```
         CS 61B:  Selected Solutions (Practice for the Final Exam)


[1]  [a]  insertion sort

          71808294
          17808294
          01788294
          01278894
          01247889

     [b]  selection sort

          71808294
          01878294
          01278894
          01248897
          01247898
          01247889

     [c]  mergesort

          71808294
          17,80,8294
          17,08,8294
          17,08,28,94
          17,08,28,49
          0178,28,49
          0178,2489
          01247889

     [d]  quicksort

          71808294
          21808794
          21088794
          210|4|8798
          0|12|4|8798
          012|4|7898
          012|4|7|8|98
          01247889

     [e]  heapsort

          71808294 \
          81807294  |
          82807194  | bottomUpHeap()
          82897104  |
          82897140 /
          08897241
          01897842
          01298874
          01249887
          01247898
          01247898
          01247889
```

```
[2]  Extend each item so that it has a "secondary key," which is the index of
     the item in the initial array/list.  If two items have the same primary
     key, the tie is broken using the secondary key, so no two items are ever
     considered equal.

          5 8 7 5 8 8 3 7     =>     5/0 8/1 7/2 5/3 8/4 8/5 3/6 7/7

[3]  [a]  DFS:  abdcegfhi
          BFS:  abcdegifh
     [b]  DFS:  efihgcdba
     [c]  gh, ac, hi, ab, cd, cg, fg, ce    (cg may come before cd instead)

[4]  [a]  [i]  O(|V|^2)    [ii]  O(|V|)
     [b]  Visited in preorder.  Finished in postorder.
     [c]  With BFS, it's done exactly the same as with DFS.
          (See Homework 9 for a description of how it's done with the latter.)
     [d]  for (each vertex v in the graph) {
            if (v has not been visited) {
              increment the count of connected components
              perform DFS on v, thereby marking all vertices in its
                                     connected component as having been visited
            }
          }

          This algorithm requires that you don't "unmark" the marked vertices
          between calls to DFS.

[5]  [a]          3                   12
                 / \                 /
                1   12              3
                /\  /              / \
              0  2 11             1   11
                  /              /\   /
          =>     5         =>   0  2 5
                / \                 / \
               4   7               4   7
                  / \                 / \
                 6   9               6   9
                    / \                 / \
                   8   10              8   10

     [b]        4.5
               /   \
              3     5
             / \     \
            1   4     11
           / \       / \
          0   2     7   12
                   / \
                  6   9
                     / \
                    8   10
```

```
     [c]          3                 3                   10
                 / \               / \                 /  \
                1   5             1   10              3    11
               /\   /\           /\   /\             / \     \
              0  2 4  11        0  2 5  11          1   5     12
                     / \            / \  \         /\   /\
          =>       10   12  =>     4   9  12  =>  0  2 4  9
                   /                  /                  /
                  9                  7                  7
                 /                  / \                / \
                7                  6   8              6   8
               / \
              6   8

     [d]          3                 3                       10
                 / \               / \                     /  \
                1   5             1   5                   5    11
               /\   /\           /\   /\                 / \    \
              0  2 4  11        0  2 4  10              3   7    12
          =>         / \    =>         / \      =>     / \ / \
                    7   12            7   11          1  4 6  8
                   / \               / \   \         / \
                  6   10            6   8   12      0   2
                     /
                    8
```

[6] [a] If the find operations are done first, then the find operations take O(1) time each because every item is the root of its own tree. No item has a parent, so finding the set an item is in takes a fixed number of operations.

Union operations always take O(1) time. Hence, a sequence of n operations with all the finds before the unions takes O(n) time.

[b] This question requires amortized analysis. Find operations can be expensive, but an expensive find operation is balanced out by lots of cheap union operations. The accounting is as follows.

Union operations always take O(1) time, so let's say they have an actual cost of $1. Assign each union operation an amortized cost of $2, so every union operation puts $1 in the bank.

Each union operation creates a new child. (Some node that was not a child of any other node before is a child now.) When all the union operations are done, there is $1 in the bank for every child, or in other words, for every node with a depth of one or greater.

