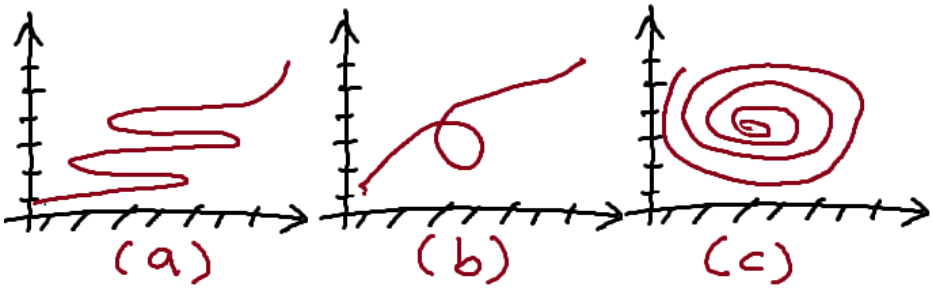
Data Structures and Analysis of Algorithms — Quiz 1
November 6, 2013

- *This is page 1. The exam has 5 pages.*
- The exam is open book; "Introduction to Algorithms" by Cormen is allowed.
- *Write your name, student id and today's date.*
- *You have one* **mandatory skip ticket**. *You have to use it towards one of the last two parts as indicated.*
- The mark on each question estimates the time you should spend on the problem in minutes. Use that to figure out when to move to the next question.
- Read all questions before you start. This will help you know where to start. If you feel stuck, you probably misunderstood the question. Read it again. Still stuck, ask for clarification. Do not leave a question without an answer and show your work even if partial for partial credit.
- Be confident and do not look around.
- Full grades are awarded for descriptions fine enough to be readily translated into code. Less accurate and refined descriptions will get partial credit.

Part zero. Estimation (5 pts.) Reason about the time needed to traverse the curve lines in the diagram. Which could take the shortest time? the longest time? why? Suggest reasonable realistic situations where such curves occur.



solution: The curly lines in c ***take longer than those in*** a***. The line in*** b ***could be smaller than both, if we do not go into the loop, or go in it only one or two times, but if we also could go into the loop several times.***

(a) could be a road in a valley, (c) a road ascending around a hill, and (b) could be an highway with entrance and exit service roads.

Part one. Counters (15 pts.) Consider a counter from 0 to $n-1$ that uses $\log_2 n$ bits. Consider the flip of a bit (from 0 to 1 or from 1 to 0 as one operation. What is the worst case required running time of incrementing the counter by 1 (going from i to $i+1$)? What is the running time of n calls to increment the counter? ***solution: the worst case requires flipping all bits as in flipping*** 0111111 ***to*** 1000000***. The number of bits is*** $\log_2 n$***.***

Amortized behavior analysis is needed here to compute the behavior of n ***calls to increment. When going steady from*** 0 ***to*** n ***with*** n ***successive calls to increment the counter, we rarely flip all the bits. We flip the least significant bit*** n ***times as it changes from odd to even numbers. We flip bit*** 1 ***only*** $n/2$ ***times. We flip bit*** i $n/2^i$ ***times. And we flip the most significant bit only once.***

In total we flip the bits $\Sigma_{i=1}^{\log_2 n} n/2^i$ ***which approaches*** $2n$ ***as*** n ***grows.***

Part two. Asymptotic order (10 pts.)

Sort the following functions and place them in ascending order of growth rate. That is function $f_2(n)$ follows function $f_1(n)$ in the list iff $f_1(n) = \mathcal{O}(f_2(n))$.

$3^n \quad 2n \quad n^2 \log n \quad n(\log n)^2 \quad n^{1.4} \quad n^n \quad n^2 \quad n!(n-1)! \quad 3^{3^n} \quad 2^{2^n}$

solution: $2n \quad n(\log n)^2 \quad n^{1.4} \quad n^2 \quad n^2 \log n \quad 3^n \quad n!(n-1)! \quad n^n \quad 2^{2^n} 3^{3^n}$

Part three. Sorted Doubly Linked lists (15 pts.)

- Provide a complete recursive definition of a *sorted doubly linked list* (SDLL) with a *head* and a *tail* and where the base case is an *empty* SDLL.
- Write a recursive algorithm `isSortedDoublyLinkedList` that takes the head and the tail of an *SDLL* and checks whether it respects the definition of an SDLL. If it is, then the algorithm returns 0, otherwise, it returns the offending node.
- Modify the algorithm so that it also computes the pointer to the median of the SDLL (the element just in the middle).

solution: An SDLLNode is a tuple $\langle d, next, prev \rangle$ ***where*** d ***denotes data, and*** $next$ ***and*** $prev$ ***designate pointers to DLLNodes.***

