CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 2
Submit via Gradescope

Name: Xuran Feng
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(Extra Credit)*** *Implement MergeSort without recursive calls.*

Solution:

2. ***(5 points)*** *Exercise 6.1-4, explain why.*

Solution:
The smallest element must reside in any one of leaves, a distinct max-heap maintains property that node element should be greater than all of its children's elements (if it has children); so for any non-leaf node within a max-heap, its children must have smaller values which means this non-leaf parent node can't have the smallest element; however, leaf node doesn't have any children so it doesn't have to worry that there must be smaller values than itself, so the smallest value must reside in one of the leaf nodes.

3. ***(5 points)*** *Exercise 6.1-6, explain why.*

Solution:
No, it's not a max heap because 6 has a child node with value 7 that is greater than 6, this violates max-heap property that node element should be greater than all of its children's elements (if it has children).

4. ***(10 points)*** *Exercise 6.2-1.*

Solution:
First swap 3 with its largest child node which is 10, and we get a new heap array [27,17,10,16,13,3,1,5,7,12,4,8,9,0]; we find 3 still needs to be swapped with its largest child which is 9 because max-heap property is still violated, after second swap we get [27,17,10,16,13,9,1,5,7,12,4,8,3,0], this new heap now maintains max-heap property and we finish the max-heapify of 3.

5. ***(10 points)*** *Exercise 6.3-1.*

Solution:
Because leaf nodes must be valid max-heap, we start a loop from the last non-leaf node and maxheapify this node till we maxheapify the root. The last non-leaf node is 10 and 10 is smaller than its child 22, so we swap 10 with 22 to get new heap array [5,3,17,22,84,19,6,10,9], now we fix the subheap of [10,22,9]; we continue adjusting next non-leaf node 17 and swapping it with its child 19 to get [5,3,19,22,84,17,6,10,9]; next non-leaf node is 3 and we swap it with its greatest child 84 to get [5,84,19,22,3,17,6,10,9]; finally we come to the root node 5 and we swap it with its greatest node 84 to get [84,5,19,22,3,17,6,10,9], however 5 still needs to be swapped with its new greatest child 22 to get [84,22,19,5,3,17,6,10,9], again 5 still needs to be swapped with its new greatest child 10 to get [84,22,19,10,3,17,6,5,9], now our heap is valid and we finish BUILD-MAX-HEAP.

6. ***(15 points)*** *Problem 6-2.*

Solution:
a. Since a d-ray heap is still a full tree, we can find the relationship between parent and child array index to represent this heap. The relationship between parent(with index k) and its first child(with

index i) is $i = k * d - (d - 2) = (k - 1)d + 2$, the second child would have index $(k - 1)d + 2(+1)$, the third child will have index $(k - 1)d + 2(+2)$..., we can infer that the jth child will have index $(k - 1)d + 2 + (j - 1) = (k - 1)d + j + 1$ for $j = 1...d$; for the other way, if we already know a child has index i, we can infer parent index $k = \frac{i-(j+1)}{d} + 1$ for $j = 1...d$, and it can be rearranged as $k = \frac{i-2}{d} + 1 - \frac{j-1}{d}$ =(floor$(1 + \frac{i-2}{d})$), since the last term $\frac{j-1}{d}$ can be truncated.

b. For a d-ray heap with height h, we can know that $d^0 + d^1 + d^2 + ... + c \cdot d^h = n$, the last layer could be $c * d^h$ with c as a constant ≤ 1 because it may be not full. By solving this, we know $h = \theta(log_d n)$.

c. referenced from textbook, for HEAP-EXTRACT-MAX itself, it only takes $\theta(1)$ time to get the max, after getting the max, MAX-HEAPIFY will takes $O(log_d n)$ ($log_d n$ is also height in part b) to adjust and maintain the heap property.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size<1:
    error "heap underflow"
max=A[1]
A[1]=A[A.heap-size]
A.heap-size=A.heap-size - 1
MAX-HEAPIFY(A,1)
return max

MAX-HEAPIFY(A,i)
l=LEFT(i)
r=RIGHT(i)
if(l≤A.heap-size and A[l]>A[i])
    largest=l
else largest=i
if(l≤A.heap-size and A[l]>A[largest])
    largest=r
if largest≠i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

d.referenced from textbook, the running time for MAX-HEAP-INSERT itself is $\theta(1)$, but it still needs $O(log_d n)$ ($log_d n$ is also height in part b) to call HEAP-INCREASE-KEY(part e) because the element needs to float up at most the d-ray tree height to locate its final position.

```
MAX-HEAP-INSERT(A,key)
A.heap-size = A.heap-size+1
A[A.heap-size]=−∞
HEAP-INCREASE-KEY(A,A.heap-size,key)
```

e. referenced from textbook, the running time for this is still $O(log_d n)$ ($log_d n$ is also height in part b) because the element needs to float up at most the d-ray tree height to locate its final position.

```
HEAP-INCREASE-KEY(A,i,key)
if key<A[i]
    error "new key is smaller than current key"
A[i]=key
while i>1 and A[Parent(i)<A[i]]
    exchange A[i] with A[Parent(i)]
    i=Parent(i)
```

7. ***(5 points)*** *Exercise 7.2-1.*

Solution:

$T(n) = T(n-1)+\theta(n) = T(n-2)+\theta(n-1)+\theta(n) = T(n-k)+\theta(n-(k-1))+\theta(n-(k-2))+...+\theta(n)$, the recursion stops when $k = n-1$, so $T(n) = T(1)+\sum_{j=0}^{n-1}\theta(n-j) = T(1)+\sum_{j=1}^{n}\theta(j)$, by using natural sum series formula, we know $T(n) = T(1)+\theta(\frac{(1+n)n}{2})$, which has a time complexity $\theta(n^2)$.

