NAME:__________________________STUDENT NUMBER: ________________.

# DEFERRED AND SUPPLEMENTAL FINAL EXAMINATION

COMP-250 A – Introduction to Computer Science
School of Computer Science, McGill University

April 15th 2004, 14:00-17:00
Examiner: Mathieu Blanchette

**INSTRUCTIONS**
Write your name at the top of this page.
Answer directly on exam paper
Two blank pages are added at the end in case you need extra space.
This examination is worth 50% of your final grade.
The total of all questions is 100 points.
The value of each question is found in parentheses next to it.
This is an open book exam though sharing materials with other students is *not* permitted.
No calculator or laptop computer, cell phones, etc. is allowed.
This exam comprises 13 pages, including the cover page.
All logs are in base 2.

**SUGGESTION: READ ALL THE QUESTIONS BEFORE YOU START! THE NUMBER OF POINTS IS NOT ALWAYS PROPORTIONAL TO THE DIFFICULTY OF THE QUESTIONS. SPEND YOUR TIME WISELY! GOOD LUCK!**

Grading scheme:

| | |
|---|---|
| 1. | / 30 |
| 2. | / 6 |
| 3. | / 10 |
| 4. | / 16 |
| 5. | / 12 |
| 6. | / 13 |
| 7. | / 13 |
| Total. | / 100 |

## Question 1: (30 points, 2 points each) True or False

Specify whether the following statements are true or false. Give a two-line justification for each. Credits will be given only if the justification is correct.

*a)* If $f(n)$ is $O(h(n))$, then $2^{f(n)}$ is $O(2^{h(n)})$

*b)* $n + \sin(n)$ is $O(10 + \cos(n))$

c) If an algorithm *A* runs in time $O(n \log n)$ and an algorithm *B* for the same problem runs in time $O(n^2)$, then for any $n>0$, there must exists an input of size *n* on which algorithm *A* is faster than algorithm *B*.

d) Although many sorting algorithms have average-case running time $O(n \log n)$, they all have worst-case running time $O(n^2)$.

e) You have a small computer with a memory that can store at most 1000 integers. Then, it might be better to sort an array of 900 integers using selectionSort than using mergeSort (as implemented in class).

f) Suppose a hash table has *K* buckets, the buckets are dictionaries implemented with a balanced binary search tree, and the hash table contains a total of *N* elements. Then, under the best possible choice of hash function, any find(key) operation will take time $O(N/K \log(N/K))$.

g) In a country where coins have values 1 ¢, 2 ¢, 5 ¢, and 6 ¢, the greedy algorithm for making change is not optimal.

h) To increase the Google pageRank of your homepage, you should include in your homepage links pointing to many important web sites like yahoo, amazon, mcgill.

i) If someone discovered a polynomial-time algorithm to solve the k-CLIQUE, decision problem, then we would also have a polynomial algorithm for SAT.

j) A decision problem cannot be at the same time NP-Complete and undecidable.

k) A Monte Carlo algorithm may have a non-zero probability of returning an incorrect result, but this probability can be made as small as desired at the price of longer running time.

l) Alice sends an encrypted message to Bob using the RSA public-key cryptography system. If Eve had access to the encrypted message and was able to factorize Alice's public key, then Eve would be able to decrypt the message.

m) The regular expression $(ab)^*b(aaa+bb)^*a(b)^*$ contains the string ababababbbaaaaaabba

n) In a two-player game, a position is winning for player A if and only if all possible moves available to A from that position lead to winning positions for A.

o) The human genome is the most powerful, adaptable, and efficient program that exists.

## Question 2. (6 points) Linked lists

Why does *removeLast*() on a single-linked list takes time *O(n),* where *n* is the number of nodes in the list. Why does it take time *O(1)* on a doubly-linked list? Answer in a short paragraph for each questions. You *don't* need to fill the page.

## **Question 3:** (10 points) Data structures and algorithms

Consider the following problem:
**Given:** An unsorted array *A* of *n* distinct integers, and an integer *z*
**Problem:** Return the number of pairs of indices *i* and *j* between 0 and *n*-1 such that *A[i]+A[j] = z*.

Write an algorithm that solves this problem in worst-case time O(*n* log *n*).
You can use any of the algorithms and data structures seen in class without redefining them.

## **Question 4.** (16 points) Towers of Providence

The *Towers of Providence* is a variation of the classic Towers of Hanoi problem, where the only difference is that there are now four rods instead of three. Rods are denoted 1, 2, 3, and 4, and we have *n* disks of different sizes. Originally, all the *n* disks are on rod 1, stacked in decreasing size from bottom to top. Our goal is to transfer all the disks to rod 4, and the rules are that we can only move one disk at a time, and no disk can be moved onto a smaller one.

