Kyle Daling

CSCI 305

Assignment 3

May 17, 2017

Question 1

Line	Cost	Times
build_max_heap()	$\Theta(n)$	1
for $i = n$ downto 2	C_1	n
exchange A[1] and A[i]	C_2	$n-1$
heapsize -= 1	C_3	$n-1$
max_heapify(A,1)	??	$n-1$

The runtime of the algorithm is dependent on the runtime of max_heapfiy(A,1) within the for loop. When called at the root of the heap, max_heapify's runtime is bounded by $\mathcal{O}(h)$ where h is the height of the heap. In the for loop, exchange will always move a leaf to the top of the heap. The exchanged leaf will always make its way down the heap to a leaf node since it is smaller than all of the non-leaf nodes. Therefore the best case runtime of max_heapify will is bounded by $\Omega(h)$. The height of the heap is given by $\lg(n)$ where n is the number of nodes in the heap. Every iteration of the for loop the heap decreases in size by 1. Therefore we can calculate the runtime of all calls to max_heapify with the following summation.

$$\sum_{i=2}^{n} \lg i = \lg 1 + \sum_{i=2}^{n} \lg i = \sum_{i=1}^{n} \lg i = \lg 1 + \lg 2 + \ldots + \lg n = lg(1 \times 2 \times \ldots \times n) = \Theta(n \lg n)$$

Therefore the runtime of heapsort will be dominated by the runtime of max_heapify which is $\Theta(n \lg n)$. Therefore heapsort is both $\mathcal{O}(n \lg n)$ and $\Omega(n \lg n)$.

Question 2

The algorithm I developed to compute the k-th smallest integers is below.

```
build_min_heap()
for i = 1 to k
    min.push(heap_extract_min())
return min
```

To prove this algorithm is $\mathcal{O}(n + k \lg n)$ I will find the components of the runtime.

Line	Cost	Times
build_min_heap()	$\Theta(n)$	1
for $i = 1$ to k	C_1	k + 1
min.push(heap_extract_min())	??	k
return min	C_2	1

The heap_extract_min adds a constant amount of work to the runtime of min_heapify which is $\Theta(h)$ where h is the height of the tree given by $\lg n$. Every time an element is removed from the heap the size decreases by one. Therefore the total runtime for the inner line of the for loop is given by the sum:

$$\begin{aligned}\sum_{i=1}^{k} \lg(n-i) &\leq \sum_{i=1}^{k} \lg n \\ &= \lg n + \lg n + \ldots + \lg n \\ &= k \lg n\end{aligned}$$

Therefore line 3 will be bounded by $\mathcal{O}(k \lg n)$.

Therefore the total runtime for the algorithm will be:

$$\begin{aligned}\Theta(n) + C_1(k+1) + \mathcal{O}(k \lg n) + C_2 &= \Theta(n) + kC_1 + C_1 + \mathcal{O}(k \lg n) + C_2 \\ &= \mathcal{O}(n + k \lg n)\end{aligned}$$

Question 3

The algorithm I developed to report the keys smaller than *key* is below.

```
smaller_than(A, key, i=1)
left = left(i)
right = right(i)
rtrn = []
if A[i] <= key:
    rtrn += [A[i]]
if left <= n and A[left] <= key:
    rtrn += self.smaller_than(A, key, left)
if right <= n and A[right] <= key:
    rtrn += self.smaller_than(A, key, right)
return rtrn
```

Assume that the += operator adds the two lists together in constant time.

This algorithm is somewhat similar to max_heapify. This algorithm takes a minimum heap and the termination key as input. The algorithm starts on the root node of the min-heap. If the current node is less than or equal to *key*, it is added to the list. If the left child is less than *key*, the algorithm recurses on the left child. If the right child is less than *key*, the algorithm recurses on the right child. We know from the properties of min-heaps that if a root element of the heap is larger than *key*, then all sub-nodes in the heap must also be larger than the root, which forces the remaining elements in the heap to be larger than *key*. Therefore the algorithm will only visit the nodes in the heap that are less than *key* giving a runtime of $\Theta(k)$ where k is the number of elements returned by the algorithm.

Question 4

a.

	Max Stack Depth	Running time
A is sorted	$\Theta(n)$	$\Theta(n^2)$
A with all elements equal	$\Theta(n)$	$\Theta(n^2)$
A is reversely sorted	$\Theta(n)$	$\Theta(n^2)$

When A is sorted the pivot will always be the maximum element in the partition. The algorithm will first push the right partition with length 0, then push the left partition with a length one less than the current length. The while loop will then continue with the left partition. Therefore each iteration of the while loop where $p < q$ will add a new tuple to the stack, ultimately giving n tuples on the stack. Every call to partition will swap every element in the partition with itself, then swap the pivot with itself. This gives partition a runtime of $\Theta(n)$ and partition will be called n times, giving total runtime $\Theta(n^2)$.

When A has all elements equal, the elements are all in sorted order. The algorithm behaves the same as when all elements were sorted.

When A has element in reverse sorted order the pivot will alternate between being the smallest and largest value in the partition. Starting the first iteration of the while loop where $p = 1$ and $r = n$ partition will swap the last element in the partition to the first position in the partition. The algorithm will then push the valid right partition onto the stack, then will push the invalid left partition onto the stack. The algorithm pops the invalid partition off the stack and discards it. The algorithm then pops off the next valid partition swaps the pivot with itself (because the pivot is the largest element in the partition) and continues the sequence. This case will have slightly more than half the number of tuples on the stack, upper bounded by $\lceil \frac{n+1}{2} \rceil$. The runtime will remain identical to both previous cases, partition will remain with runtime $\Theta(n)$ and will have to be run n times, yielding a runtime of $\Theta(n^2)$.

b.

	Max Stack Depth	Running time
A is sorted	$\Theta(1)$	$\Theta(n^2)$
A with all elements equal	$\Theta(1)$	$\Theta(n^2)$
A is reversely sorted	$\Theta(n)$	$\Theta(n^2)$

Changing the order of pushes does not change the runtime of the algorithm.

When A is sorted, the valid left partition will be pushed first, then the invalid right partition will be pushed. The next iteration of the while loop discards the invalid right partition. The next iteration starts the process over again. There will only be the a maximum of 2 tuples on the stack, one each from the last right and left partition.

When A is reversely sorted the maximum stack depth will be $\lceil \frac{n+2}{2} \rceil = \Theta(n)$.