Foundations of Computing – Algorithms and Data Structures, Exam Spring 2012 (June 1)

Jesper Larsson, ITU

What to bring You can bring any written aid you want. This includes the course book and a dictionary. You cannot bring electronic aids (such as a laptop) or communication devices (such as a mobile phone). If you really want, you can bring an old-fashioned pocket calculator (not one that solves recurrence relations).

Where to write answers All questions are supposed to be answered on the problem sheet. If for some reason you are unable to write on the problem sheet (e.g., if you need to fully revise an answer), clearly mark this on the problem sheet and use an extra sheet of paper for your answer.

Where to write your name On all papers with answers.

What to check. In the multiple-choice questions, there is one and only one correct answer. Only checking the correct answer, and no other, gives you full score. However, the scoring method[1] is defined so that there is no point in guessing one answer if you have no idea which is correct. Instead, to demonstrate partial knowledge, you are allowed to check two or more boxes. If you don't check anything, or check all boxes, your score is zero. Otherwise, if the correct answer is among your checked answers, your score is positive; and if the correct answer is *not* among the checked answers you get a negative score. If you check boxes at random, your expected score is zero.

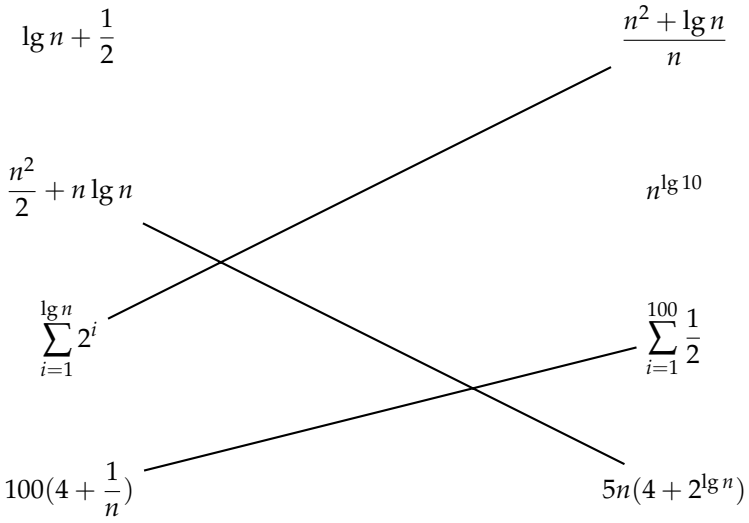
Score examples for a multiple-choice question worth 1 pt.:

☑ Correct. ☐ Incorrect ☐ Incorrect ☐ Incorrect	+1	☐ Correct. ☑ Incorrect ☐ Incorrect ☐ Incorrect	-0.3333
☑ Correct. ☑ Incorrect ☐ Incorrect ☐ Incorrect	+0.5	☐ Correct. ☑ Incorrect ☑ Incorrect ☐ Incorrect	-0.5
☑ Correct. ☑ Incorrect ☑ Incorrect ☐ Incorrect	+0.2075	☐ Correct. ☑ Incorrect ☑ Incorrect ☑ Incorrect	-0.6226

[1] Gudmund Skovbjerg Frandsen, Michael I. Schwartzbach: A singular choice for multiple choice. SIGCSE Bulletin 38(4): 34–38 (2006)

1.

(a) (*3 pt.*) Draw lines to connect 3 of 4 expressions on the left side with the expression on the right side that denote the same order of growth (i.e., disregarding both lower-order terms and constant factors).



(b) (*2 pt.*) Order the following expressions from the asymptotically smallest to the asymptotically largest (in left to right order):

A. $2^n \cdot 10^{-6}$

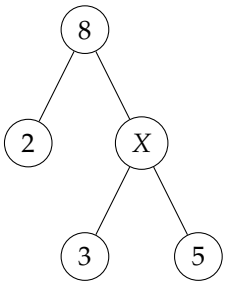
B. $\sqrt{n}$

C. n

D. $n^2 / \lg n$

Give your answer here:
B < C < D < A

2. Consider the following tree:



Which, if any, of the following are possible choices for X to make the tree…

(a) (*1 pt.*) …a valid binary search tree?

