Patrick Randell 836026

Design of Algorithms Assignment 2 – Analysis

Question 4 – Average Case Complexity of Quicksort

a) If the pivot is guaranteed to lie between the top 75% and top 25% of elements in an unsorted array, then the worst case in this range of pivot selections, would be when the pivot selected is on the very fringe of the top 25%, or top 75%. This would mean that the array is divided into two unequal parts, one with 75% of the original array, the other with 25%. The following assumes the pivot can be chosen in this range in constant time.

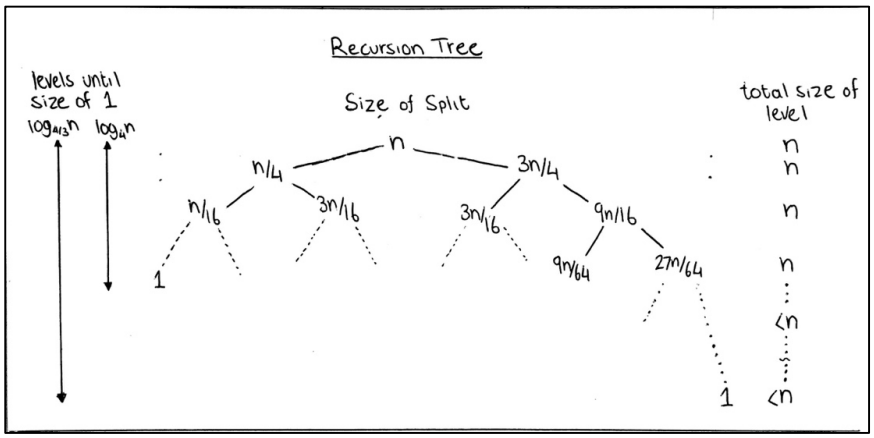

Figure 4.1 – Recursion Tree for ¼ - ¾ split.

As we can see from the recursion tree, on the left side, there are $\log_4 n$ levels until we reach a sub-problem size of one. On the right hand side, there are $\log_{4/3} n$ levels. For each of the first $\log_4 n$ levels, there are n total elements to be partitioned (incl. the pivot, for simplicity), and thus the partitioning time for these levels is $O(n)$ (Assuming it takes constant time to place an element in it's position). For the remaining levels, the total elements are *less than* n. Thus, the partitioning time is *at most* $O(n)$. Thus, with $\log_{4/3} n$ total levels bounded above by $O(n)$ runtimes, the asymptotic time complexity is:

$$O\big(n \cdot \log_{4/3} n\big) = O(n \cdot \log n)$$

b) Because 50% of all elements lie in-between the top 25% and top 75%, we can reasonably assume that we will select a pivot in this region half (0.5 or 50%) of the time.

c) If we assume that each time that we don't get a pivot in the middle 50% region (and therefore a $O(n \cdot \log n)$ runtime), we instead get the worst case scenario where we partition the array into sizes of 1 and $n - 1$, all we are doing is essentially doubling the height/size of the recursion tree in *Figure 4.1.* This is illustrated in *Figure 4.2.*

Thus, the average case time complexity of quicksort is:

$$2 \cdot O(n \cdot \log n) = O(n \cdot \log n)$$

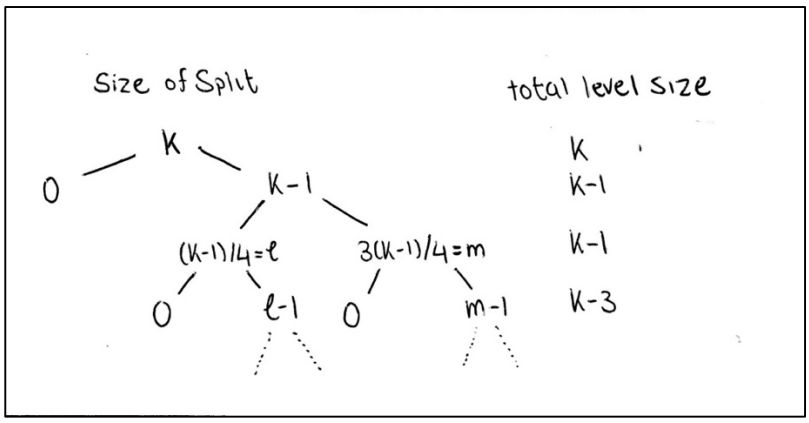

Figure 4.2 – Average Case Recursion Tree.

