Problem Set 14 - Solution

Sorting/Dijkstra's Algorithm Analysis Review

1. Trace the quicksort algorithm on the following array:

```
3, 26, 67, 25, 9, -6, 43, 82, 10, 54
```

Use the median of the first, middle, and last entries as the pivot to split a subarray. (If a subarray has fewer than 3 entries, use the first as the pivot.) Show the quicksort tree and the number of comparisons at each split. **SOLUTION**

```
        [3  26  67  25  9  -6  43  82  10  54]   median(3,9,54)=9, swap(9,3)
                        split
                         | pivot=9, # of comp = 9
                         V
        [3  -6]          9         [25  67  26  43  82  10  54]
          split                               split
            | piv=3,#c = 1                      | piv=43,#c = 6
            V                                   V
      [-6]  3                   [25  10  26]   43     [82  67  54]
                                    split                 split
                                      | piv=25,#c=2         | piv=67,#c=2
                                      V                     V
                                [10]  25   [26]       [54]  67  [82]
```

Total number of comparisons = 20

2. Given the following input array:

```
3, 26, 67, 25, 9, -6, 43, 82, 10, 54
```

1. Trace the linear time build-heap algorithm on this array, to build a max-heap. How many comparisons did it take to build the heap?
2. Starting with this max-heap, show how the array is then sorted by repeatedly moving the maximum entry to the end and applying sift-down to restore the (remaining) heap. How many comparisons did it take to sort the heap?

SOLUTION

1.
```
               Array                                    Sift Down    Comparisons
        ---------------------------------------------   ---------    -----------
        3   26   67   25   9   -6   43   82   10   54       9             1

        3   26   67   25   54   -6   43   82   10   9       25            2

        3   26   67   82   54   -6   43   25   10   9       67            2

        3   26   67   82   54   -6   43   25   10   9       26            4

        3   82   67   26   54   -6   43   25   10   9        3            5

        82   54   67   26   9   -6   43   25   10   3       done
```

2.
```
               Array                                    Sift Down    Comparisons
        ---------------------------------------------   ---------    -----------
        82   54   67   26   9   -6   43   25   10   3
                        swap(82,3)
        3   54   67   26   9   -6   43   25   10   82        3             4


        67   54   43   26   9   -6   3   25   10   82
                        swap(67,10)
        10   54   43   26   9   -6   3   25   67   82       10             6


        54   26   43   25   9   -6   3   10   67   82
                        swap(54,10)
        10   26   43   25   9   -6   3   54   67   82       10             4


        43   26   10   25   9   -6   3   54   67   82
                        swap(43,3)
        3   26   10   25   9   -6   43   54   67   82        3             4


        26   25   10   3   9   -6   43   54   67   82
                        swap(26,-6)
        -6   25   10   3   9   26   43   54   67   82       -6             4


        25   9   10   3   -6   26   43   54   67   82
                        swap(25,-6)
        -6   9   10   3   25   26   43   54   67   82       -6             2


        10   9   -6   3   25   26   43   54   67   82
                        swap(10,3)
        3   9   -6   10   25   26   43   54   67   82        3             2


        9   3   -6   10   25   26   43   54   67   82
                        swap(9,-6)
        -6   3   9   10   25   26   43   54   67   82       -6             1


        3   -6   9   10   25   26   43   54   67   82
                        swap(3,-6)
        -6   3   9   10   25   26   43   54   67   82      done
```

3. A *stable* sorting algorithm is one which preserves the order of duplicate elements when sorted. For instance, suppose the following pairs of values are sorted on the **first** value in the pair:

```
(3,sun)  (2,mars)  (4,moon)  (3,venus)
```

then the output of a stable sorting algorithm would be:

```
(2,mars)  (3,sun)  (3,venus)  (4,moon)
```

Notice that `(3,sun)` comes before `(3,venus)`, preserving the order of the input for elements that have the same sortable value of `3`, hence *stable*.

However, if the output is this:

```
(2,mars)  (3,venus)  (3,sun)  (4,moon)
```

then the sorting algorithm is not stable since it does not preserve the input order of `(3,sun)` before `(3,venus)`.

For each of insertion sort, mergesort, and quicksort, tell whether the algorithm is stable or not.

SOLUTION

- Insertion sort is a stable sort.

- Mergesort is a stable sort.
- Quicksort is not a stable sort.

Problems 4-8 below are on analyzing the worst case big O running time of Dijkstra's shortest paths algorithm. Assume the graph has *n* vertices, and *e* edges with positive weights. And, except for problem #8 the graph is stored in adjacency linked lists.

The first step in the analysis is to identify the primary activities that are to be counted towards the running time. These are:

- (A)dding a vertex to the fringe
- (P)icking the minimum distance vertex from the fringe
- (C)hecking the neighbors of a vertex, for possible distance update
- (U)pdating (lowering) the distance of a vertex that is in the fringe

We will ignore the time to set the initial distance of a vertex. The reason is it takes unit time to set the distance in the distance array, but is done done as a part of the (A)dding a vertex to the fringe, which will take at least unit time. So as far as big O is concerned, determining the time to (A)dd fringe vertices will also account for distance initialization.

We will also ignore the time to set the previous vertex. Using a similar argument as above, it takes unit time to set the previous vertex in the previous array. But this is done as a part of (A)dding a vertex to the fringe, or (U)pdating the distance of a vertex, and the (A)dd and (U)pdate will take at least unit time themselves. So accounting for the (A)dd and (U)pdate times will suffice as far as the big O is concerned.

The fringe is the variable part as far the data structures are concerned (since distance and previous vertex are recorded in arrays), and the worst case depends on exactly how the fringe is implemented. You will compute the running times for different versions of the fringe in problems 5-7.

