ICS 311, Spring 2016, Problem Set 05, Topics 9 (Heaps) & 10A (Quicksort) -- Solutions

---

#1. Peer Credit Assignment

1 Point Extra Credit for replying

You should have named your group partners and allocated 6 points total across them for class 2/17 and 2/24 (section 1). If you did not, please email the TA with your points assignment.

#2. Analysis of d-ary heaps (11 pts)

In class you did preliminary analysis of ternary heaps. Here we generalize to *d*-ary heaps: heaps in which non-leaf nodes (except possibly one) have *d* children.

a. (3) How would you represent a *d*-ary heap in an array? Answer this question by

- Giving an expression for **Jth-Child(i,j)**: the index of the *j*th child as a function of *j* and the index *i* of a given node, and
- Giving an expression for **D-Ary-Parent(i)**: the index of the parent of a node as a function of its index *i*.
- Checking that your solution works by showing that D-Ary-Parent(Jth-Child(i,j)) = i (if you start at node i, apply your formula to go to a child, and then your other formula to go back to the parent, you end up back at i).

Solution:

Jth-Child(i, j) = d(i - 1) + j + 1
D-Ary-Parent(i) = floor((i - 2)/d + 1)

Partial proof of correctness:
D-Ary-Parent(Jth-Child(i,j))
= D-Ary-Parent(d(i - 1) + j + 1)
= floor((d(i - 1) + j + 1 - 2)/d + 1)
= floor((di - d + j - 1)/d + 1)
= floor(i - 1 + j/d - 1/d + 1)
= floor(i + (j-1)/d) *// but j <= d so (j-1) < d so floor((j-1)/d) = 0 so*

= i

b. (2) What is the height of a *d*-ary heap of *n* elements as a function of *n* and *d*? By what factor does this height differ from that of a binary heap of n elements?

> **Solution :** $\Theta(\log_d n) = \Theta(\lg n / \lg d)$, differing from binary heaps by a constant factor of 1/lg d.

c. (3) Give an efficient implementation of EXTRACT-MAX in a *d*-ary max-heap. (Hint: consider how you would modify existing code.) Analyze its running time in terms of *n* and *d*. (*Note that d must be part of your Θ expression.*)

> **Solution:** Use Heap-Extract-Max, but change Max-Heapify to compare to all d children. (Student may write code here.)
> Time is proportional to height of heap times number of children:
> $\Theta(\lg n / \lg d) * d = \Theta(d \lg n / \lg d)$ or $\Theta(d \log_d n)$

d. (3) Give an efficient implementation of INSERT in a *d*-ary max-heap. Analyze its running time in terms of *n* and *d*.

> **Solution:** Use existing Insert but change Heap-Increase-Key to call the D-Ary-Parent. (Student may write code here.)
> Since Heap-Increase-Key climbs the tree, in the worst case it must climb $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

#3. Tracing Quicksort (12 pts)

Show the operation of Partition (not randomized) on this 1-based array:

A = [1, 6, 2, 8, 3, 9, 4, 7, 5], p=1, q=9

and the two sub-partitions that result as directed below. In other words, you will trace the three calls to Partition that are highest in the recursion tree. (They are *not* the first three calls: part **a** is for the first call and part **b** is for the second call, but part **c** is for the call that takes place after all the recursive calls breaking down the first partition.)

In order to make the desired response format clear and to make it easy for the TA to grade, we are providing a template for your response. You are to fill in wherever the underscore character "_" appears. Use a plain text editor with *fixed-width font*. Be sure

to fill in all fields marked with underscore: use search to make sure you get them all. We start you off with the first few lines: continue in the same pattern.

a. (4) Call to Partition (A, 1, 9) made in Line 2 of the initial call to Quicksort:

Initially:
A = [1, 6, 2, 8, 3, 9, 4, 7, 5], i=0, j=1, pivot = A[r] = A[9] = 5

Trace at the conclusion of each pass through the loop lines 3-6
A = [1, 6, 2, 8, 3, 9, 4, 7, 5], i=1, j=1, exchanged A[1] with A[1]
A = [1, 6, 2, 8, 3, 9, 4, 7, 5], i=1, j=2, no exchange
A = [1, 2, 6, 8, 3, 9, 4, 7, 5], i=2, j=3, exchanged A[2] with A[3]

... you fill in the rest until the loop exits ... **Solution in bold:**

A = [1, 2, 6, 8, 3, 9, 4, 7, 5], i=2, j=4, no exchange
A = [1, 2, 3, 8, 6, 9, 4, 7, 5], i=3, j=5, exchanged A[3] with A[5]
A = [1, 2, 3, 8, 6, 9, 4, 7, 5], i=3, j=6, no exchange
A = [1, 2, 3, 4, 6, 9, 8, 7, 5], i=4, j=7, exchanged A[4] with A[7]
A = [1, 2, 3, 4, 6, 9, 8, 7, 5], i=4, j=8, no exchange

After the swap in line 7:
A = [1, 2, 3, 4, 5, 9, 8, 7, 6], i=4, j=9, exchange A[5] with A[9]

What does Partition(A, 1, 9) return? **5**

Continuing execution of the top level call to Quicksort, identify the two partitions that will be handled by the recursive calls to Quicksort at this level:
On what subarray will Quicksort in line 3 be called? A[**1, 4**]
On what subarray will Quicksort in line 4 be called? A[**6, 9**]

Now in parts b and c we trace these two calls in a manner similar to above.

b. (3) Call to Partition handled in the first call to Quicksort line 3:

