SOLUTIONS for practice final for EECS 380, 2001
Profs Markov and Brehob

Available in Postscript and PDF

Total pages: **5**
Exam duration: **1hr 50min**.
Write your name and uniqname on every sheet, including the cover.

Maximum score: 100 points + 15 extra. Extra credit points do not affect the curve.
To be eligible for extra credit, you need to earn at least *70 regular points*.

All complexity estimates are for runtime (not for memory), unless specified otherwise.

1. 30 points. **Algorithmic Complexity**

   Each line in the table corresponds to an algorithm or an algorithmic problem. Write **P** for problems and **A** for algorithms. A *problem* gives input and output, but an *algorithm* additionally entails a particular method of achieving this output. Fancy data structures (e.g., heaps, BSTs and hash-tables) often imply specific algorithms. Simple containers (e.g., arrays and linked lists) are typically used to store input or output and may *restrict* possible algorithms.

   For each algorithm, write its Theta-complexities.
   For each problem, write Theta-complexities of a **best possible algorithm** that solves the problem. There can be multiple correct answers, especially, if there is a trade-off between average-case and worst-case performance.
   No explanation necessary.

   You can assume that `operator<` and `operator==` for values stored in *containers* run in *O(1)* time. You cannot make any additional assumptions about algorithms/problems unless instructed by Prof. Brehob or Prof. Markov.

   Each line is worth **2 points**. Each wrong or missing answer on a line costs **-1 point**.
   Minimum per line **= 0 points**.

	Algorithm or Problem:	?	Best-case Theta()	Avg-case Theta()	Worst-case Theta()
1.	Find a given value in an unsorted *N-by-N* matrix.	**P**	*1*	N^2	N^2
2.	Binary search over *N* elements	**A**	*1*	*log N*	*log N*
3.	Find the largest element in an unsorted array with *N* elements	P	N	N	N
4.	Print all values appearing at least twice in a sorted stack of size *N*	P	N	N	N
5.	Insert a new element into a sorted singly-linked list with *N* elements so that the list remains sorted	P	1	N	N
6.	Given two unsorted arrays of *N* and *N/10* elements, say whether they have at least one common element	P	1	N log N	N log N
7.	Shaker sort of a doubly-linked list with *N* elements, using "early termination".	A	N	N^2	N^2
8.	Duplicate a queue of *N* elements	P	N	N	N
9.	One invocation of the `partition()` function used in the `quicksort` algorithm. Assume in-place partitioning of a complete array with *N* elements using a given pivot	P/A	N	N	N
10.	Given a pointer to an element in a singly-linked list with *N* elements, remove that element from the list	P	1	1*	N**
11.	Sort *N* 8-bit characters stored in an array.	P	N	N	N
12.	Remove the middle element from an unsorted array of *N* elements	P	1	1	1
13.	Compute *N!* for a given *N* using a straightforward recursive algorithm	A	N	N	N
14.	Find the combination of *N* decimal digits that opens a bank safe. The safe opens when you enter the right combination, and you can try as many combinations as you wish. No other feedback is available	P	1	10^N	10^N
15.	Print all diagonal values of a given *N-by-N* matrix	P	N	N	N

* Suppose the pointer points to A. Copy the successor B into A and remove old copy of B.
** The worst-case happens when A is the last element. (This can be prevented with a sentinel)

2. 10 points. **STL**

Fill in the blanks

a. "STL" stands for ____**Standard Template Library**______
b. A *range* can be defined by two __**iterators**______
c. STL's `sort()` and `binary_search()` functions take an optional _**comparison**_ function-object
d. One can use class __**map**__ from STL as an implementation of Abstract Symbol Table.
e. *Iterators* of linked list classes in STL do not allow __**random**_ access.

3. 20 points. **Fancy containers (heaps, generic trees, search trees, hash-tables, etc)**

a. **10 points**. Follow instructions from Question 1.

	Algorithm or Problem:	Best-case Theta()	Avg-case Theta()	Worst-case Theta()
1.	Print all values stored at nodes of a given tree with *N* nodes	N	N	N
2.	Convert a binary heap of *N* elements into a sorted array	don't bother	NlogN	NlogN
3.	Test whether a given array with *N* values is in a binary-heap order	1	1 or N	N
4.	One search in a BST of *N* elements. Assume that the tree is perfectly balanced and the search results in a miss	log N	log N	log N
5.	One successful look-up in a hash table with *N* elements and *load ratio*[*] *1.0*. The hash-table uses separate chaining with singly-linked lists. Assume that hash-function can be computed in *O(1)* time. Note: elements contained in the hash-table may be *poorly dispersed*.	1	1	N

[*] The *load ratio* of a hash-table with *N* elements and *M* buckets is *N/M*.

b. **5 points**. Consider `struct Key { char p1, p2, p3 };`
and the following hash-functions (modulo hash-table size).

```
1. unsigned f1(struct Key& s) { return s.p1+5*s.p2; }
2. unsigned f2(struct Key& s) { return 10*s.p1+100*s.p2+1000*s.p3; }
3. unsigned f3(struct Key& s) { return 11*s.p1+101*s.p2+1001*s.p3; }
```

Assume a hash-table of size 1250 with linear probing.
Mark each hash-function as **good** or **bad**. Use space below to explain.