Question 5 – Lexographical Optimisation with Paths.

The way I decided to tackle this problem was to use a modified version of Dijkstra's algorithm that selects nodes based on their smallest minmax edge, and amongst these, the one with the shortest path. Using a min-heap for the priority queue with a modified Heapify function, that assigns priorities based first off the smallest minmax edge, and breaks ties using the shortest path. Dijkstra's runs as normal, except that it updates the distance only if it has the same, or updates the minmax value first. A path rebuilding function is used, that uses the array of previous values and rebuilds the path to a Stack which can be popped in order.

Using an Adjacency List Representation for the graph, and a Min-Heap for the priority queue, used to pass the nodes with minimum minmax edge weight to Dijkstra's, this process has a run time complexity of $O(E \cdot \log V)$. The separate print function returns a Stack containing the path to the end node, that can be popped in order, and the total cost of this path.

```
function HEAPIFY (arr [minmax, dist], n, i)          ▷ 2D array with minmax & dist
    left ← LEFT(i)
    right ← RIGHT(i)
    if left < n & (minmax[left] < minmax[i] or
    (minmax[left] = minmax[i] & dist[left] < dist[i])) do
        smallest ← left
    else do
        smallest ← i

    if right ≤ n & (minmax[right] < minmax[smallest] or
    (minmax[right] = minmax[smallest] & dist[right] < dist[smallest])) do
        smallest ← right
    if smallest ≠ i do
        SWAP(minmax[i], minmax[smallest])          ▷ Essentially swap arr pos's
        SWAP(dist[i], dist[smallest])
        HEAPIFY(arr, n, smallest)
```

```
function MINMAXWEIGHTPATH({V, E}, v_s, v_e)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
        minmax[v] ← ∞
    dist [v_s] ← 0
    minmax[v_s] ← 0
    Q ← INITMINHEAP(V)          ▷ See Heapify function, update heap uses same concept
    while Q is non-empty do
        u ← EXTRACTMIN(Q)
        for each (u, w) ∈ E do
            tempminmax ← MAX(weight(u, w), minmax[u])
            if w ∈ Q & tempminmax = minmax[w] then
                if dist[u] + weight(u, w) < dist[w] then
                    dist[w] ← dist[u] + weight(u, w)
                    prev[w] ← u
                    UPDATEMINHEAP(Q, w, dist[w], minmax[w])
            if w ∈ Q & tempminmax < minmax[w] then
                minmax[w] ← tempminmax
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATEMINHEAP(Q, w, dist[w], minmax[w])
    PRINTPATHANDCOST(prev, dist, v_s, v_e, V)
function UPDATEMINHEAP(Q, w, dist[w], minmax[w])
    ▷ Every Q node holds position and priority (minmax value), and dist value
    i ← position[w]
    Q[i] → minmax ← minmax[w]
    Q[i] → dist ← dist[w]
    while (i < 0 & (Q[i] → minmax ≤ Q[⌈(i−1)/2⌉] →minmax)) do
        if (Q[i] → minmax = Q[⌈(i−1)/2⌉] →minmax) then
            if (Q[i] → dist < Q[⌈(i−1)/2⌉] → dist) then
                SWAP(Q[i], Q[⌈(i−1)/2⌉])
        else do
            SWAP(Q[i], Q[⌈(i−1)/2⌉])
        i ← (i−1)/2

function PRINTPATHANDCOST(prev, dist, v_s, v_e, V)
    S ← INITSTACK(V)
    PUSH(S, v_e)
    u ← prev[v_e]
    while v_s < u do
        PUSH(S, u)
        u ←prev[u]
return (S, dist[v_e])
```

We can see that updating the distance and previous values, and calculating the tempminmax value using MAX() function are constant time operations (MAX() is a simple comparison). UPDATEMINHEAP will be a $\log V$ operation, as at most it moves an item the height of the heap.

For the modified Dijkstra's algorithm, we can see that updating the cost of nodes is the basic operation, which is executed $(E + V)$ times. Assigning variables takes constant time, but updating the min heap and recursively restoring its heap properties is a $O(\log V)$ operation. Thus this takes $O((E + V) \cdot \log V) = O(E \cdot \log V)$. The printing function's runtime depends on the position of the end node, and the number of nodes in the path back to the source. If we assume (as a worst case scenario) it's the last node, and every node is in the path, then the pushing and popping operations occur V times each, $2V$ in total.