☐ 4, but not 7 ☐ 7, but not 4 ☐ Either 4 or 7 ☑ Neither 4 nor 7

(b) (*1 pt.*) …a valid binary heap?

☐ 4, but not 7 ☑ 7, but not 4 ☐ Either 4 or 7 ☐ Neither 4 nor 7

(c) (*1 pt.*) …a valid part of a trie, with keys consisting of strings of digits?

☐ 4, but not 7 ☐ 7, but not 4 ☑ Either 4 or 7 ☐ Neither 4 nor 7

3. (*1 pt.*) Suppose we need to implement a stack for some application application. In addition to general expectations on software development (efficiency, bug-free code etc.) we want to keep the total space usage as low as possible. We know in advance how many, N , elements will be pushed on the stack, and that all N elements are pushed before any are popped. What is the most suitable stack representation?

- ☐ A linked list of elements in the order they were pushed
- ☑ An array of size N holding elements in the order they were pushed
- ☐ An array of size N holding elements according to the van Emde Boas tree layout
- ☐ A Huffman tree

4. Consider the following piece of code.

```
for (int i = 0; i < a; i++) {
    for (int j = 0; j < b; j++) {
        output(convert(i) + convert(j));
    }
}
```

Which is the correct expression for the number of calls to the method `convert`...

(a) (*½ pt.*) ...if $\mathtt{a} = \mathtt{b} = N$?

☐ $\sim N$ ☐ $\sim 2N$ ☑ $\sim 2N^2$ ☐ $\sim 4N^2$

(b) (*½ pt.*) ...if $\mathtt{a} = 1, \mathtt{b} = N$?

☐ $\sim N$ ☑ $\sim 2N$ ☐ $\sim 2N^2$ ☐ $\sim 4N^2$

(c) (*½ pt.*) ...if $\mathtt{a} = N, \mathtt{b} = 1$?

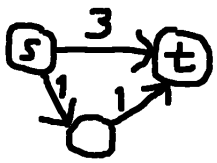
☐ $\sim N$ ☑ $\sim 2N$ ☐ $\sim 2N^2$ ☐ $\sim 4N^2$

(d) (*½ pt.*) ...if $\mathtt{a} = \mathtt{b} = N/2$?

☐ $\sim \frac{1}{2}N$ ☐ $\sim N$ ☐ $\sim \frac{1}{4}N^2$ ☑ $\sim \frac{1}{2}N^2$

5. (*2 pt.*) Consider the possibility of a weighted digraph with three vertices – one source s, one sink t, and one other vertex – in which breadth-first search would fail to find a shortest path from s to t, but Dijkstra's algorithm would succeed. Either draw such a graph, or explain why one cannot exist.

Give your answer here:



6. Consider the following Java class:

```
public class SimpleIntContainer {
    private class Node {
        int i;
        Node next;

        Node(int i, Node next) { this.i = i; this.next = next; }
    }

    private Node head = null;

    public void add(int i) {
        if (!contains(i)) head = new Node(i, head);
    }

    public boolean contains(int i) {
        for (Node p = head; p != null; p = p.next) {
            if (p.i == i) return true;
        }
        return false;
    }
}
```

(a) (*1 pt.*) Describe, in one or two sentences, the function of the class from the perspective of a client (just *what* it does, not *how* it does it, and not how much time or space it uses for it).

Give your answer here:
Keeps a set of integers (duplicates discarded), which can be queried only for presence of an element.

Assume that N distinct elements have been added in some order, every possible order equally likely. Then:

(b) (*1 pt.*) What is the average time complexity of a call to the method `add`?

☐ Constant ☐ Logarithmic in N ☑ Linear in N ☐ Quadratic in N

(c) (*1 pt.*) What is the average time complexity of a call to `contains` that returns *false*?

☐ Constant ☐ Logarithmic in N ☑ Linear in N ☐ Quadratic in N

(d) (*1 pt.*) What is the average time complexity of a call to `contains` that returns *true*?

☐ Constant ☐ Logarithmic in N ☑ Linear in N ☐ Quadratic in N

(e) (*1 pt.*) Consider replacing the previous code for the method `contains` with the following pair of methods:

```
public boolean contains(int i) {
    return contains(head, i);
}

public boolean contains(Node x, int i) {
    return x != null && (x.i == i || contains(x.next, i));
}
```

What, if any, is the effect of the change?