Let's say that a find(u) operation costs $1 if u is a root. For any other node, the find operation costs an additional $1 for each parent pointer the find operation traverses. So the actual cost is $(1 + d), where d is the depth of u.

Assign each find operation an amortized cost of $2. This covers the case where u is a root or a child of a root. For each additional parent pointer traversed, $1 is withdrawn from the bank to pay for it.

Fortunately, path compression changes the parent pointers of all the nodes we pay $1 to traverse, so these nodes become children of the root. All of the traversed nodes whose depths are 2 or greater move up, so their depths are now 1. We will never have to pay to traverse these nodes again.

Say that a node is a grandchild if its depth is 2 or greater. Every time find(u) visits a grandchild, $1 is withdrawn from the bank, but the grandchild is no longer a grandchild. So the maximum number of dollars that can ever be withdrawn from the bank is the number of grandchildren. But we initially put $1 in the bank for every child, and every grandchild is a child, so the bank balance will never drop below zero. Therefore, the amortization works out.

Union and find operations both have amortized costs of $2, so any sequence of n operations where all the unions are done first takes O(n) time.

[7] [a] insert 4, 2, 1, 3, and 6 in that order.
     [b] insert 4, 5, 6, 1, 3, and 2 in that order.

[8] [a] If you need inexact matches. For example, if you want to find the item less than or equal to 5 that's closest to 5. Hash tables can only do exact matches. (If exact matches are all you need, however, hash tables are faster.)

[b] If each single operation absolutely must run in O(log n) time. OR If most operations are find()s, and the data access patterns are uniformly random. (2-3-4 trees are faster for these operations because they don't restructure the tree. But splay trees do better if a small proportion of the items are targets of most of the finds.)

[c] If memory use is the primary consideration (especially if a 2-3-4 tree holding all the items won't fit in memory).

[d] None. find() and remove() on a heap take worst-case Theta(n) time, and they're more complicated than in an unordered array. insert() on a heap takes worst-case Theta(log n) time, versus Theta(1) for an unordered array.

[e] When you don't need to find the minimum key.

```
[10]  [a]          A
               A       E
             C   S   R   T
            U D T U T R

      [b]          A
               A       E
             C   S   R   R    (Note that two nodes are different than in [a].)
            U D T U T T
```

```
[12]  DATASTRUCTURE
      DACASTRUTTURE
      DACA|E|TRUTTURS
      DACA|E|RRUTTUTS
      DACA|E|RR|S|TTUTU
      DACA|E|R|R|S|TTUTU
               ^
```

[13] Radix sort takes b/log2 r passes, so the overall running time of radix sort is

```
                  n + r
     t =  b (ln 2) -----
                  ln r
```

To find the value of r that minimizes t, set dt/dr to zero.

```
     dt           ln r - (n + r)/r
     -- = b (ln 2) ---------------- = 0
     dr               (ln r)^2
```

Therefore, ln r = (n + r)/r. Given that n = 493, with a calculator and some trial-and-error you can determine that r = 128 is the optimal radix.

```
[14]  [a]  z:  consider the case where, half-way through the sort, the last key
               in the "sorted" list is changed to a very low number.

                     1 3 5 7 9│8 4 2 6 10
                zap! 1 3 5 7 0│8 4 2 6 10

               Using an in-place insertion sort implementation that searches
               from the end of the sorted array, the remaining keys will never
               get past the zero.

                     1 3 5 7 0 2 4 6 8 10

               Note that if a key in the "unsorted list" is zapped, no harm is
               done at all.

      [b]  y:  If an item in the "sorted list" is zapped, only that one item
               is affected.  If an item in the "unsorted list" is zapped to
               a value lower than the last item in the sorted list, that item
               will be out-of-place, but other items are still sorted.

      [c]  z:  Consider merging two lists, where the first item in one of the
               lists gets zapped to a very high value.  You'll wind up with two
               consecutive sorted portions.  (After further merge operations,
               there will still be two consecutive sorted portions.)

                                    /== 100 3 5 7 9 11
                             merge  \== 2 4 6 8 10 12

      [d]  y:  Radix sort uses no comparisons at all, so the zapped item
               doesn't affect how the others are ordered.
```