Solution

- f1() does not depend on p3 therefore keys that only differ at p3 will not be dispersed. **BAD**
- All values of f2() are divisible by 10. Since the table size is also divisible by 10, at most 10% of the hash buckets can be used w/o hash collisions. **BAD**
- f3() depends on all fields and is a linear function whose coefficients are relatively prime with the table size. **GOOD**

c. **5 points**. Fill in the blanks.
Markov section only
In BSTs, **_left_** and **_right_** *rotations* have time complexity *Theta(_**1**_)*. They are explicitly used in **_root__** insertion and **_partitioning_** algorithms. Two BSTs can be *joined* using a **_recursive_** algorithm, which applies **_root_ _insertion_** to one of the trees. The worst-case complexity of such a *join* algorithm is *Theta(_*N^2*_)*, but the best case can be faster when **__a pivot exists such that all values in the first tree are smaller and all values in the second tree are larger than the pivot__.**

Brehob section only
Each node in a *2-3-4* tree has **_1__**, **_2__** or **_3__** keys in it. **_red-black_** trees are an implemention of *2-3-4* trees. Insertion into a *2-3-4* tree has worst-case complexity *Theta(_**logN**_)* and search has worst-case complexity *Theta(_**logN**_)*.

4. 20 points. **Algorithm design: Recursion / Divide and Conquer / Dynamic Programming**

Implement the following C++ function

```
void makeBalancedBST(unsigned *begin, unsigned numElem);
```

which takes an **unsorted** array and makes a balanced BST out of it, stored left to right so that children of element *k* be *2*k* and *2*k+1*. You must achieve worst-case complexity $O(numElem\ log^2(numElem))$ and explain how you did it. **15 points for the case when numElem is a power of two minus one (say, 3, 7 or 15), 5 additional points for the general case.** Use a separate page.

Solution: a a complete working program for the general case is provided.

5. 20 points. **Questions related to HWKs and Projects**

a. 5 points. Provide a dictionary produced by the **Huffman algorithm** applied to this input: AAABAABCCDCC. **No explanation necessary.**

Explanation: Frequencies: A(5), C(4), B(2) and D(1). Huffman algorithm: first merge the least frequent letters B and D (cumulative frequency is 3). Then merge the least frequent letters/subtrees: BD and C (cumulative frequency 7). Then merge the resulting subtree with A. One of possible ways to assign bits to the edges of the tree gives the following prefix-free dictionary.

Answer: A: 0, C: 10, B:110, D:111
(alternative correct answers are possible)

b. 5 points. Heapify the digits of your student ID. Start with the digits in the original order and **show the process step by step.**

Solution: for this problem one can use the linear-time make_heap algorithm or call push_heap *N* times. The latter may be easier to remember, but requires more work.

Linear-time make_heap on (1 2 3 4 5 6 7):
(1 2 3 **4** 5 6 7)   (1 2 **7** 4 5 6 **3**)   (1 **5** 7 4 **2** 6 3)
(**7** 5 **1** 4 2 6 3)   (7 5 **6** 4 2 **1** 3)

c. 10 points. You are given a function that takes *N* planar points and returns all points on the boundary of the convex hull listed clockwise. Provide an algorithm (in pseudocode or valid C++) that sorts *N* doubles using that function **and** spends *O(N)* time outside that function.

Solution:

- Find the smallest and the largest values (one linear-time pass).
- Scale all original numbers by subtracting *min* and dividing by *(max-min)* (one linear-time pass).
  // The relative order is preserved, but all numbers are now between 0 and 1.
- For every number *alpha*, compute the point on the unit circle whose polar angle is *alpha*. The exact formulae for coordinates are *x=cos(alpha), y=sin(alpha)* (one linear-time pass).
  // Note that pi=3.1415926... and pi/2>1. // Therefore, the points will not "wrap up" around the circle.
- Run the convex hull algorithm.
  // Note that that all those points will be on the convex hull.
  // Additionally, the convex hull algorithm orders the points clockwise.
- Read off the points in the clockwise order and apply inverse transformations: find *alpha* using the atan2() function or otherwise, then multiply by (max-min) and add min (one linear-time pass).

6. **Extra credit**: 15 points. **‘‘Comments not available’’**.

In this question you are given a printout of a C++ function, with coke spilled over the comments (=> you can’t read the comments). You need to explain what the function does, illustrate by several representative examples, give worst-case/best-case Theta() for runtime and substantiate these complexity estimates.

```
int L2(const char * A, const char * B)
// COMMENTS NOT AVAILABLE
{
    int m=strlen(A), n=strlen(B), i, j;
    int L[m+1][n+1]; // g++ extension to C++
    for (i = m; i >= 0; i--)
      for (j = n; j >= 0; j--)
      {
         if (A[i] == ’\0’ || B[j] == ’\0’) { L[i][j] = 0; }
         else if (A[i] == B[j]) L[i][j] = 1 + L[i+1][j+1];
         else L[i][j] = max(L[i+1][j], L[i][j+1]);
      }
    j=L[0][0];
    return j;
}
```

Source code courtesy of Prof. David Epstein.

Solution (idea only): This program computes the length of the longest common subsequence of two sequences. Its complexity is *Theta(mn)* where *m* and *n* are the lengths of the two sequences. The asymptotic runtime is the same in all cases due to the two nested loops w/o break instructions inside.