COMP9024 18s2

Week 05 Problem Set
Analysis of Algorithms

Data Structures and Algorithms

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

1. (Big-Oh Notation)

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

   b. Show that if *p(n)* is any polynomial in *n*, then *log p(n)* is O(*log n*)

   c. Show that $\sum_{i=1}^{n} \log i$ is O(*n log n*)

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

   [show answer]

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 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?

   [show answer]

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 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
```

[show answer]

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 $p(x)$.

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

   [show answer]

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?

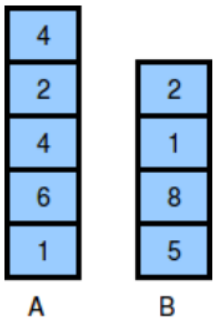
   Hint: The sum of two vectors V_1 and V_2 is defined as usual, e.g. (2.3,-0.1,0,0,1.7,0,0,0) + (0,3.14,0,0,-1.7,0,0,-1.8) = (2.3,3.04,0,0,0,0,0,-1.8).

   [show answer]

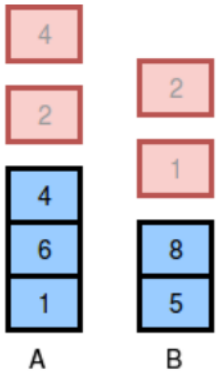
6. **Challenge Exercise**

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.

[show answer]