Final Exam

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?

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.

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 $O(n)$.

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 $O(m \log n)$.

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 $O((\log_2 k)n \log_k n)$.

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***?

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.

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 $O(n)$.

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).