COS 226 **Algorithms and Data Structures** **Spring 2004**

Final Solutions

1. **Analysis of algorithms.**

   Answers may vary.

   (a) Amortized: an amortized analysis provides a worst-case guarantee for a sequence of N operations. Any individual operation can be slow, but there is a performance guarantee for the sequence. For example, when inserting elements into an array, a common strategy is to double the size of the array when needed. This doubling operation is very expensive. But it can only happen after a long sequence of cheap insertions.

   Array/stack/binary heap/hash table with repeated doubling, splay tree, union-find, Fibonacci heap.

   (b) Worst-case: a worst-case analysis provides a guarantee on the running time (in terms of the size of the problem) for *any* possible input. This was the most common style of analysis used in the course.

   Mergesort, heapsort, red-black tree, Knuth-Morris-Pratt, Graham scan, rectangle intersection, Prim, Kruskal, Dijkstra, Bellman-Ford, Ford-Fulkerson with shortest augmenting path heuristic.

   (c) Average case: an average case analysis provides a performance expectation assuming the input comes from a (specific) random input distribution.

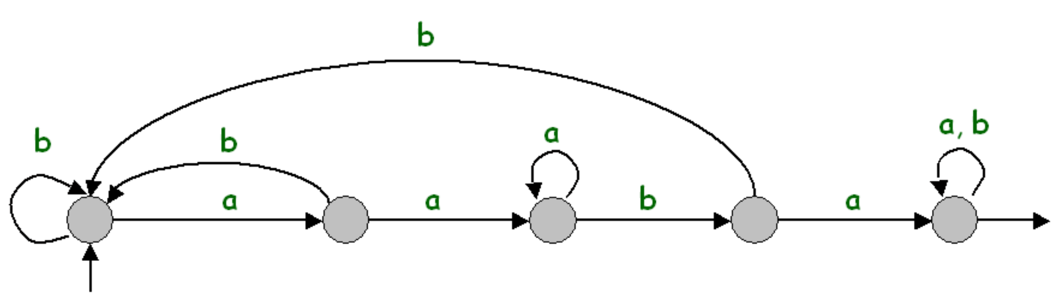
   Hashing where keys are uniformly distributed, quicksort where input is a random permutation, BST where input is a random permutation, brute force string search assuming inputs are random bitstring, quick elimination assuming random points in plane.

   (d) Randomized: a randomized analysis provides a performance expectation for any possible input. The randomness occurs from the algorithm itself, not from any assumption on the input.

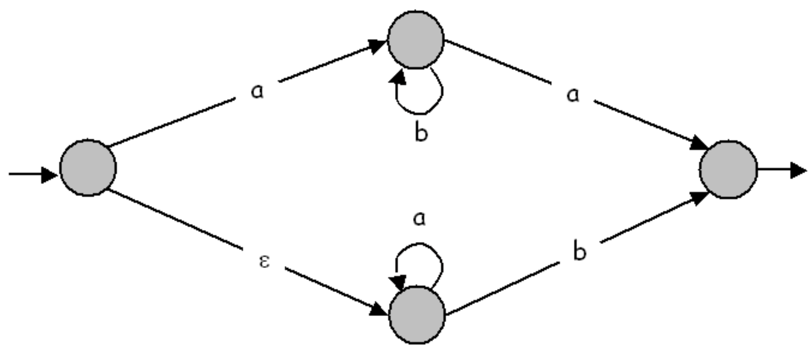
   Quicksort with random partition element, randomized BST, Karp-Rabin with random hash function.

2. **String searching and pattern matching.**

   (a)



(b)



3. **Convex hull.**

```
A
AB
ABC
ABCD
ABCDE
ABF
ABFG
ABFH
ABFI
```

4. **Discretized Voronoi diagram.**

   (a) $create(R)$. Use a helper data type `Pixel` to manipulate pixels. Initialize an R-by-R grid of pixels to `null` so that `nearest[i][j]` is the inserted pixel closest to (i, j).

   (b) $find(i, j)$. Return `nearest[i][j]`.

   (c) $insert(x, y)$. Create a new `Pixel` object `p` with coordinates (x, y). For each i and j, check whether (i, j) is closer to `p` than it is to `nearest[i][j]`. If it is, update `nearest[i][j]`.

5. **Undirected graphs.**

   (a) The preorder traversal order is: `ABCDEGHF`. The preorder numbers are `01234756`.

   (b) The postorder traveral order is: `DFHGECBA`. The postorder numbers are `76504132`.

6. **Minimum spanning tree.**

   (a) The key observation is that to compute an MST you only need to be able to compare edge weights. The smallest value is e^{-24} and the biggest is e^{-9}. If we run Kruskal's algorithm, we discover the edges in the following order

```
B-H  F-H  C-F  A-F G-H  E-F  D-E
```

(b) If $-\infty \leq x \leq e^{-17}$ we obtain an MST by including B-D and deleting D-E.

7. **Max flow, min cut.**

(a) There are two possible shortest augmenting paths: s-3-2-5-t and s-3-2-6-t.

(b) The residual capacity of the shortest augmenting path is 1 in both cases. The original flow has value 25, so the resulting flow has value 26.

(c) The flow is not optimal. In both cases the augmenting path s-4-3-2-5-t remains.

8. **Data compression.**

(a) `5 rdrcbaaaaba`

(b) `carabadabra`

9. **Linear programming.**

$$\begin{array}{ll}
\text{maximize} & 320(F_1+C_1+B_1)+400(F_2+C_2+B_2)+360(F_3+C_3+B_3)+290(F_4+C_4+B_4) \\
\text{subject to:} & F_1+F_2+F_3+F_4 \leq 12 \\
& C_1+C_2+C_3+C_4 \leq 18 \\
& B_1+B_2+B_3+B_4 \leq 10 \\
& F_1+F_2+F_3+F_4 = C_1+C_2+C_3+C_4 \\
& F_1+F_2+F_3+F_4 = B_1+B_2+B_3+B_4 \\
& F_1+C_1+B_1 \leq 20 \\
& F_2+C_2+B_2 \leq 16 \\
& F_3+C_3+B_3 \leq 25 \\
& F_4+C_4+B_4 \leq 23 \\
& 500F_1+700F_2+600F_3+400F_4 \leq 7000 \\
& 500C_1+700C_2+600C_3+400C_4 \leq 9000 \\
& 500B_1+700B_2+600B_3+400B_4 \leq 5000 \\
& F_1,F_2,F_3,F_4,C_1,C_2,C_3,C_4,B_1,B_2,B_3,B_4 \geq 0
\end{array}$$

We note that the first two constraints are redundant since they are implied by the next 3 constraints.

10. **Reductions.**

Create a new directed graph G' with the same set of vertices. For each undirected edge v-w in G, add two directed edges to G': one edge from v to w with distance $c(v)$, and one from w to v with distance $c(w)$. The shortest path from s to t in G' is the path in G that minimizes the sum of the vertex weights. To see why, observe that we pay the price $c(v)$ whenever we leave vertex v. Since we leave each vertex on the path once, this will sum to the right value. Since all the edge weights are nonnegative, there is no incentive to revisit a vertex so we will use each original edge in at most once (either in the forward or reverse direction).