COS 226	Algorithms and Data Structures	Fall 2006

Final Solutions

1. **Analysis of algorithms.**

   (a) There exists a constant $c > 0$ such that for any array of N elements, heapsort takes at most $cN \lg N$ steps (pairwise comparisons and exchanges).

   (b) Any comparison-based sorting algorithm must make $\Omega(N \log N)$ comparisons in the worst case.

   (c) 2 hours.

   (d) 1 hour.

2. **Algorithm analogies.**

   (a) Hamilton path

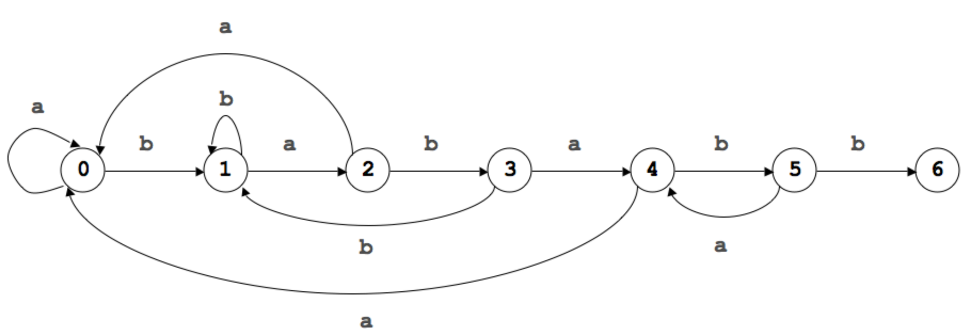
   (b) ternary search trie

   (c) Dijkstra's algorithm

   (d) ccw

   (e) binary heap

3. **String searching.**



4. **Convex hull.**

   (a) List the points in the order that they are considered for insertion into the convex hull.

```
J  G  H  I  E  F  D  A  B  C

1.  J -> G -> H
2.  J -> G -> H -> I
3.  J -> G -> E
4.  J -> G -> E -> F
5.  J -> G -> E -> D
6.  J -> G -> E -> A
7.  J -> G -> E -> A -> B
8.  J -> G -> E -> A -> B -> C
```

   (b) A set of points is *convex* if for any two points p_1 and p_2 in the set, all of the points on the line segment from p_1 to p_2 are also in the set.

5. **BFS and DFS.**

   (a) DFS preorder: A B D E C F H G I

   (b) DFS postorder: B H F C I G E D A

   (c) BFS levelorder: A B D E I C F G H

6. **Algorithm throwdown.**

Red-black tree	Ternary search trie
arbitrary `Comparable` keys	faster for string keys
worst-case guarantee	longest prefix match

Dijkstra's algorithm	Bellman-Ford-Moore
faster	handles negative weights
undirected graphs	negative cycle detection

Burrows-Wheeler	LZW compression
better compression ratio	faster

Red-black tree	Hash table
performance guarantee	$O(1)$ average case
range search	

Breadth-first search	Depth-first search
shortest path	topological sort
	strongly connected components

7. **Minimum spanning tree.**

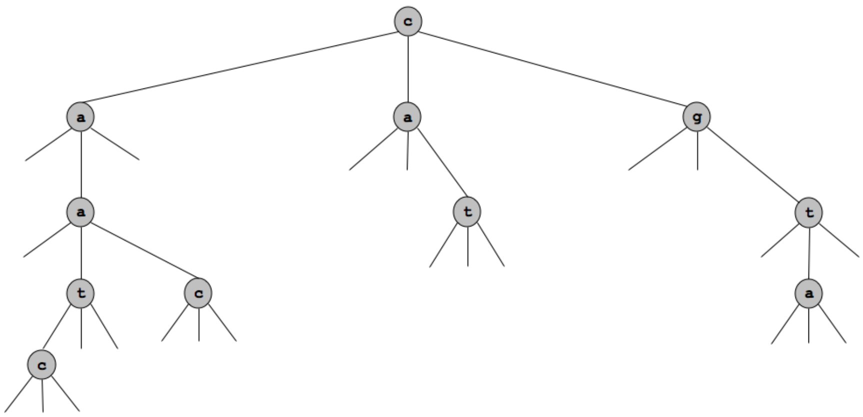
(a) `C-D B-C A-D E-F G-I E-G F-H D-I`

(b) `A-D C-D B-C D-I G-I E-G E-F F-H`

8. **Data compression and tries.**

(a) `c a g t aa ac ca aat ta aac ct`

(b)



9. **Linear programming.**

$$
\begin{array}{lrcrcrcrcrcrcrcl}
\text{maximize} & -26A & - & 30B & - & 20C & & & & & & & & & & \\
\text{subject to:} & A & + & B & + & 2C & & & & & & & & & = & 200 \\
 & 3A & + & 6B & + & 3C & + & S_1 & & & & & & & = & 45 \\
 & 9A & + & 2B & + & 4C & & & - & S_2 & & & & & = & 85 \\
 & 5A & + & 9B & + & 6C & & & & & + & S_3 & & & = & 95 \\
 & -5A & + & -9B & + & -6C & & & & & & & + & S_4 & = & 95 \\
 & A & , & B & , & C & , & S_1 & , & S_2 & , & S_3 & , & S_4 & \geq & 0
\end{array}
$$

10. **Reductions.**

Given an instance $x_1, \ldots x_N$ of ElementDistinctness, form the instance $(x_1, 0), \ldots, (x_N, 0)$ for ClosestPair. The elements in the ElementDistinctness problem are distinct if and only if the closest pair of points has distance strictly greater than 0.

Remark. There is an $\Omega(N \log N)$ lower bound for ElementDistinctness in the quadratic decision tree model of computation. This reduction proves that there is also an $\Omega(N \log N)$ lower bound for ClosestPair.

11. **Sorting and hashing.**

(a) Sort the N elements. Then, scan through the elements and check if any two adjacent elements are equal. Use heapsort to guarantee $O(N \log N)$ performance, while using $O(1)$ extra memory.

Note that quicksort does not guaranteed $O(N \log N)$ performance. Also, it uses $\Omega(\log N)$ extra space for the function call stack.

(b) Create an empty set of elements. For each element of the N elements, check if it's already in the set. If it is, you've found a duplicate; otherwise insert it into the set. Use a hash table to obtain $O(1)$ average time per operation.

12. **Shortest path with landmark.**

(a) Compute the shortest path from v to x using Dijkstra's algorithm. Then compute the shortest path from x to w using Dijkstra's algorithm. Concatenate the two paths.

Correctness follows since all of the edge weights are positive: if the shortest landmark path used a non-shortest path from v to x, we could shorten it by substituting a shortest path from v to x. The same argument applies to the path from x to w.

(b) Pre-compute the following two quantities. Here x is fixed, and we compute the quantity for every vertex u.

- $\bar{d}(u, x)$ = length of the shortest path from u to x.
- $d(x, u)$ = length shortest path from x to u.

Use Dijkstra's algorithm (with x as the source) to compute $d(x, u)$. This computes $d(x, u)$ for every vertex u in $O(E \log V)$ time. Use Dijkstra's algorithm on the reverse graph $\bar{G}$ (with x as the source) to compute $\bar{d}(u, x)$.

To process a shortest landmark path query from v to w, return $\bar{d}(v, x) + d(x, w)$.