Thus, the entire process has a runtime complexity of $O(E \cdot \log V + 2V) = O(E \cdot \log V)$. (This assumes more edges than vertices, otherwise it's simply:

$$O((E + V) \cdot \log V)$$

Question 6 – Weighted Graph Reduction

The simplest way to approach this problem is to break every weighted edge into single edges of length 1, by inserting additional nodes as 'stepping stones' for the original edge. These nodes will always have an in-degree and out-degree of 1, and the number of these nodes for any given weighted edge will be equal to 1 less than the weight of that edge. This is illustrated below.

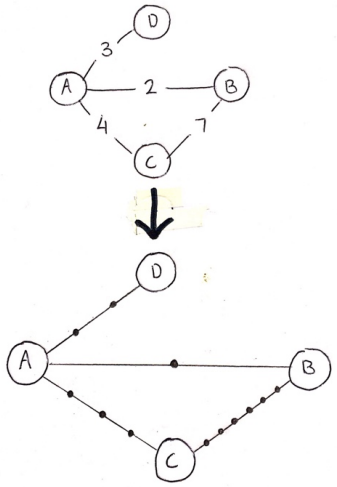

Figure 6.1 – Transform from Weighted to Unweighted Graph

We can see that the number 'stepping stone' nodes is always 1 less than the weight of the edge they're replacing. The key idea here is to give well chosen names to the stepping stone nodes, so that once $UNWEIGHTEDAPSP$ outputs all the shortest paths, we can simply ignore those that include stepping stone nodes.

```
function WEIGHTEDAPSP(G{V, E})
    G' ← INITNEWGRAPH(V)                          ▷ Add existing nodes to G'
    for each u ∈ G(V) do
        for each (u, v) ∈ G(E) do
            for i ← 1 to weight(u, v) − 1 do
                uvi ← CREATENODE(G')              ▷ Unique name (e.g AB1)
                if i equals 1 do
                    u ← ADDEDGE(G', u, uvi)
                    uvi ← ADDEDGE(G', uvi, u)
                else if i equals weight(u, v) − 1 do
                    v ← ADDEDGE(G', v, uvi)
```

```
                    uvi ← ADDEDGE(G', uvi, v)
                    REMOVEEDGE(G, v, u)          ▷ No repetition of edges
                else do
                    uv(i − 1) ← ADDEDGE(G', uv(i − 1), uvi)
                    uvi ← ADDEDGE(G', uvi, uv(i − 1))

    list ← UNWEIGHTEDAPSP(G'{V, E})
    for each (u, v) ∈ G(V) in list do        ▷ Only add paths from original nodes
        newlist ← ADDPATH
    return newlist
```

The output from $UNWEIGHTEDAPSP(G'\{V, E\})$ will already contain the shortest paths from our original nodes, it will just contain a lot of other shortest paths between stepping stone nodes which we don't care about. Since we are dealing with un-directed graphs, the REMOVEEDGE function ensures that we don't traverse over any edges (and therefore create more nodes) twice.

Question 7 – Heaps

a) Heap Algorithm Analysis

```
function PROBLEM1A()
    n ← GETINPUT()
    arr ← INITARRAY(n + 1)
    for i ← 1 to n do
        arr[i] ← GETINPUT()
    for i ← parent(n) to 1 do
        HEAPIFY(arr, n, i)
    for i ← 1 to n do
        PRINT(arr[i])
```

This roughly translates to the Psuedocode below, as the loop to initialise the array will not add to the asymptotic time complexity (as we are looping through n elements regardless).

```
function CONSTRUCTHEAP(arr, n)
    for i ← PARENT(n) to 1 do
        HEAPIFY(arr, n, i)
    for i ← 1 to n do
        PRINT(arr[i])

function HEAPIFY(arr, n, i)
    left ← LEFT(i)
    right ← RIGHT(i)
    if left < n and arr[left] > arr[i] do
        largest ← left
    else do
        largest ← i
    if right ≤ n and arr[right] > arr[largest] do
        largest ← right
    if largest ≠ i do
```