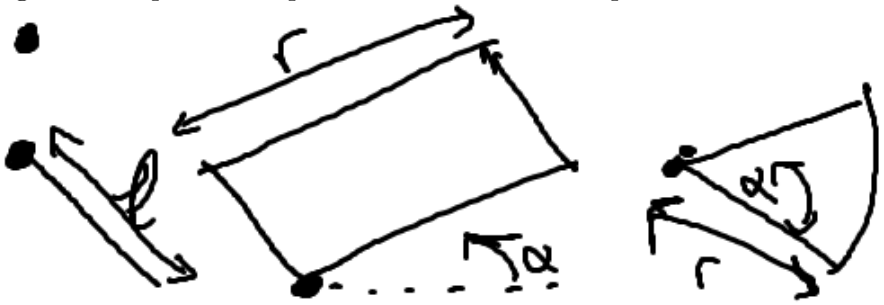
The value nil ***denotes invalid pointer value. A pointer with a*** nil ***value denotes an*** **empty SDLL.** ***An SDLL is a couple*** $(head, tail)$ ***where*** $head$ ***and*** $tail$ ***are pointers to SDLLNodes. If SDLL is a empty then*** $head = tail = nil$***. Otherwise,*** $head$ ***points to a node where*** $(head.next, tail)$ ***is an SDLL, and*** $(head, tail.prev)$ ***is an SDLL. Sortedness needs to be conditional on the validity of the next and prev pointers and checks the data member*** d***. The rest follows.***

Part four. Shapes and inheritance (40 pts.)

MUST DO (a), (b), and (c) FOR LABS TO COUNT.

(d) is pseudo-code for 18 pts. and you can do it on its own.

A shape is a basic concept that you can query for its number of sides, and its surface. We are interested in shapes that are defined and can be built by moving other shapes. Points, lines, parallelograms, and arcs are all shapes. A line is defined by a starting point moving in one direction (defined by an angle) for a specific length. A parallelogram is a line moving in one direction (given by an angle) for a specific length. An arc is a line moving in rotation around its starting point for a specific angle. A box is a special parallelogram. A square is a special box. A circle is a special arc.



(a) Write a base C++ class `shape` that supports the basic concept. Write C++ classes that inherit from `shape` and from each other to describe points, lines, parallelograms, arcs, boxes, squares, and circles.

(b) Declare and support a member method
`shape & move( double angle, double magnitude, bool fixStarting)` that creates a shape out of the motion of the current shape if applicable (only applicable in point and line) and reduces to a point if the motion does not produce a supported shape. For example, ℓ = `p.move(0,5,false)` creates a horizontal line of length 5 that starts at point p, `l.move(45,6,false)` creates a parallelogram, and ℓ`.move(45,0,true)` creates an arc.

(c) Use one of the C++ STL containers that you are familiar with to create a container of shapes. Use a pseudo-random generator to create a number of starting points, angles, and magnitudes and generate shapes out of them and insert the shapes into the selected container.

(d) **Pseudocode.** You are given a set of shapes $\{s_1, s_2, \ldots, s_n\}$. When two shapes intersect, either one shape kills the other, or they both give birth to new shapes. In this problem we are only interested in computing the survivor shapes and we ignore the newborns. Lines and pointee arcs (strictly smaller than half circles) are dangerous. They look like knives.

If the head of a dangerous shape a is inside another shape x and x is not dangerous, then x is declared dead. If the heads of two dangerous shapes are inside each other, then the more pointee shape (smaller angle, sharper knife) wins and the other dies. If both are as sharp, then both die. **If a shape is dead, then it can not kill another shape**.

Suggest a data structure to represent the set of shapes, and design an efficient algorithm that computes the surviving shapes by deleting the dead shapes. Analyze and describe the asymptotic running behavior of your algorithm.

Deep and redundant shape reflections (not graded.)

- How many sides does a point have?
- How many sides does a line have? what is sharper than a line?
- How many sides does a circle have? how many sides does an arc have?
- How many sides does a point have?
- Can you think of a shape with one side? two sides?

Part five. Abo-Algo in landscape architecture (40 pts.)

No one knows why Abo-Algo ended up in a landscape architecture course. It could be that he made a mistake when he intended to pick a computer architecture course. It also could be that the registrar's website had a bug that sent him there.

Anyway, Abo-Algo found himself carefully fitting expensive pieces of marble of different *shapes* into a pathway of a royal garden. For simplicity, Abo-Algo assumed that the pathway is of rectangular shape and that the pieces are also all of rectangular shape. *(no tricks, not necessarily related to the previous problem.)*

Given n rectangular marble pieces, each defined in terms of its width and length, and given the width and length of the pathway, write an algorithm that uses the maximum number of marbles to cover the pathway without an overlap.