☐ The method sometimes returns *true*, where it would previously have returned *false*

☐ The method sometimes returns *false*, where it would previously have returned *true*

☐ The function of the method is unchanged, but its time complexity changes with a factor log N

☑ Both the function of the method and its time complexity, disregarding constant factors, are unchanged

(f) (*2 pt.*) Consider changing the internal representation of the data structure to obtain a smaller time complexity of `contains` (in relation to the original; no reference to (e) is intended). In one sentence, describe how this could be accomplished, or state why it is not possible.

Give your answer here:
The simplest and most effective change should be to use a hash table instead of a linked list.
A (preferably balanced) binary search tree is also an acceptable answer.

7. Consider the problem of sorting a huge array of uniformly random 128-bit numbers. The array resides in memory (RAM), but is much bigger than will fit into any cache.

(a) (*1 pt.*) Which of the following algorithms would be the most suitable:

☐ Selection sort ☐ Insertion sort ☑ Quicksort ☐ Heapsort

(b) (*3 pt.*) Another good method (plausibly better in many cases) would be to sort on the first 32 bits in two passes of LSD radix sort (16 bits per pass), and then finish with running a single insertion sort on the whole array. Explain why insertion sort is a good choice to finish with in this case.

Give your answer here:
With random numbers, most of the elements will differ in the first bits. So, after the two radix sorting passes (which are done quickly, in linear time) the array is mostly sorted. Just a few local adjustments remain, and insertion sort works greatin that situation.

8. (*2 pt.*) We are given a file consisting of several million bytes. Out of the 256 possible byte values, only 10 are used, namely the ten digits (byte values 48–57). Apart from this restriction, the file appears completely random: all the digits are equally probable, and there are no repeating patterns beyond what can be expected in a random string.

We encode the file using a compression program based on Huffman coding. Approximately how large can we expect the encoded output to be in relation to the original file size?

☑ 43% ☐ 50% ☐ 100% ☐ 110%

9. Consider the problem of finding a specific element in an array of N integers, using pairwise compare operations. What is the worst case lower bound for the number of compares...

(a) (*1 pt.*) ...if we know nothing of the elements?

☐ No other lower bound than a constant can be found
☐ $\sim \lg N$, because in the worst case we can cut down the possible candidates by at most half by each comparison
☑ $\sim N$, because we need to look at all elements
☐ $\sim N \lg N$, because this is the lower bound of sorting

(b) (*1 pt.*) ...if the elements in the array are known to be reverse sorted order (i.e, larger elements before smaller ones)?

☐ No other lower bound than a constant can be found
☑ $\sim \lg N$, because in the worst case we can cut down the possible candidates by at most half by each comparison
☐ $\sim N$, because we need to look at all elements
☐ $\sim N \lg N$, because this is the lower bound of sorting

10. (*2 pt.*) For sorting N elements, a lower bound stating that the number of operations is at least proportional to $N \lg N$ can be proven, by considering the height of a decision tree where each node corresponds to comparing two elements. Yet, the number of operations used by key-indexed counting has order of growth $N + R$ for alphabet size R, which is linear in N in the case where R is at most linear in N. Which of the following explains this discrepancy?

- ☐ The $N + R$ bound for key-indexed counting applies only on average, while the general $N \lg N$ lower bound holds in the worst case.
- ☑ An array indexing operation for alphabet size R has R different outcomes, while a compare operation has only two. The number of levels in the decision tree for key-indexed counting is thus reduced by a factor $\lg R$.
- ☐ R must be at least proportional to $N \lg N$ in order for the decision tree to have $N!$ leaves, and therefore height $N \lg N$. Hence, the proof does not apply when R is only linear in N.
- ☐ Key-indexed counting applies to sorting only single characters, and is thus a component of string sorting. For sorting the complete strings, the $N \lg N$ time bound applies.

11. The following is a short implementation (lacking in error checking and generality) of a symbol table data structure that maps positive integers to other integers.

```
public class PosIntMap {
    private int[] a = new int[2 * 1000];

    private int getPosition(int key) {
        int p = key % 1000;
        while (a[2 * p] > 0 && a[2 * p] != key) { p = (p+1) % 1000; }
        return p;
    }

    public void insert(int key, int value) {
        int p = getPosition(key);
        a[2 * p] = key;
        a[2 * p + 1] = value;
    }

    public int find(int key) {
        return a[2 * getPosition(key) + 1];
    }
}
```

(a) (*1 pt.*) What is the data structure used as the symbol table implementation?