$SWAP(arr[i], arr[largest])$
$HEAPIFY(arr, n, largest)$

In this case, $SWAP$ is a straight forward function. $LEFT$, $RIGHT$ & $PARENT$ are simple $O(1)$ functions that simply transform an array index to that of its left, right, or parent node.
Thus, the time complexity of $CONSTRUCTHEAP$ is governed by the n calls to $HEAPIFY$. The time complexity of $HEAPIFY$ depends entirely on the height of the element i being passed to $HEAPIFY$, and the proportion of total elements at that height. We know that the height of a heap's root is equal to $h = \log_2 n = \log n$. Thus, the total work required by $HEAPIFY$ is equal to the distance (in terms of height) a node at a given height has to move to restore the heap property. The total work required by $CONSTRUCTHEAP$ will be number of elements at the given heights, multiplied by the work of $HEAPIFY$ at that height:

Level:	Bottom	1	2	3	…	$h-1$	h
Level × Number:	$0 \cdot \frac{n}{2}$	$1 \cdot \frac{n}{4}$	$2 \cdot \frac{n}{8}$	$3 \cdot \frac{n}{16}$	…	$(h-1) \cdot 2$	$h \cdot 1$

The total work of $CONSTRUCTHEAP$, $T(n)$ will be the sum of all these terms:

$$T(n) = \sum_{h=1}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h)$$
$$< O\left(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

Using the properties of Big-Oh notation to remove the ceiling function, the constant associated with 2^{h+1} and increase the upper sum to infinity, we now have a converging geometric series:

$$\sum_{a=0}^{\infty} x^a = \frac{1}{1-x} \; since \; (x < 1)$$
$$\Rightarrow \sum_{a=0}^{\infty} a x^a = \frac{x}{(1-x)^2} \; (diff \; both \; sides \; and \times by \; x)$$

thus,

$$T(n) = O\left(n \cdot \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} \right)$$
$$= O(n \cdot 2)$$
$$= O(n)$$

Thus, $CONSTRUCTHEAP$ is an $O(n)$ function.

b) Heap Algorithm Analysis

function $PROBLEM1B()$
    $n \leftarrow$ GETINPUT()
    $arr \leftarrow$ INITARRAY$(n+1)$
    for $i \leftarrow 1$ to n **do**
        $arr[i] \leftarrow$ GETINPUT()
    for $i \leftarrow parent(n)$ to 0 **do**
        $HEAPIFY(arr, n, i)$

for $i \leftarrow 1$ to $(n+1)/2$ **do**
    $RIGHTHEAPIFY(arr, n, i)$
for $i \leftarrow 1$ to n **do**
    $PRINT(arr[i])$

This roughly translates to the Psuedocode below, as the loop to initialise the array will not add to the asymptotic time complexity (as we are looping through n elements regardless).

function $CONSTRUCTRIGHTHANDEDHEAP(arr, n)$
for $i \leftarrow PARENT(n)$ to 0 **do**
    $HEAPIFY(arr, n, i)$
for $i \leftarrow 1$ to $(n+1)/2$ **do**
    $RIGHTHEAPIFY(arr, n, i)$
for $i \leftarrow 1$ to n **do**
    $PRINT(arr[i])$

function $RIGHTHEAPIFY(arr, n, i)$
left $\leftarrow LEFT(i)$
right $\leftarrow RIGHT(i)$
if *right* $\leq n$ **and** $arr[$*right*$] < arr[$*left*$]$ **do**
    $SWAP(arr[$*right*$], arr[$*left*$])$
    $HEAPIFY(arr, n,$ *left*$)$

In this case, $SWAP, LEFT, RIGHT, PARENT$ & $HEAPIFY$ are the same as in part a) of this question. Thus, the time complexity of $CONSTRUCTRIGHTHANDEDHEAP$ will be the addition of the time complexity of $CONSTRUCTHEAP$, and the $\approx \frac{n}{2}$ calls to $RIGHTHEAPIFY$.

