CS 411 Analysis of Algorithms, Fall 2012

Midterm Exam Solutions

The Midterm Exam was given in class on Wednesday, October 17, 2012.

A1. **Time Complexity.** In each part, indicate the (time) order of a fast algorithm to solve the given problem. **Justify your answers.** You may assume that *n* is the number of items in the given list, and that the basic operations are all the standard "C" operators.

A1a. Given a list of numbers, return the sum of the numbers.

We must read all the input, so at least linear time. We can do this with a simple loop. Thus: $\Theta(n)$.

A1b. Given a list of (possibly very large) numbers, return a list of numbers containing every number that appears in the original list exactly once, with duplicates removed. *For example, given* [1, 2, 5, 1, 5, 5, 3]*, the algorithm might return* [5, 1, 2, 3]*.*

We can do this by sorting (log-linear time) and then copying the list, while not copying any item that is equal to the previous item (linear time). Thus: $\Theta(n \log n)$.

A2. **Solving a Problem.** Indicate how you would design a software package (function, class, etc.) to solve the following problem. You may assume that you have high-quality implementations of all standard algorithms and data structures available to you.

Given a list of 100,000 numbers, which you may assume to be all distinct, print (in any order) the 1000 numbers in the list with the lowest values.

Many answers are possible. Here are some reasonable ones:

- Make a minheap out of the list, and then do heap-delete 1000 times, printing the resulting 1000 numbers.
- Use a good selection algorithm (e.g., Introselect) to find the 1000th smallest item, and then print all items less than or equal to this value.
- Sort the list (e.g., with Introsort or Merge Sort), and then print the first 1000 numbers in the resulting list.

Note that the first two options above are linear-time in the size of the whole list, while the last is log-linear-time in the size of the whole list.

A3. **Efficiency.** In each part, analyze the time efficiency of the given function. Indicate what you use for "*n*", and what your basic operation is (these should be reasonable choices). Express the efficiency using big-*O*, Ω, or Θ, whichever is most appropriate.

A3a.
```
void foo(std::vector<int> & v)
{
    assert(v.size() != 0);
    for (std::size_t i = 0; i != v.size(); ++i)
    {
        for (std::size_t k = 0; k != i; ++k)
            v[i] += v[k];
    }
}
```

n is `v.size()`.
Basic operation: `v[i] += v[k]`.
Two nested ordinary loops, so $\Theta(n^2)$.

A3b.
```
// RAIter must be a random-access iterator type
template <typename RAIter>
void bar(RAIter first, RAIter last)
{
    std::size_t size = last - first;
    if (size <= 1)
        return;

    RAIter middle = first + size/2;
    bar(first, middle);
    bar(middle, last);

    for (RAIter it = first; it != last; ++it)
        (*it) += size;
}
```

n is the number of items in the given list.
Basic operation: `(*it) += size`.
Recurrence for number of steps required: $T(n) = 2T(n/2) + n$.
Use the Master Theorem. $a = 2$, $b = 2$, $d = 1$.
So $a = b^d$. Result: $\Theta(n^d \log n) = \Theta(n \log n)$.

A4. **Gaussian Elimination.**

A4a. What is the time order of Gaussian elimination? Be sure to specify what your "*n*" means.

Two possible answers:

- For *n* equations in *n* unknowns: $\Theta(n^3)$.
- For *n* being the number of values in the input: $\Theta(n^{3/2})$.

A4b. Suppose we are doing Gaussian elimination on a system whose first two equations begin as follows:

$$2x_1 - 3x_2 + x_3 + \ldots$$
$$9x_1 + 2x_2 - 4x_3 + \ldots$$

We would not be *required* to swap the first two equations. However, we would probably do the swap, anyway. Why? How does this help us?

Gaussian elimination is usually done with *partial pivoting*. This helps to avoid loss of significance, which can lead to inaccurate results. When we do partial pivoting, we swap to get the coefficient with the greatest absolute value in the upper-left corner. Since |9| > |2|, we would swap the first two equations here.

A5. **Quickselect.**

A5a. What problem does the Quickselect algorithm solve? (Describe the input and output; do not merely name the problem.)

