COS 226	Algorithms and Data Structures	Spring 2004

Midterm Solutions

1. **9 sorting algorithms.**

   0 4 2 5 7 1 3 6 8 9

   Similar to Spring 2003 exam.

   (a) Original input

   (b) Insertion: the algorithm has sorted the first 12 strings, but hasn't touched the remaining 23 strings.

   (c) Bubble: after 12 phases of bubblesort, the smallest 12 strings are guaranteed to be in their final sorted order (and actually 15 are). `chug` was bubbled down so it's not selection sort.

   (d) LSD: the strings are sorted on their last character.

   (e) MSD. The strings are sorted on their first character.

   (f) 3-way radix quicksort: after 3-way partitioning on the `c` in `chug`, all smaller keys are in the top piece, all larger keys are in the bottom piece, and all keys that begin with `c` are in the middle piece.

   (g) Heapsort: the first phase of heapsort puts the keys in reverse order in the heap.

   (h) Mergesort: the algorithm has sorted the first 18 strings and the last 17 strings. One final merge will put the strings in sorted order.

   (i) Quicksort: after partitioning on `chug`, all smaller keys are in the top piece, all smaller keys are in the bottom piece.

   (j) Selection: the smallest 12 strings are in their final sorted order. `chug` didn't move so it's not bubble sort.

2. **Analysis of algorithms.**

   (a) N. There are n string concatenations, where the pairs of strings to be concatenated have lengths $1, 2, 4, \ldots, N/2$. Thus, the overall running time is proportional to $2 + 4 + 8 + ... + N = 2N - 2$.

   (b) N^2. It performs N string concatenations where the sums of the lengths of the strings to be concatenated are $1, 2, 3, \ldots, N$. The overall running time is proportional to $1 + 2 + \ldots + N = N(N + 1)/2$.

   (c) $N \log N$. Each function call makes two recursive calls on inputs half the size, and combines them with a linear amount of work. You should recognize the running time as the solution to the classic divide-and-conquer recurrence, just like mergesort.

3. **Hashing.**

   `N O P B R I G`

4. **Markov model.**

(a) Very similar to the Markov data type on the language modeling assignment.

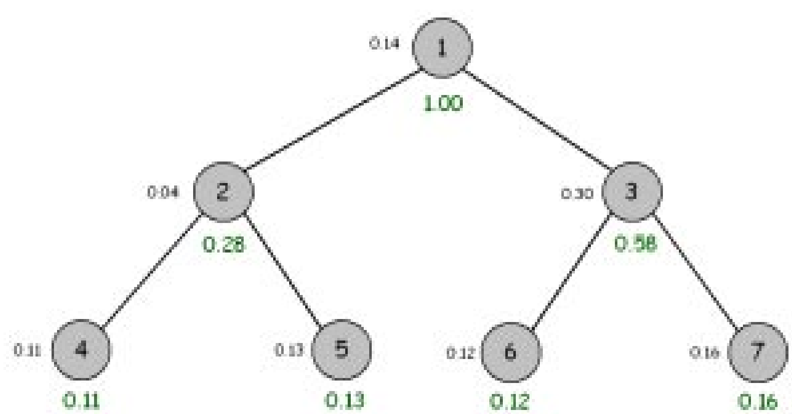
```
double r = Math.random();
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum = sum + p[i];
    if (r < sum) return i;
}
```

(b) Form the cumulative probabilities (ala key-indexed sorting), generate a random number between 0 and 1, and binary search for the interval containing it.

	1	2	3	4	5	6	7
p[i]	0.14	0.04	0.30	0.11	0.13	0.12	0.16
sum[i]	0.14	0.18	0.48	0.59	0.72	0.84	1.00

Preprocessing takes $O(N)$ time to compute the cumulative sums. Binary search takes $O(\log N)$ time per query.

(c) This is tricky one. The key is to design a data structure that makes modifying cumulative sums easy. This is essentially what we did to find the ith largest element in a BST. Recall, to do that, we maintained an extra integer variable in node x to count the total number of nodes in the tree rooted at x. In the Makov model application, we maintain an extra real variable in node x to counts the total sum of probabilities in the tree rooted at x. The picture below illustrates the tree (although it's not in BST order, but the ordering isn't crucial).