The time complexity of $RIGHTHEAPIFY$ can be analysed by considering the worst case scenario, where every parent node has it's left node, initially bigger than its right node, additionally to this, the difference in size between every left and right child node is large enough such that every sub-node down to the root from the right child is bigger than the left child. In other words, **every call to $RIGHTHEAPIFY$ will result in a swap, and a $HEAPIFY$ call to the bottom of the heap.**

In this case, the work done at every call to $RIGHTHEAPIFY$ will be equal to swapping the right and left child, and moving the new left child down to the bottom of the heap to restore the heap property (Since the new right child is bigger, the right sub-heap will already be heapified). This work will be equal to:

$$1 + height(Parent) - 1 = height(Parent)$$

The total work over the n/2 calls will then be the number of nodes at each height, multiplied by the work done by $RIGHTHEAPIFY$ at that height:

Level/Height:	Bottom (0)	1	2	3	…	$h-1$	h
Level × Number:	$0 \cdot \frac{n}{2}$	$1 \cdot \frac{n}{4}$	$2 \cdot \frac{n}{8}$	$3 \cdot \frac{n}{16}$	…	$(h-1) \cdot 2$	$h \cdot 1$

We can now see that the same situation as for $CONSTRUCTHEAP$ arises, where the time complexity is $O(n)$ and thus the total time complexity of $CONSTRUCTRIGHTHANDEDHEAP$ will be:

$$\begin{aligned} T(CONSTRUCTRIGHTHANDEDHEAP) &= T(CONSTRUCTHEAP) + O(n) \\ &= O(n) + O(n) \\ &= O(n) \end{aligned}$$

Note, that the likely hood of every call to $RIGHTHEAPIFY$ resulting in a swap is extremely unlikely.

Question 8 – Heap Top-*k*

The section of Prof. Dubious' claim where the problem arises is:

"Clearly, the largest element in H is at the root. Since the left and right subtrees are also heaps the next two largest elements must be at the root of each of these subtrees, i.e., the left and right child of the root."

The distinction to be made is that the root of a subtree is only the largest element of that/their OWN tree/subtree. The left and right children of a root, need not be the next largest elements of the tree belonging to that root, to still have the property that every node Is larger than its children (i.e, be a heap).

As illustrated in *Figure 8.1* The top 3 elements of a max heap need not necessarily be in the top 2 layers, as 80 (in layer 3) is larger than 30 (in layer 2). While it is true that the largest $2^k - 1$ elements must be **within** the top $2^k - 1$ layers, this claim is not really significant.

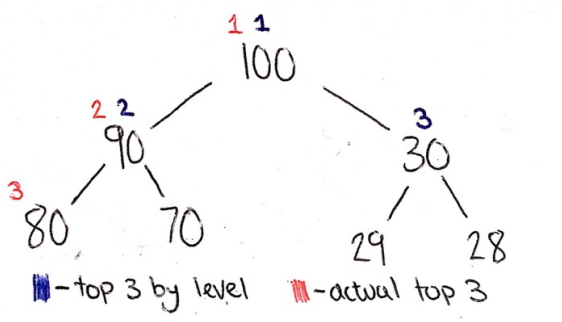

Figure 8.1 – Max Heap

In order to obtain the top-*k* elements using heaps, one can use a min-heap with *k* elements, and sequentially replace the root with the remaining n-k elements, until the top k remain in the heap.

Resources

- **Dan Sunday,** *iSurfer.org, "Fast Convex Hull of a 2D Simple Polyline"* <http://geomalgorithms.com/a12-_hull-3.html>
- **Various Authors,** *GeeksforGeeks: "Bellman–Ford Algorithm | DP-23", "Sorted Array to Balanced BST", "Level Order Tree Traversal", "Convert Normal BST to Balanced BST", "Shortest Path for Directed Acyclic Graphs".*

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>
<https://www.geeksforgeeks.org/sorted-array-to-balanced-bst/>
<https://www.geeksforgeeks.org/level-order-tree-traversal/>
<https://www.geeksforgeeks.org/convert-normal-bst-balanced-bst/>
<https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/>
• **Lars Kulik,** *COMP20007 Design of Algorithms, "Greedy Algorithms: Prim and Dijkstra - Lecture 8".* Semester 1,2019.
• **Lars Kulik,** *COMP20007 Design of Algorithms, "Graph Traversal - Lecture 7".* Semester 1,2019.
• **Emir Demorivic,** *COMP20007 Design of Algorithms,* ***various lectures****.* Semester 1,2019.
• **Tobias Edwards,** *COMP20007 Design of Algorithms,* ***various workshop tute sheets and coding examples****.* Semester 1,2019.