☐ An unsorted array ☐ A sorted array ☐ A trie ☑ A hash table

(b) (*1 pt.*) What sequence of values is printed by the following operations?

```
PosIntMap m = new PosIntMap();
m.insert(1, 4);
m.insert(1001, 4004);
m.insert(1, 5);
StdOut.println(m.find(1001));
StdOut.println(m.find(1));
StdOut.println(m.find(2));
```

Give your answer here:
4004, 5, 0

(c) (*1 pt.*) Assume that we have called `insert` N times, where $N < 500$, all with different positive integers as the *key* value, which were chosen uniformly at random from the range of positive integers. What is the expected ("average") time for the $N + 1$st insertion?

☑ Constant ☐ Logarithmic in N ☐ Linear in N ☐ Quadratic in N

(d) (*1 pt.*) Same question as (c), except that the inserted key values are $1017, 2017, 3017, \ldots, 1000N + 17$?

☐ Constant ☐ Logarithmic in N ☑ Linear in N ☐ Quadratic in N

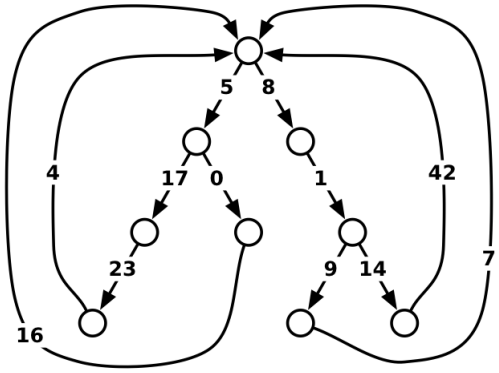
(e) (*1 pt.*) Given that N, the number of inserted keys, again chosen uniformly at random, is close to but less than 1000, what is the expected time for calling `find(`x`)`, where x is a value that has *not* been inserted?

☐ Constant ☐ Logarithmic in N ☑ Linear in N ☐ Quadratic in N

(f) (*1 pt.*) What happens if we call `insert` one thousand times, each time with a different positive integer as the *key* parameter?

Give your answer here:
The hash table fills up, and we run out of slots. In the next insertion, getPosition goes into an infinite loop.

12. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight



A looped tree.

In this example, the shortest path from the leftmost leaf to the rightmost leaf is 27.

(a) (*1 pt.*) Given a looped graph and two vertices s and t, we wish to find the shortest path from s to t. Clearly, Dijkstra's algorithm solves the problem. What is the running time in the number of nodes N?

(b) (*5 pt.*) Design a faster algorithm for this problem. If you want, you can make use of existing algorithms, models, or data structures from the book. Estimate the running time of your construction. Be short and precise.

Give your answer on the next page.

Give your answer here:

(a)

Proportional to $N \lg N$. (Note that number of edges is no more than $2N$.)

(b)

1. Find out if t is below s on the same "leg", in which case we have a unique path. Disregarding loopback edges, do DFS or BFS from s. This takes linear time. If t is found, we found the unique path. An alternative is to invert the graph, so we can go backwards to find s from t, also in linear time. If a path from s to t is found without going through the root, we are done. Otherwise continue:
2. Find the unique path from the root to t. Disregarding loopback edges, do DFS or BFS from the root. Finds the path from the root to t in linear time. An alternative is to invert the graph, so we can go backwards to find the path, also in linear time. What remains is to go from s to the root.
3. Find the shortest path from s to the root. The shortest paths from all nodes to the root can be found "bottom up", choosing the shortest of two at each branching node. Gives us the path in linear time. An alternative is to disregard outgoing edges of the root (making the graph acyclic), and run the the topological-sort-order shortest-path algorithm to find the shortest paths from s. Also linear time.

(Note that using the Bellman–Ford algorithm is not a correct answer, as this is no improvement in general. Looped trees exist where it is worse than for Dijkstra. Finding an example is left as an exercise.)