8. ***(5 points)*** *Exercise 7.2-2, explain why.*

Solution:

The running time is $\theta(n^2)$. If all elements have same value, the outer $[1:n]$ loop always swaps an element with the last position because all the other elements before last position(pivot) need to be less or equal than this element. The recurrence formula is just $T(n) = T(0)+T(n-1)+\theta(n)$ which has a time complexity $\theta(n^2)$ from Exercise 7.2-1.

9. ***(5 points)*** *Exercise 7.2-3.*

Solution:

If the whole array is sorted in descending order, the outer [1:n] loop always moves an element to an end position(first index or second index of that subarray). For example, [9,8,7,6,5,4,3,2,1] will be [1,8,7,6,5,4,3,2,9] after first swap, now the pivot position is the last index and we do a quick sort of first eight elements [1,8,7,6,5,4,3,2]; although this subarray now is not a strictly descending sequence, the first element 1 still needs to be compared with any other element to find its position, and turns out its position is exactly the first index; we continue doing quick sort of last seven elements [8,7,6,5,4,3,2] and repeat the previous two processes again and again till the base case. So we can know the recurrence formular is $T(n) = T(n-1)+T(0)+\theta(n)$ and from Exercise 7.2-1 we know the time complexity is $\theta(n^2)$.

10. ***(15 points)*** *Exercise 7.4-1.*

Solution:

We can prove this by using induction method.Our induction hypothesis is $T(q) \geq cq^2$ and $T(n-q-1) \geq c(n-q-1)^2$ for $0 \leq q \leq n-1$, so $T(n) = \max_{0\leq q\leq n-1}(T(q)+T(n-q-1))+\theta(n) \geq c\cdot\max_{0\leq q\leq n-1}[q^2+(n-q-1)^2]+\theta(n) \geq c\cdot(n^2-2n+1)+\theta(n)$ according to Exercise 7.4-3 conclusion. If we want to make $c\cdot(n^2-2n+1)+\theta(n) \geq c\cdot n^2$ true, then $c \leq \frac{n}{2n-1}$ must be valid, we divide n and get $c \leq \frac{1}{2-\frac{1}{n}}$, since when n is large enough $\frac{1}{2-\frac{1}{n}}$ will converge to $\frac{1}{2}$ from infinity, $c \leq \frac{1}{2}$ is always valid, we can find a constant c and finish our proof.

11. ***(15 points)*** *Exercise 7.4-2.*

Solution:

The best case of quicksort is $T(n) = \min_{0 \le q \le n-1}(T(q)+T(n-q-1))+\theta(n)$. Our induction hypothesis is $T(q) \ge c \cdot qlgq$ and $T(n-q-1) \ge c \cdot (n-q-1)lg(n-q-1)$ for $0 \le q \le n-1$, so $T(n) \ge c \cdot min_{0 \le q \le n-1}[qlgq + (n-q-1)lg(n-q-1)]+\theta(n)$, by takeing derivative of $qlgq+(n-q-1)lg(n-q-1)$ with $0 \le q \le n-1$, we can compute the minimum value is when $q = \frac{n-1}{2}$, minimum is $(n-1)lg\frac{n-1}{2}$, now we want to make $c \cdot (n-1)\lg\frac{n-1}{2}+\theta(n) \ge c \cdot nlgn$ true, after rearranging, we get $c \cdot nlg\frac{n-1}{n} - c\lg(n-1) - c \cdot lg2 \cdot (n-1)+n \ge 0$, divide n on both side, $c \cdot lg\frac{n-1}{n} - c \cdot \frac{\lg(n-1)}{n} - c \cdot lg2 \cdot \frac{(n-1)}{n} + 1 \ge 0$, the first term and second term will converge to 0 when n is large enough, the third term will converge to $c \cdot lg2$, so now we can find a constant $c \le \frac{1}{lg2}$ to finish the proof.

12. ***(10 points)*** *Exercise 7.4-3.*

Solution:

$q^2+(n-q-1)^2 = 2q^2+(2-2n)q+n^2-2n+1$, we take first derivative of q to get $4q+2-2n$, by solving $4q+2-2n=0$ we get $q = \frac{n-1}{2}$, because the original polynomial function has a positive parameter 2 of term $2q^2$, so the parabola opens up and after plugging in the q to the original function meaning, we get a minimum value of this parabola, because q has to fall in $[0, n-1]$, then only when $q = 0$ or $q = n-1$ the function may reach its maximum value, when $q = 0, 2q^2+(2-2n)q+n^2-2n+1 = n^2-2n+1$, when $q = n-1, 2q^2+(2-2n)q+n^2-2n+1 = n^2-2n+1$, so the maximum of original function is n^2-2n+1 when $q = 0$ or $q = n-1$.

13. ***(Extra credit)*** *Problem 6-3.*

Solution: