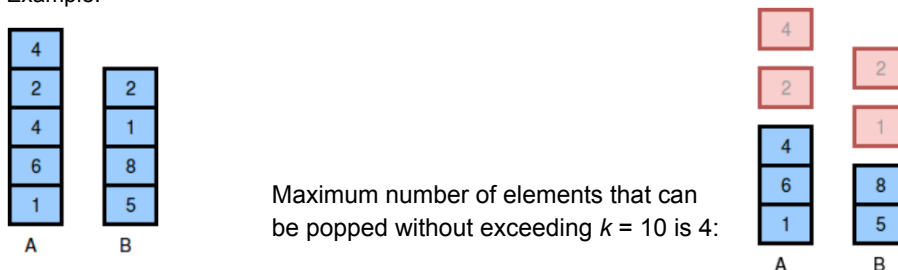
```

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. 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:



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  $k \geq 0$ 
  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  $sum \leq k$  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.