Quickselect solves the *selection problem*: Given a list of orderable items and a number *k*, return the *k*th smallest value. Equivalently, return item *k* in the list as it would be if it were sorted.

A5b. Quickselect has very poor worst-case performance. What can we do about this?

Quickselect has linear-time average-case performance, but quadratic-time worst-case performance. The standard solution is to keep track of the recursion depth, and switch to a linear-time worst-case selection algorithm if this depth exceeds some limit. (The usual algorithm to switch to is Blum-Floyd-Pratt-Rivest-Tarjan selection, and the resulting algorithm is *Introselect*.)

B1. **Solving Recurrences.** In each part, solve the given recurrence exactly.

B1a. $x(n)=x(n-1)+4$ for $n\geq 2$. $x(1)=7$.

This is a first-order linear recurrence. General solution: $x(n)=4n+k$.

Plugging in the initial condition $x(1)=7$, we get $4+k=7$, and so $k=3$.

Solution: $x(n)=4n+3$.

B1b. $x(n)=x(\lfloor n/3\rfloor)+2$ for $n\geq 3$. $x(1)=0$. $x(2)=0$.
Note: $\lfloor k\rfloor$ is the "floor" of k: the greatest integer less than or equal to k.

Substituting, we see that $x(1)=0$, $x(3)=2$, $x(9)=4$, $x(27)=6$. The solution should be something like $x(n)=2\log_3 n$.

Substituting some more: $x(1)=0$, $x(2)=0$, $x(3)=2$, $x(4)=2$. $x(5)=2$, $x(6)=2$. $x(7)=2$, $x(8)=2$, $x(9)=4$. I will guess that $x(n)=2\lfloor\log_3 n\rfloor$.

This is easily checked for $n=1,2$. For $n\geq 3$, we need to show that the guess makes the equation $x(n)=x(\lfloor n/3\rfloor)+2$ true.

$x(n)=2\lfloor\log_3 n\rfloor=2\lfloor\log_3\lfloor n/3\rfloor\rfloor+2=x(\lfloor n/3\rfloor)+2$, and so the guess is correct.

Solution: $x(n)=2\lfloor\log_3 n\rfloor$.

B1c. $x(n)-4x(n-1)+4x(n-2)=0$ for $n\geq 2$. $x(0)=3$. $x(1)=3$.

This is a second-order linear recurrence. The characteristic polynomial is x^2-4x+4, which has just one root: $x=2$. Using the appropriate case, we obtain the general solution: $x(n)=a_1 2^n+a_2 n 2^n$.

Plugging in the first initial condition $x(0)=3$, we get $a_1=3$.

Then plugging in the second initial condition $x(1)=3$, we get $a_2=-\frac{3}{2}$.

Solution: $x(n)=3\cdot 2^n-\frac{3}{2}\cdot n 2^n$.

B2. **Heap Sort.** Heap Sort requires Θ(*n* log *n*) time and uses only constant additional space. No other sorting algorithm we have covered has these properties (and neither, I believe, do any of the algorithms covered in CS 311). Why, then, is Heap Sort *not* the primary algorithm used for most sorting? **Give two different reasons.**

The two primary reasons are that Heap Sort is slower than other algorithms, and it is not stable. Another reason is that Heap Sort requires random-access data, and so is not a good choice for sorting (say) a Linked List.

B3. **Red-Black Trees.**

B3a. What is the order of the CRUD operations for a Red-Black Tree?

All four of the CRUD operations are logarithmic-time for a Red-Black Tree.

B3b. Despite their superior performance (see part a), Red-Black Trees are used less and less for in-memory associative data. Why is this?

Hash Tables are the usual choice for in-memory associative data, since they offer much better average-case performance than balanced search trees.

B3c. A balanced search tree of some sort is generally considered to be a good implementation for an external associative data (e.g., a large key-value dataset kept on external storage). However, Red-Black Trees are rarely used for this, either. Why not?

External data is often *block-access*: reading a single byte gets you the rest of the block for no additional cost. B-Trees, and the various variants of these, are balanced search trees whose performance can be optimized for block-access data. This is done by setting the size of a node to the size of a block (or two blocks, etc.). Red-Black Trees, with just one item in each node, cannot be optimized in this way.