Final Exam Sample Solution

CS410, Summer 1998

10 August 1998

Please note:

- This exam is closed book, closed note. Sit every-other seat.
- Put your answers in this exam.
- The order of the questions roughly follows the course presentation and not necessarily order of difficulty.
- There are 8 questions totalling 100 points and 1 extra credit question. Point values are indicated for each question. In particular, the first 4 questions are shorter and worth fewer points.
- Feel free to separate pages; you can re-staple them at the end of the exam.
- You have 120 minutes to complete the exam. Do not continue writing after time is up.
- Partial credit is available for all questions. However, keep your answers concise – extraneous ramblings will take your time and may hurt you.
- Answers will be posted on the web.

1. [**7 points**] When using a hashtable, we transform a key to an `int` and then hash the `int` to a bucket. In the table we store the key and its corresponding value. We might also choose to use extra space to store the `int`.

   (a) Explain how storing this extra information might speed up lookup operations.

   (b) Explain how storing this extra information might speed up table resizing.

   (c) Which of your arguments above is irrelevant if table collisions never occur?

   (a) Comparing `int`s is presumably faster than comparing keys. We can compare the transformed key we are looking up against the transformed keys stored in the table and only compare the keys if the `int`s match.

   (b) When resizing, we must re-hash every item in the table. By storing the transformed keys, we do not need to re-transform the keys.

   (c) The first argument is irrelevant. We will never have distinct items in the same bucket, so we will never avoid a key comparison by using an `int` comparsion.

2. [**7 points**] Consider double hashing.

   (a) What is the advantage of double hashing over the other methods of open address hashing we learned?

   (b) Explain why this advantage is lost if the two auxiliary hash functions used in double hashing are the same.

   (a) It uses m^2 of the $m!$ possible permutations of the m buckets as opposed to only m used by linear or quadratic probing. In the other methods, once two items collide, they will continue to collide as they search for an open bucket.

   (b) If the two hash functions are the same, items that initially collide will always collide. Hence we cannot do better than with quadratic probing.

3. [**7 points**] Suppose we are inserting elements into an array. If the array becomes full, we make a new array twice as large, copy all of the elements into the new array, and continue. The total time to complete n insertions is $O(n)$. Explain why the argument below is **not** a correct amortized analysis establishing this bound. Note: You are *not* being asked to give a correct analysis.

   On insertion put one "chip" on the element inserted. The cost of insertion is still $O(1)$. *Now when resizing needs to occur, use the chips on the elements to pay for the copying. Then the amortized cost of copying is free. Since each insertion is* $O(1)$ *and copying is paid for in advance, the total time for* n *insertions is* $O(n)$.

   After an item is copied it no longer has a chip on it. So when we go to do a second resizing, as few as half of the items could have chips on them. Hence copy is not free according to this accounting scheme.

4. [**7 points**]

Suppose we need a data structure with the following operations:

- `Initialize(`n`)`: Put n objects called $1, 2, \ldots, n$ in n separate sets.
- `Find(`x`)`: Get the set containing x.
- `Union(`x`, `y`)`: Merge the sets containing x and y.
- `UnUnion()`: Undo the effect of the most recent `Union` operation. It is an error to perform two `UnUnion` operations without a `Union` in-between (or to perform an `UnUnion` before any `Union`). There may be `Find` operations between a `Union` and `UnUnion`, however.

We studied an implementation of union-find that could perform n unions and m finds in total time $O(m \log^* n)$. (Remember than any finds used during union were counted in the m.)

(a) Explain why the $O(m \log^* n)$ implementation cannot efficiently support the `UnUnion` operation. Be specific.

(b) Briefly describe an implementation that could perform n unions or ununions and m finds in total time $O(m \log n)$.

(a) Path compression. After path compression, the tree inserted by the union may not have the same elements. Hence removing the correct elements from the larger tree cannot be done efficiently.

(b) Use union-by-rank but not path compression.
(Full credit was also given for schemes that kept a copy of the old trees, although this was not the intended solution.)

5. In mergesort, for problems of size larger than 1, we divide the problem into two smaller problems, solve the smaller problems, and use them to help solve the larger problem.

   (a) [**10 points**] Explain how you could write a version of mergesort that divided the problem into k smaller problems. The running time of your program should be $O(kn \log_k n)$. Pseudo-code will probably be helpful. You do not need to focus on low-level details, but be sure to discuss all parts of the algorithm.

   (b) [**4 points**] Briefly argue that your program runs in time $O(kn \log_k n)$.

   (c) [**4 points**] Explain briefly how you could improve your program to run in time $O((\log_2 k)n \log_k n)$.

   (a) Divide the array into k equal-size ranges and recursively sort each of them. Then merge the k sorted arrays by keeping k "fingers" – one in each array. Have the finger point to the smallest element in each array that has not been copied to the answer array. While not done, find the smallest of the k values, copy it, and advance that finger.

   (Another unintended solution that works is to merge the first two subproblems, then merge the third with the result of the first two, then merge the fourth with the result of the first three, and so on. This is less straightforward and the analysis is trickier, but it does work.)

   (b) It is easiest to see with a recursion tree. The height of the tree is $O(\log_k n)$. The work at each level is for the merges. Each element is part of one merge at each level. Each step of a merge takes time $O(k)$. So each level of the tree does total work $O(kn)$. So the total time is $O(kn \log_k n)$.

   (The analysis for the unintended solution also has a tree of depth $O(\log_k n)$ and a merge for a problem of size n takes $O(kn)$. Showing this $O(kn)$ part is more difficult, however. The total time for the k two-array merges is $\sum_{i=0}^{k}(in/k + n/k) = k(k+1)n/2k + n/k = O(kn)$.)