Initially:
A = [1, 2, 3, 4, ...], i=0, j=1, pivot = A[r] = A[4] = 4

Trace at the conclusion of each pass through the loop lines 3-6
A = [1, 2, 3, 4, ...], i=1, j=1, exchanged A[1] with A[1]
A = [1, 2, 3, 4, ...], i=2, j=2, exchanged A[2] with A[2]
A = [1, 2, 3, 4, ...], i=3, j=3, exchanged A[3] with A[3]

After the swap in line 7:
A = [1, 2, 3, 4, ...], i=3, j=4, exchanged A[4] with A[4]

What does this second call to Partition return? **4**

c. (3) Call to Partition handled in the first call to Quicksort line 4:

Initially:
A = [..., 9, 8, 7, 6], i=5, j=6, pivot = A[r] = A[9] = 6

Trace at the conclusion of each pass through the loop lines 3-6
A = [..., 9, 8, 7, 6], i=5, j=6, no exchange
A = [..., 9, 8, 7, 6], i=5, j=7, no exchange
A = [..., 9, 8, 7, 6], i=5, j=8, no exchange

After the swap in line 7:
A = [..., 6, 8, 7, 9], i=5, j=9, exchanged A[6] with A[9]

What does this third call to Partition return? **6**

d. (2) What patterns do you see in the second and third calls to Partition? How do you expect that this behavior will affect the runtime of Quicksort on data with these patterns? (You may answer without doing a formal analysis.)

Solution:
Pattern in the second call to Partition:

- **Continues to do useless exchanges for n=3, n=2 and n=1 subarrays.**

Pattern in the third call to Partition:

- **This now behaves like the previous case as the pivot is the largest element: it does useless exchanges for n=3. But then on the call after that the smallest element will be the pivot; it does no exchange for n=2. It alternates.**

Effect on the runtime of Quicksort on data organized as in these partitions:

- **We know that (nonrandomized) quicksort on sorted data is O(n^2). Although not as bad as fully sorted, the above trace suggests that it is also O(n^2) when sorted sequences are interleaved or embedded in the data. We need randomization for more than just the sorted case.**

#4. 3-way Quicksort (10 pts)

In class we saw that the runtime of Quicksort on a sequence of *n* identical items (i.e. all entries of the input array being the same) is $O(n^2)$. All items will be equal to the pivot, so n-1 items will be placed to the left. Therefore, the runtime of QuickSort will be determined by the recurrence $T(n) = T(n-1) + T(0) + O(n) = O(n^2)$ To avoid this case, we are going to design a new partition algorithm that partitions the array into three parts, those that are strictly less than the pivot, equal to the pivot, and strictly greater than the pivot.

a. (5) Develop a new algorithm *3WayPartition(A, p, r)* that takes as input array *A* and two indices *p* and *r* and returns a pair of indices (*e, g*). *3WayPartition* should partition the array *A* around the pivot *q = A*[*r*] such that every element of *A[p..(e-1)]* is strictly smaller than *q*, every element of *A[e..g-1]* is equal to *q* and every element of *A[g..r]* is strictly greater than *q*.
Hint: modify Partition(A,p,r) presented in the lecture notes/book, such that it adds the items that are greater than q from the right end of the array and all items that are equal to q to the right of all items that are smaller than q. You will need to keep additional indices that will track the locations in A where the next item should be written.

Solution (student code may differ):

```
3WayPartition(A, p, r)
1  q = A[r]                    // q is the pivot
2  l = p-1                     // invariant: A[p…l] are all less than the pivot
3  g = r+1                     // invariant: A[g...r] are all greater than the pivot
4  if (r > p)
5       i = p
6       while (i < g)
7               if (A[i] > pivot)
8               g = g - 1
9                       exchange A[i] and A[g]
10              else if (A[i] < pivot)
11                      l = l + 1
12                      exchange A[i] and A[l]
```

```
13                      i = i + 1
14              else
15                      i = i + 1
16  e = l+1     // A[l] is the last one that is smaller than pivot q, so A[l+1] = q
18  return (e, g)
```

The correctness of the algorithm follows from the loop invariants that A[p...l] are all less than the pivot q, A[g...r] are greater than the pivot q. It is easy to verify that the properties of the loop invariants are maintained for these two loop invariants.

The procedure terminates because in each iteration either *i* increases or *g* decreases, i.e. the distance between *i* and *g* always decreases.

b. (3) Develop a new algorithm *3WayQuicksort* which uses *3WayPartition* to sort a sequence of *n* items. Again, *3WayPartition* returns a pair of indices (e, g).

Solution:

```
3WayQuicksort(A, p, r)
1  if (p < r)
2       (e, g) = 3WayPartition(A, p, r)
3       3WayQuicksort(A, p, e-1)
4       3WayQuicksort(A, g, r)
```

c. (2) What is the runtime of *3WayQuicksort* on a sequence of *n* random items? What is the runtime of *3WayQuicksort* on a sequence of *n* identical items? Justify your answers.

Solution:

The expected runtime of 3WayQuicksort on a sequence of *n* random items is the same as expected runtime of randomizedQuicksort, i.e. O(n log n)

The runtime of 3WayQuicksort(A, 1, n) on a sequence of *n* identical answers is O(*n*) because the first call on 3WayPartition will return (e,g) = (0, n+1), because all items will be equal to the pivot A[n]. Thus, the recursive calls will be 3WayQuicksort(A, 1, 0) and 3WayQuicksort(A, n+1, n), which will return immediately because both times (p > r). Thus, the runtime is just a single call to 3WayPartition, which takes O(n) time.