---

4. What is the total running time, through the entire run of the algorithm, for (C)hecking the neighbors of each vertex for possible distance update? Does this depend on the fringe?

   SOLUTION

   The running time for (C)hecking the neighbors of each vertex does not depend on the fringe, since the neighbors and edge weights are obtained from the adjacency linked lists, and the current distance is obtained from the distance array.

   The total number of neighbors of all vertices is *e*, if the graph is directed, and *2e*, if the graph is undirected. Each neighbor contributes one unit of time toward the distance check, so we get *O(e)* for the total running time.

---

5. If the fringe was implemented as an unordered linked list (not arranged in any specific order of distances), what would be the big O worst case running time of each of the A, P, and U components of the algorithm, through the entire run from start to end? What would be the total worst case big O running time of the algorithm?

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since fringe is an unordered linked list, adds can be done at the front of the linked list in O(1) time each, so the total time is *(n-1)**O(1), which is *O(n)*
   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Identifying the minimum distance vertex in an unordered linked list of size *k* will take *(k-1)* time units. So the series of minimum picks will add up as follows:

     $$(n-2) + (n-3) + \ldots + 2 + 1 = (n-1)*(n-2)/2$$

     which is $O(n^2)$
   - (U): Updating distance of vertex in fringe.
     The updates are done in the distance array, the structure of the fringe itself is not relevant. Each update can be done in O(1) time. In the worst case, every neighbor distance check results in an update. Since there are *O(e)* neighbor distance checks in all, this gives a total time of *O(e)*

   Adding these times, and factoring in the time of *O(e)* for the (C) component worked out in the previous problem, gives us a total time of:

   $$O(n) + O(n^2) + O(e) + O(e) = O(n^2)$$

---

6. Repeat the analysis for a fringe implemented as a linked list maintained in ascending order of distances.

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since fringe is an ordered linked list, each add needs to first find the correct spot, which will take *(k-1)* comparisons in the worst case for a linked list of length *k*. So the total time is:

     $$0 + 1 + 2 + \ldots + (n-3) + (n-2) = (n-1)*(n-2)/2 = O(n^2)$$

   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Since the linked list is arranged in ascending order of distances, the minimum distance vertex is the first item, and can be accessed in unit time. So the total time is *O(n)*
   - (U): Updating distance of vertex in fringe.
     While the actual updates are done in the distance array, because the fringe is maintained in ascending order of distances, an update would in general require a repositioning of the vertex in the linked list.

     The new distance would need to be compared with the distance of the vertex before it in the linked list. If the new distance is less, then another comparison would need to made with the vertex two spots before this vertex, etc. In the worst case, this sequence of comparisons would go all the way to the front of the list, so that if the list is of length *k*, then *(k-1)* comparisons would need to be made to move the last item all the way to the front.

     For the worst case for all updates, we assume that all *e* updates are done on the maximum possible fringe size, which is *n-2* (updates are done after picking the first vertex out of the initial fringe of size (n-1), which leaves (n-2) in the fringe).

     The time for a single update on this fringe size would be O(*e*), as outlined in the previous paragraph, so time for all *e* updates would be O(*en*)

   Adding these times, and factoring in the time of *O(e)* for the (C) component, gives us a total time of:

   $$O(n^2) + O(n) + O(en) + O(e) = O(n^2+en)$$

---

7. Repeat the analysis for a fringe implemented as a min-heap that supports inserts, deletes, and priority updates in *O(log n)* time.

   SOLUTION

   Here are the times for the individual components:

   - A: Initially adding *n-1* vertices to the fringe.
     Starting with an empty fringe, add one vertex at a time on fringes of increasing size. Since the fringe is a min-heap, adding (which includes sifting up) in a heap of size *k* would take *O(log k)* time. So the total time is:

     $$\log 1 + \log 2 + \log 3 \ldots + \log (n-1) = \log (n-1)! = O(n\log n)$$

   - P: Picking min dist vertex from the fringe.
     Starting with a fringe of size *n-1*, pick vertices from fringes of decreasing size. Again, since deleting from a heap of size *k* would take *O(log k)* time, the total time is:

     $$\log (n-2) + \log(n-3) \ldots + \log 2 + \log 1 = \log (n-1)! = O(n\log n)$$

     (The first term is log(n-2) because sift down is done after deleting the top of the heap, on the remaining (n-2) items.)

   - U: Updating distance of vertex in fringe.
     Updating the priority (which here means decreasing the distance) of an item in a heap of size *k* takes *O(log k)* time. For the worst case for all updates, we assume that all *e* updates are done on the maximum possible fringe size, which is *n-1*, so time for all *e* updates would be O(*elogn*)

   The total running time, adding the time for all components, and bringing in the time for (C) is:

   $$O(n\log n) + O(n\log n) + O(e\log n) + O(e) = O((n+e)\log n)$$

---

8. Repeat the analysis for a fringe implemented as a min-heap that supports inserts, deletes, and priority updates in *O(log n)* time, for a graph stored in an adjacency *matrix* format instead of adjacency linked lists.

   SOLUTION

   In #7 above, we analyzed the running time for an updatable-heap based fringe, and it turned out to be O((*n*+*e*)*logn*). However this assumed that the graph was stored in adjacency linked format, which resulted in an O(*e*) running time for the (C) component. But if the graph is stored in an adjacency matrix, the running time of the (C) component would change to $O(n^2)$. All other components would have the same running time since they all depend only on the fringe.

   Since the (C) component was not the dominant one in the original analysis, we can simply replace the old running time of (C) with $O(n^2)$, which changes the running time to:

   $$O(n\log n) + O(e\log n) + O(n^2) = O(n^2+e\log n)$$

   (Since n^2 is of a higher order than *nlogn*. the *nlogn* term is taken out)