Midterm Examination 2
GEM 1501: Problem Solving for Computing

Thursday 27.03.2014, duration half an hour

Matriculation Number: TEST SOLUTIONS.

Rules
This test carries 10 marks and consists of 5 questions. Each questions carries 2 marks; full marks for a correct solution; a partial solution can give a partial credit.

Question 1 [2 marks].
Explain how bubble sort works and write down its worst case time complexity.

Solution. So sort an array of n elements by Bubble Sort, one does n scans of the array where each scan goes from the beginning to the end of the array and swaps an element with the neighbour whenever the element with the lower index is larger than the element with the larger index. Due to this, in each scan, large elements bubble up until they meet a larger one, hence the name of this sorting algorithm. One might stop the algorithm early in the case that a scan does not find any elements to swap.

The worst case complexity is $O(n^2)$, as in the case that the first half of the array has $n/2$ large elements and the second half has $n/2$ small elements, one needs at least $n/2$ scans, as each scan moves only one large element accross the border between the two halves.

Question 2 [2 marks].
Determine the worst case runtime complexity of the following program using the parameter n being the number of array elements of a:

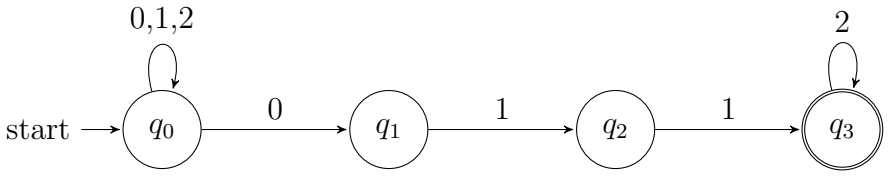
```
function findthree(a)
  { var n = a.length; var m = false;
    var i; var j; var k;
    for (i=0;i<n;i++)
      { for (j=i+1;j<n;j++)
          { for (k=j+1;k<n;k++)
              { if ((a[i] != a[j]) && (a[j] != a[k])
                    && (a[i] != a[k])) { m = true; } } } }
    return(m); }
```

Write down the worst case time complexity in Θ-notation and indicate whether a better run time is possible with another algorithm. If so, give the program; if not, say why it cannot be done.

Solution. There are three nested loops. If one restricts the range of the i in the first loop from 0 to $n/3$, the j in the second loop from $n/3$ to $2n/3$ and the k in the third loop from $2n/3$ to n, then one gets as a lower bound that the nested loops go $n^3/8$ times through the innermost body of the loops. Thus the runtime is $\Theta(n^3)$. This can be improved with the following linear time program.

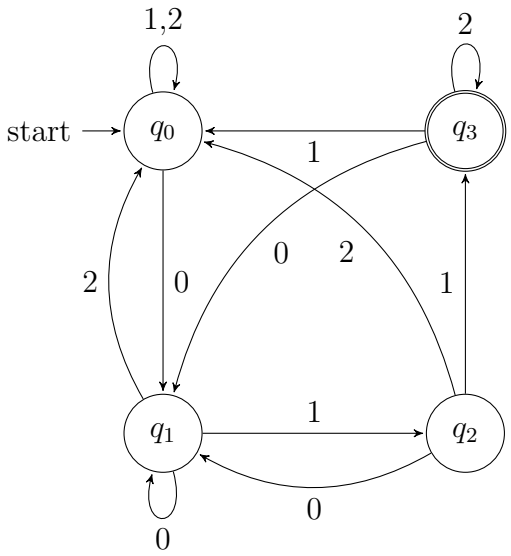
```
function findthree(a)
  { var n = a.length; var m = false;
    var i=1;
    while ((i<n)&&(a[i]==a[0])) { i++; }
    var j=i+1;
    while ((j<n)&&((a[j]==a[0])||(a[i]==a[j]))) { j++; }
    if (j<n) { m = true; }
    return(m); }
```

Question 3 [2 marks].
Consider a non-deterministic finite automaton with the following state transition diagramme.



Make an equivalent deterministic finite automaton.

Solution.



Question 4 [2 marks].
Some function f satisfies $f(n) > 0$ for all n and $f(n+m) = f(n) \cdot f(m)$. Furthermore, let

$$g(n) = f(0) + f(1) + \ldots + f(n)$$

for all n. The following algorithm uses a program for f as a subroutine and computes g using a divide-and-conquer algorithm.

```
function f(n) { ... }

function g(n)
  { if (n < 1) { return(f(0)); }
    if (n < 2) { return(f(0)+f(1)); }
    if (n < 3) { return(f(0)+f(1)+f(2)); }
    var m = Math.floor(n/2); var k = f(m+1);
    return(g(m)+k*g(n-m-1)); }
```

When it computes $g(n)$, this program uses $\Theta(n)$ many calls of f. Use dynamic programming or a similar method, make a new program which computes $g(n)$ using only $\Theta(\log(n))$ many calls of f.

Solution. One way to solve is to use a dynamic array which stores those intermediate values which are computed. The reason is that the program actually computes only few values of g and these can be stored and retrieved from the array `a`.

```
function f(n) { ... }

var a = new Array();

function g(n)
  { if (n in a) { return(a[n]); }
    if (n == 0) { a[n] = f(0); }
    else if (n == 1) { a[n] = f(0)+f(1); }
    else if (n == 2) { a[n] = f(0)+f(1)+f(2); }
    else { var m = Math.floor(n/2); var k = f(m+1);
           a[n] = g(m)+k*g(n-m-1); }
    return(a[n]); }
```

Question 5 [2 marks].
Let a graph be given by an array `edge` such that each entry in `edge[k]` is an array `[v,w]` being equal to `[edge[k][0],edge[k][1]]` representing an edge going from vertex `v` to vertex `w`. Make a function which has as input an array `edge` and a starting vertex `v` and as outputs a vertex `w` such that `w` can be reached from `v` and either `w` can also be reached from itself by a loop (perhaps going through other vertices) or `w` is a sink, that is, a vertex without outgoing edge.

For example, consider the case where `edge` contains the three array elements `[0,1]`, `[0,2]` and `[1,2]`. If the algorithm is run with inputs `edge` and `0` then it has to return `2` as `2` is the sink.

Solution. The goal of this exercise is to implement a greedy algorithm which is also keeping track of all the nodes visited before (in an array `u`). Whenever possible, it follows from a node an outgoing edge. Whenever it ends up in a node w which was visited before or which does not have an outgoing edge, it returns this node.

```
function search(edge,v)
  { var w = v;
    var u = new array();
    var n = edge.length;
    var m = 0;
    while ((m<n)&&(!(w in u)))
      { if (w == edge[m][0])
          { u[w] = 1; w = edge[m][1]; m = 0; }
        else { m++; } }
    return(w); }
```

END OF EXAMINATION.