To generate a sample, pick a number between 0 and 1 at random. In the example above, if the number is between 0.00 and 0.28, go left; if it's between 0.28 and 0.28 + 0.58 = 0.86, go right; otherwise return the site in the current node. If you go left, then the number is between 0.00 and 0.28. If it's between 0.00 and 0.11, go left again; if it's between 0.11 and 0.11 + 0.13 = 0.24 go right; otherwise return the site in the current node. To change the probability of a site i, follow the path from the site up to the root, adding the net change to each node. All operations take $O(\log N)$ time if the tree is balanced.

Note that we don't actually need the BST ordering since we could maintain an array of size N that maps from integers to tree nodes. Moreover, we could make the tree complete, and walk the tree heap-style instead of with explicit pointers.

5. **Desecrated quicksort.**

   (a) **Correctness.** No. The correctness of a sorting algorithm means that the resulting array is in ascending order, and it contains exactly the same elements that you began with. Neither property is satisfied here because of a catastrophic off-by-one error that causes `L[lo-1]` and `H[hi-1]` to be lost.

   (b) **Running time.** $\Theta(N \log N)$ average case, $\Theta(N^2)$ worst case The asymptotic running time is the the same as quicksort, but we do substantially more work (allocating extra arrays and copying back and forth) per function call so the constants are proportionally higher. As with classic quicksort, if the input is in ascending order the running time goes quadratic. Unlike Sedgewick's code, this sort also goes quadratic if there are lots of duplicates.

   (c) **Memory usage.** $\Theta(N)$ average case, $\Theta(N^2)$ worst case A total disaster. One of the virtues of quicksort is that it is in-place. The key thing to notice is that this algorithm is not in-place since it uses several extra auxilliary arrays. Much like the mythical unicorn, a sorting algorithm that uses quadratic memory is seldom witnessed. Alarmingly, this version of quicksort can consume a quadratic amount of memory!

   Now, we do a more detailed analysis (that was not required for full credit). The first invocation of the function allocates two extra arrays of size N. This is a total of $3N$, which is already atrocious, and worse than mergesort. But things get worse since the function is called recursively, and new memory is allocated upon each invocation (but freed and garbage-collected upon returning). If the input is in descending order,[1] the depth of the function call stack will be N, with arrays of size $1, 2, \ldots, N$. This leaves a total of $3(1 + 2 + \ldots N) \approx 5N^2/2$ memory in use at one time!

   The average space usage is more difficult to analyze since memory is freed and garbage-collected upon returning from a function. Also note that the array `H[]` is not shrunk until after the recursive call on the low piece. If all of the partitions split the file exactly in half, then memory usage is $5N$. Empirically, memory usage appears to grow more like $10N$, in large part, because the partitions do not split the files perfectly in half.

   (d) **Stability.** No. It's not stable, even if you fix the off-by-one error to make it sort correct. We don't ordinarily think of making quicksort stable because of all the swapping with the in-place partitioning phase. However, with extra memory, it's easy to make quicksort stable because you aren't swapping elements around. The only thing you can screw up is dealing with keys equal to the partitioning element. As you might expect by now, the author blows the opportunity to make quicksort stable (even if by accident)! Here's a simple change to the partitioning step that would make it stable.

```
for (int i = 1; i < N; i++) {
   if (a[i] < a[0]) L[lo++] = a[i];
   if (a[i] > a[0]) H[hi++] = a[i];
}
```

[1] It is amusing to note that the best case memory usage occurs when the input is in ascending order since the depth of the function call stack is now 2. Sadly, this is the same input that makes the running time go quadratic!

6. **Red-black trees.**

Inserting T is easy - just change the color. Inserting Q requires two rotations, and makes I the new root. As a check, observe that the keys are in BST order, and every path from the root to a leaf has the same number of blank links (in this case 1).