Data Structures and Algorithms

Degree in Computer Science, Group I

Final Exam, June 2015

1. [1 pt] For a given array of n elements ($n \geq 0$), we need to design an algorithm that returns the sum of all the products of the pairs of values situated in different positions, with a resulting complexity in $\mathcal{O}(n)$. The specification of the algorithm is as follows:

$\{0 \leq n < \mathit{length}(v)\}$
fun *sumProducts* (**int** $v[\,]$, **int** n) **return int** p
$\{p = \sum i, j : 0 \leq i < j < n : v[i] * v[j]\}$

We have coded the following two algorithms to solve the problem:

(a)
```
k = n; p = 0; s = 0;
while (k>0)
 { p = p + v[k-1] * s;
   s = s + v[k-1];
   k = k - 1;
 };
return p;
```

(b)
```
k = 0; p = 0; s = 0;
while (k<n)
 { p = p + v[k] * s;
   s = s + v[k];
   k = k + 1;
 };
return p;
```

Write the **invariant** of the while loop for both algorithms.

2. [3 pt] We have a set of n activities for which we know their starting time c_i and their finalization time f_i, i.e., the interval in which the activity i ($0 \leq i < n$) runs is $[c_i, f_i]$. Design a backtracking algorithm that prints all the subsets of activities containing r or more activities that do not overlap among them (i.e. their intersection is void).

You can assume that the intervals for each activity are represented in an array of a pairs of integers (the starting and finalization times) sorted in ascending order according to their starting time.

3. [3 pt] We can use binary trees in order to represent the trails in a mountainside. The root of the binary tree is the top of the mountain, from which only one or two trails may start. The different trails may divide into two new ones when we walk down the mountainside, and the new trails never connect again with any other. A hiker is on top of the mountain (the root of the tree) and realizes that in some intersections (marked with an 'X' in the tree) there are friends who need his help in order to reach the top. He has to go down to each 'X' and help them reach the top one by one. Using the following header

```
int helpTime(const Arbin<char> &a);
```

code a function that, for any binary tree with the described content, returns the time it will take the hiker help all his friends to reach the top, provided that every trail segment between two intersections takes one hour to be traversed (in each direction).

4. [3 pt] In a school, all the teachers of different subjects with the same group of students have a list of absences. For each student, they mark how many times the student has missed a class (0 if the student has never missed the class). In order to manage this information, they have designed an abtract data type *Absences* with three generating operations:

- *addStudent*, which adds a new student with 0 absences in all the subjects.

- *addAbsence*, which increments 1 unit the number of absences of a student in a given subject.
- *addSubject*, which adds a new list with the same students of the other subjects, initializing them with 0 absences.

To code this ADT, they have decided that the list of absences in a subject is represented by a dictionary, using a `StudentId` (an identifier of the student) as the key and the number of absences of the student in that subject as the associated value. In addition, all the lists of absences are stored in a dictionary that uses `SubjectId` (identifier of the subject) as the key and the list of absences in that subject as the associated value. All the lists of all the subjects for the same group have the same students:

```
class Absences
{public :
    void addStudent(const StudentId& a);
        //adds the student to all the subjects with 0 absences
    void addAbsence(const StudentId& a,const SubjectId& s);
        //adds one absence of a student in a subject
    void addSubject(const SubjectId& s);
        // adds a new subject with all the students
        // of the other subjects with 0 absences

         ...other methods...
 private:
     Dictionary<SubjectId,Dictionary<StudentId,int> > absence_lists;}
```

At the end-of-term meeting, they share their information and they need to add the following operations to the *Absences* ADT:

- *noAbsences*, which returns a list sorted in alphabetical order (over `StudentId`) containing all the students that have not missed any class in any subject.
- *totalAbsences*, which, for a given student, returns the accumulated number of absences in all subjects.

1.[0,5 pt] Select and justify the implementation of the two Dictionaries and explain, according to this decission, what you need from the types `SubjectId` and `StudentId`, and what is the complexity of the three aforementioned generating operations.

2.[0,75 pt] Code the operation *totalAbsences* and indicate its cost.

3.[1,25 pt] Code the operation *noAbsences* and indicate its cost.

4.[0,5 pt] Explain how to improve the representation of the type *Absences* so that the two former operations, *noAbsences* and *totalAbsences*, are more efficient without incrementing the cost of the generating ones.