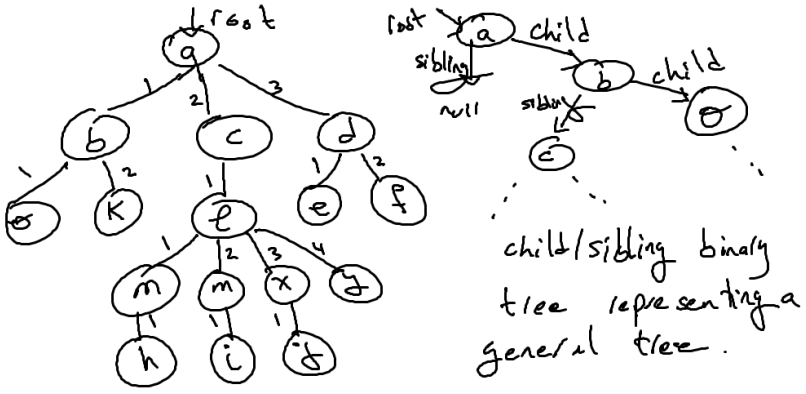
Your algorithm should be as efficient as possible. ***solution: classical tiling problem. look in the book. briefly, one way is to generate all layouts of rectangles with no overlaps, and pick the one with the maximum rectangles. then you can eliminate the layouts that are symmetrical. for example, starting with an empty pathway, one needs only to consider placing rectangle*** r_1 ***in one corner of the pathway as the other three corners are just symmetrical. This***

eliminates 3/4 of the layouts. However, the problem is still intractable in general even though terminates in reasonable time in interesting practical situations.

Part six. Trees and binary trees (30 pts.) ***SKIP TICKET***

We can always represent a general tree where each node has several children with a binary tree where a node has at most two descendants. In particular, consider a binary tree node with two descendants, where one descendant denotes the first child of the node, and the second descendant denotes the next sibling of the node.

- Consider the general tree in the figure below. Fully transform it into a corresponding binary tree (complete the figure manually).



- Convince your self that the i^{th} child of a node n is the $(i-1)^{th}$ sibling of a its first child. Write code that returns the i^{th} child of a node.

- Provide an algorithm that takes the root of a binary tree representing a general tree and a node in it, and returns the depth of the node as defined in the general tree. You can assume that each node in the binary tree has a pointer to its ancestor in the binary tree. For example, a call with root and x as the node should return 3 as depth.

- Provide an algorithm that takes a node in the binary tree representing the general tree and that returns the parent of the node as defined in the general tree. For example, a call with c as the node should return a as the parent. While a call with o as the node should return b as the parent.

Part seven. Graph traversal (40 pts.) ***SKIP TICKET***

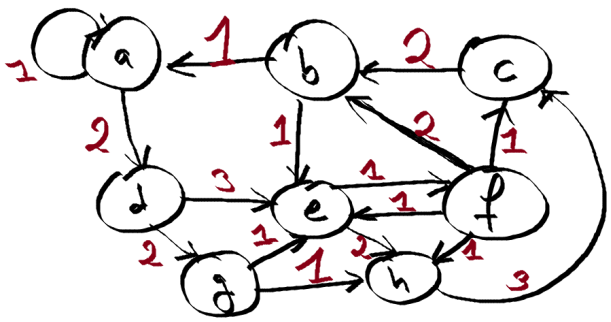
The `traverse` algorithm below takes a graph g and a node a in the graph and traverses the graph. The symbols ϕ and E represent sets of nodes and edges, respectively. The computation of the algorithm and its running time highly depend on the choice and implementation of these sets and also on the mechanism used to mark a node visited.

```
// traverse a graph g starting from node a
Traverse (g, a)
//modify to be Traverse (g, a, b)
  let φ be an empty set of nodes
  φ += a // insert a into φ

  while (φ is not empty)
    u = φ.pick() // let u be an element in φ
    φ -= u // remove u from phi

    mark u as visited in g
    print u
//  path+=u
//  if u = b return path

    let E be the edges of u in g
    foreach edge e = (u, v, ℓ) ∈ E
      // where v is the destination node
      //   and ℓ is the label
      if v is visited continue;
      φ += v // add v to φ
```



(a) Consider the graph in the figure. Consider that ϕ was implemented such that `pick` always returns the node with the minimum incoming edge. For example, if nodes d and e were both in ϕ, pick will return e since it has an incoming edge from b with a lower label than the only incoming edge to d from a. Show the printed output of the algorithm. (You do not need to implement the structure). ***solution: a,b,e,f,c,h,d,g or a,b,e,f,h,c,d,g.***

(b) Now assume that ϕ is a queue. Show the printed output of the algorithm. ***solution: breadth first: a, [b,d], [e,g], [f,h], c.***

(c) If ϕ was a stack, E was a linked list, and we used an additional boolean variable in the node to mark the node as visited. What is the running time of the algorithm. ***solution: the outer loop goes over all nodes. pop an push in a stack take constant time. and the inner loop goes over all edges. so the running time is*** $n + m$ ***where*** n ***is the number of nodes and*** m ***is the number of edges.***

(d) Let ϕ be a queue again. Modify the algorithm to take two nodes x and y, compute and return the first path (*sequence of nodes*) the algorithm finds that takes you from x to y . ***solution: find modifications in comments above.***

(e) Back to part (a), let ϕ be implemented as a balanced search tree that takes $\theta(\log n)$ to return the minimum, insert a node, and delete a node. What would be the running time of the algorithm? ***solution: following the same logic in (c), the algorithm takes*** $n \log n + m$

Best of luck!