University of Nice-Sophia Antipolis Master Mathmods
Algorithmics – Final examination 2010–2011

Exam of april 11, 2011

Length: 2h
Lecture notes are allowed

Score :

Name :
Surname :

1 Design and analysis of a string matching algorithm [6 points]

A common problem in algorithmics is the one of finding all occurrence(s) of a pattern string p within another string or body of text T. It occurs for instance in text processing or in bioinformatics. More precisely, the pattern string will be denoted by $p = p[0, \dots, m-1]$; its length is equal to m. The text string is denoted by $T = T[0, \dots, n-1]$; its length is equal to n. Both strings are build over a finite set of characters called an *alphabet* denoted by Σ. We will design and analyse the brute force algorithm for solving this problem.

1. [2 points] Design a procedure `PrefixMatch` which takes as an input p and T and returns true if p is a prefix of T. Let us recall that a word u is a *prefix* of a word w, if there exists a word v (possibly empty) such that $w = u \cdot v$ where $\cdot$ denotes the concatenation of two words.

   Example:

```
PrefixMatch(aba,abaaabaabb)=> true
PrefixMatch(bb,abaaabaabb)=> false
```

```
def PrefixMatch?(p, t)
    result = false
    for i in 0..p.length-1
        if p[i] == t[i]
            result = true
        else
            result = false
            break
        end
    end
    return result
end
```

2. [2 points] Design a procedure `Substring` which takes as an input p and T and outputs the index of every occurrence of p in T or `NIL` if there is none. Of course, you can use `Prefixmatch` repeatedly.

Example:

```
Substring(aba,abaaabaabb)=> 0,4
Substring(bb,abaaabaabb)=> 8
Substring(bab,abaaabaabb)=> NIL
```

```
def Substring(p,t)
    found = false
    long = t.length-1
    for i in 0..long
        if PrefixMatch(p,t[i..long])
            put i
            found = true
        end
    end
    if not found
        return false
        else return true
    end
end
```

3. [2 points] What is the worst-case time-complexity of your algorithm? (Please provide a detailed analysis)

* Prefix match scans input p of size m and does a character by character comparison with T.
Its (worst case) time complexity is thus O(m)

* Substring make a repeated use of Prefixmatch by running Prefixmatch on the subarray starting at index i for $i \in [\![0, n-1-m]\!]$ upper bounded by n
Its (worst case) time complexity is thus O(n·m)

2 Easy subset-sum problems

The goal of this problem is to solve easy instances of the subset-sum problem that we recall below:
INSTANCE : Let $A[0,..,n-1]$ an array of n distinct integer valued items, an integer B, the bound.

QUESTION : Is there a subset of A for which the sum of its items equals B?

There are easy instances of this problem when the elements of A form a *super-increasing* sequence.

Definition 1 *A* super-increasing *sequence is one in which the next term of the sequence is greater than the sum of all preceding tems.*

The sequence $A = [1, 2, 5, 10, 19]$ fulfills the property. This is the case only when A is sorted.

1. (1 point) Design an algorithm which returns `true` when A is sorted and `false` otherwise.

This was already seen in the programming test. By adding to the class Array

```
def sorted?
    sorted = true
    for i in 0.. self.length-2
        if self[i] > self[i+1]
            sorted = false
            break
        end
    end
    return sorted
end
```

2. (1 point) Check the behavior of your algorithm on the array $A = [1, 2, 5, 10, 19]$.

3. (1 point) What is the time-complexity of your algorithm (in the worst case)?

A single loop on an array of size n and thus clearly an O(n) time algorithm.

Since we know that A is sorted, rest to check if its elements form a super-increasing sequence.

4. (1 point) Design an algorithm for checking if A's elements form a super-increasing sequence.

Also by adding to the class Array the method:

```
def superincreasing?
    s = self[0]
    result = true
    for i in self.length-1
        if self[i] ≤ s
            result = false
            break
        else
            s += self[i]
        end
    end
    return result
end
```

5. (1 point) What is the time-complexity of your algorithm (in the worst case)?

Also a single loop on an array of size n which clearly leads to an O(n) time algorithm.