   (c) The intended solution was a homework problem: Organize the k subarrays in a heap, with key the minimum uncopied element of the array. To copy an element, remove the first element of the root of the heap, advance the finger, and heapify based on the new key. (When an array is exhausted, delete it from the heap.) Hence one step of the merge is $O(\log_2 k)$ so an entire merge of n elements is $O(n \log_2 k)$.

   (An unintended solution which is arguably better is to do the merge recursively: merge pairs of arrays, then merge pairs of the results, and so on. So each element is part of $\log_2 k$ 2-way merges.)

6. Consider a version of quicksort in which the pivot element is **not** chosen at random. Here is the implementation of `partition` given in lecture; assume quicksort uses it (and not some other implementation of partition):

```
int partition (array, l, r) {
  int p = pickPivot(l, r);
  swap(array, l, p);
  int i = l+1;
  int j = r;
  while (true) {
        while (i <= r && a[i] < a[l])
           i++;
        while (a[j] > a[l])
           j--;
        if (j <= i)
          break;
        swap(array, i, j);
  }
  swap(array, l, j);
  return j;
}
```

(a) [**4 points**] What is the asymptotic running time of quicksort if `pickPivot` returns `(l+r)/2` and the *original array of* n *elements is already sorted*?

(b) For the rest of this problem, assume `pickPivot` returns `l`:

i. [**4 points**] What is the asymptotic running time of quicksort if the *original array of* n *elements is already sorted*?

ii. [**5 points**] Consider quicksort applied to the array [6, 5, 4, 3, 2, 1]. Show the array contents at the beginning of each recursive call to quicksort.

iii. [**5 points**] What is the asymptotic running time of quicksort if the *original array of* n *elements is already* ***reverse-sorted***?

(a) $O(n \log n)$

(b) i. $O(n^2)$

ii.
```
6 5 4 3 2 1
1 5 4 3 2 6
1 5 4 3 2 6
1 2 4 3 5 6
1 2 4 3 5 6
1 2 3 4 5 6
```

iii. $O(n^2)$

7. [**18 points**] Suppose the degree requirements for the computer science major are organized as a dag where vertices are required courses and an edge (x, y) means course x must be completed prior to beginning course y. Make the following assumptions:

- All prerequisites must be obeyed.
- Every course is offered every semester.
- There is no limit to the number of courses you can take in one semester.

Sketch an efficient algorithm to compute the minimum number of semesters required to complete the degree.

Keep a "semester number" on each vertex; make the number 0 for each vertex initially. Find the vertices with in-degree zero (either directly or with topological sort). Make a queue and enqueue these vertices. While the queue is not empty:

- dequeue a vertex, call it t
- for each edge (t, v) set v's semester number to the max of its current value and t's semesters number + 1.
- if *every* other edge into v has been visited, enqueue v.

Return the the maximum semester number over all vertices.

Running time is $O(n + m)$ where G has n vertices and m edges.

(Substantial partial credit was given for doing a separate search from each in-degree zero vertex and returning the maximum depth of the resulting trees. This is a $O(nm)$ solution in the worst case, however.)

8. Assume we have a weighted, undirected graph G and we have *already computed* a minimum spanning tree T for G. Assume T is stored in an adjacency-list representation (separate from G). Assume G has n vertices.

   (a) [**5 points**] Argue that for any two vertices u and v there is exactly one simple path from u to v in T.

   (b) [**5 points**] Explain how to find the path from u to v in T in time $O(n)$.

   (c) [**5 points**] Suppose we now (after having computed T) add an edge (u, v) with weight w to G. Argue that T is no longer a minimum spanning tree if w is less than the weight of some edge on the path from u to v in T.

   (d) [**3 points**] Assume the following fact: If w is *not* less than the weight of some edge on the path from u to v in T, then T *is* still a minimum spanning tree.

   Explain how given T and a new edge (u, v) of weight w we could compute a minimum spanning tree for G plus (u, v) in time $O(n)$.

(a) Since T spans the graph, there is a path in T between any two vertices. Hence there is a simple path. Hence there is at least one. Now assume for contradiction there is more than one. Since the two paths start at u and end at v, they must "separate" at some vertex w_1 (possibly u) and "rejoin" at some vertex w_2 (possibly v). Then T has a cycle involving w_1 and w_2. But T is a tree. Contradiciton.

(b) Do DFS or BFS from u, and return the path to v. Since T has $n - 1$ edges, the running time is $O(n)$.

(c) Take T, remove an edge of the u to v path with weight greater than w and add the new edge. Claim: The resulting graph still spans G. Proof: Suppose the removed edge was (x, y). Then any path in T that used (x, y) can now use the path x to u, (u, v), v to y in its place. Claim: The resulting graph has lower total weight. Proof: Immediate. Hence by definition of MST, T is no longer a minimum spanning tree.

(d) Find the u to v path in T as described above. Get the edge with largest weight. If this weight is less than w, return T. Else return the tree that is T without this heaviest edge and with (u, v).

Extra Credit **[5 points]**

Let edge (u, v) be the heaviest edge on a cycle in a weighted graph G. Let $\mathcal{P}$ be any minimum spanning tree algorithm that:

- starts with an empty set A
- iteratively adds to A a light edge of a cut that A respects
- returns A

Prove the following claim: $\mathcal{P}$ will never add (u, v).

Let the cycle be $v, u = w_0, w_1, w_2, \ldots, w_n, w_{n+1} = v$. Since any cut must partition all vertices, for any cut that (u, v) crosses, there is some i (possibly 0 or n) such that (w_i, w_{i+1}) also crosses the cut. If (w_i, w_{i+1}) is not in A, then it is a lighter edge and will be added before (u, v). If (w_i, w_{i+1}) is in A, then A cannot respect the cut so (u, v) cannot be added.