Assume that you are given two simple methods *firstDifferent*(*i, j*) that returns the index of the first rod other than *i* and *j*, and *secondDifferent*(*i , j*) that returns the index of the rod other than *i, j*, and *firstDifferent(i, j)*. For example, *firstDifferent*(1,4) = 2, *secondDifferent*(1,4) = 3, and *firstDifferent*(2,4) = 1, *secondDifferent*(2,4) = 3.

We can solve this problem with a recursive method.
To move *n* disks from rod *i* to rod *j*:

```
If n = 1, move this disk directly to rod j, and we are done.
Otherwise (n > 1), perform the following steps:
    Let a = firstDifferent(i,j), b = secondDifferent(i,j)
    Transfer the top n - 2 disks from rod i to rod a
    Move the top disk from rod i to rod b
    Move the top disk from rod i to rod j
    Move the top disk from rod b to rod j
    Transfer the n-2 disks from rod a to rod j
```

**a) (5 points)** Write the pseudocode for a recursive algorithm that prints the sequence of moves needed to move the top *n* disks from rod *i* to rod *j*. Assume the methods *firstDifferent* and *secondDifferent* are given to you.

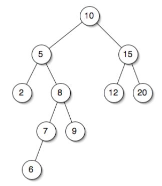
**Algorithm** *SolveProvidence(n, i, j)*
**Input:** The number *n* of disks to move from rod *i* to rod *j*. Rod *i* is assumed to hold at least *n* disks. The *n* disks moved are assumed to be the *n* smallest of the whole setup.
**Output:** Prints the sequence of moves required.
/* Write your pseudocode here */

**b) (5 points)** Let *T(n)* be the *number of moves* required to transfer *n* disks between two rods. Give a recurrence for *T(n)*. Consider only the case where *n* is odd.

**c) (5 points)** Give an explicit formula for *T(n)*. Consider only the case where *n* is odd. Simplify your formula as much as possible. Show your steps.

## **Question 5:** (12 points) Binary Search Trees and Heaps

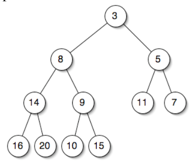
Consider the following binary search tree:



a) **(3 points)** Draw the binary search tree after an element with key 16 has been inserted.

b) **(3 points)** Draw the binary search tree after *remove(5)* has been executed. Start from the original tree, not the tree you got in (a).

Consider the following heap:



c) **(3 points)** Draw the Heap after an element with key 2 has been inserted.

d) **(3 points)** Draw the Heap after *removeMin*() has been executed. Start from the original heap, not the one you got in (c).

## Question 6: (13 points) Binary search trees

Let *T* be a binary search tree storing integers. Write the pseudocode of an algorithm (recursive or not, as you prefer) that takes as input a treeNode *x* as well as two integers *low* and *high* and prints all nodes of the subtree rooted at *x* that have a key *k* between *low* and *high* (i.e. $low \leq v \leq high$). For example, if *x* is the root of the binary search tree of question 5 and *low* = 6, *high* = 9, it should print nodes 6, 7, 8, 9, and never even consider nodes 2, 15, 12, and 20.

Your algorithm must not systematically visit the whole tree but must only visit regions of the tree where there is a chance a node with such a key may be found. The running time should not be $\Omega(high - low)$

## Question 7: (13 points) Graphs

Determining if an undirected graph can be colored with only three colors is an NP-complete problem. However, determining if it can be colored with only *two* colors is much easier. Write the pseudocode of an algorithm that determines if the vertices of a given connected undirected graph can be colored with only two colors (named 0 and 1) so that no two adjacent vertices have the same color. Assume the graph has *n* vertices numbered 0,1, 2,...*n*-1. Use the following graph ADT methods:

- *getNeighbors*(int *i*) returns the set of neighbors of vertex *i*. It is fine for you to write something like: for each vertex v in *getNeighbors*(17) do ...
- boolean *getVisited*(int *i*) returns TRUE if and only if vertex *i* has been marked as visited.
- *setVisited*(int *i*, boolean *b*) sets the visited status of vertex *i* to *b*.
- *getColor*(int *i*) returns the color of vertex *i*, either 0 or 1. If the color of vertex *i* has not been set previously, *getColor*(*i*) is undefined.
- *setColor*(int *i*, color *c*) sets the color of vertex *i* to *c*.

Page left blank intentionally. Use it if you need extra space.

Page left blank intentionally. Use it if you need extra space.