After all the previous carefuls checks, we have to find a polynomial-time greedy algorithm for solving the subset-sum problem. We first consider a useful property expressed in Lemma 1.

Lemma 1 *A valid bound B is uniquely decomposable with elements of A.*

6. (1 point) Prove the statement of Lemma 1 (or admit its result).

By contradiction. If the decomposition of B with elements of A were not unique this would imply that the sum of elements in A doesn't lead to a superincreasing sequence (fix the details)

7. (2 points) By using lemma 1, design an algorithm for solving the subset-sum problem. [Hint: start with comparing the bound B with the elements in A from the greatest to the smallest].

```
def SSsolve(B)
   max = self.length-1
   @result = Array.new
   max.downto(0) { |i|
      if B > self[i]
         @result.push(self[i])
         bound -= self[i]
      end
   }
   return @result if bound == 0
   return false if bound > 0
end
```

8. (1 point) What is the time-complexity of your algorithm (in the worst case)?

Still a single loop on an array of size n giving a time complexity of O(n)

9. (1 point) Use your algorithm for finding a solution to $A = [1, 2, 5, 10, 19]$ and $B = 16$:

19 < 16

10 — $\overset{!}{<}$ → yes @result = [10] bound = 6

5 < 6 → yes @result := [5, 10] bound := 1

2 < 1

1 — $\overset{!}{=}$ → yes @ result = [1, 5, 10] bound = 0

and to see that $A = [1, 2, 5, 10, 19]$ and $B = 22$ has no solution:

like previous question.

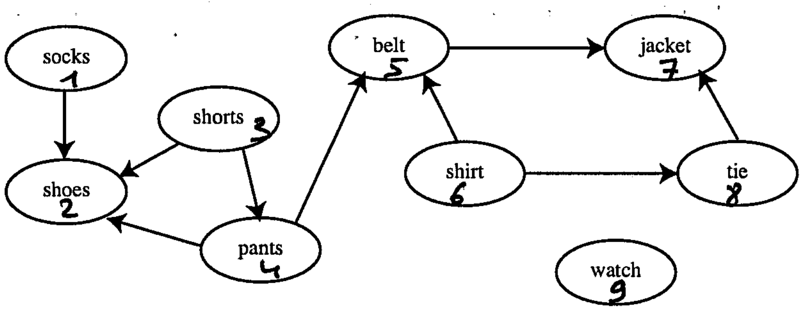
3 Algorithms on graphs [4 points]

This exercise uses the topological sort as described in the 2d lecture. Its behavior is recalled below:

- Run a recursive depth first search on the graph;
- Print each vertex before exiting the depth first search procedure on this vertex;
- After the end of the depth first search on all the vertices, you get a reverse topological sorting.

This exercise is a case study of this particular sort.

1. (2 points) Before breakfast, Mr G. has a problem when getting ready: he sometimes dresses out of order. For example, he might put his shoes on before putting the socks on, so he will have to take the shoes off, put the socks on and than the shoes back on. there is also a shirt, tie, belt, shorts, pants, watch and jacket that have to be put on in a certain order. Help him to dress thanks to a topological sort. Please detail the behavior of the algorithm.



I number the nodes on the figure
The result of the dfs is :

2, 1, 7, 5, 4, 3, 8, 6, 9

Next, I take the mirror image of this sequence (its reverse) which gives

9, 6, 8, 3, 4, 5, 7, 1, 2

and the solution (a solution, actually) is :
watch, shirt, tie, shorts, pants, belt, jacket, socks, shoes

2. (1 point) Can you find (without running the depth first search algorithm again) another ordering for the previous dressing problem?

We can start equally from any node without predecessors (there is no directed edge toward this vertex). Nodes that fulfill this property are Socks, Shorts, Shirt, Watch
Thus there are at least 4 different orderings to solve the dressing of Mr X

Another ordering (rather simple) is to end by the watch instead of starting with

3. (1 point) Can you provide an (obvious) lower bound to the number of solutions for this particular instance of the problem? Please explain which property on the vertices of the graph helps you in finding this bound.

Answer is above