# Solutions for HW2 - CS 6033 Fall 2021



## Q1 → Understanding Heap operations

1. For the following array

| 40 | 6 | 3 | 11 | 2 | 4 |
|---|---|---|---|---|---|

   (a) create a max heap using the algorithm we discussed in class (BUILD-MAX-HEAP)

   (b) remove the largest item from the max heap you created in 1a, using the HEAP-EXTRACT-MAX function from the book. Show the array after you have removed the largest item

   (c) using the algorithm from the book, MAX-HEAP-INSERT, insert 56 into the heap that resulted from question 1b. Show the array after you have inserted the item.

(a) [40, 11, 4, 6, 2, 3]

(b) [11, 6, 4, 3, 2]

(c) [56, 6, 11, 3, 2, 4]

# Q2 → Heap variation

2. Consider of 3-ary max-heap. A 3-ary max-heap is like a binary max-heap, but instead of 2 children, nodes have 4 children.

   (a) How would you represent a 3-ary max-heap in a array?

   (b) What is the height of a 3-ary max-heap of $n$ elements in terms of $n$ and 3?

   (c) Give an efficient implementation of `HEAP-EXTRACT-MAX`. Analyze its running time in terms of 3 and $n$.

   (d) Give an efficient implementation of `MAX-HEAP-INSERT`. Analyze its running time in terms of 3 and $n$.

   (e) Give an efficient implementation of `HEAP-INCREASE-KEY`$(A, i, k)$. Analyze its running time in terms of 3 and $n$.

(a)

3-ary max-heap in a array:

3-ary: Parent(i)

return $\lfloor (i+1)/3 \rfloor$

3-ary: Children(i,j) ( i refer to parent, j refer to $j^{th}$ element of parent i, and j>=1 and j<=3)

return 3i-2+j

Second notation: Left(i)=3i-1, Middle(i)=3i, Right(i)=3i+1

(b)

We denote the height of the 3-ary max-heap as h: we can have max nodes n<=(3^(h+1)-1)/2

For the min nodes we can have:n>=1+(3^h-1)/2

Now we can have (3^(h+1)-1)/2 ≥ n ≥ 1+(3^h-1)/2, so we can have $\log_3(2n-1) \geq h \geq \log_3(2n+1)-1$. Therefore h is $\Theta(\log_3 n)$

(c)

We can implement the HEAP-EXTRACT-MAX(A) from textbook. First, we retrieve the Max root, then we move the last element to the root, then we implement MAX-HEAPIFY(A). One thing we need to be careful here, since now every parent has 3 children, during the procedure of MAX-HEAPIFY(A), 3 children and 1 parent have to be compared to get the largest. (at most 3 comparisons are needed). The running time of HEAP-EXTRACT-MAX is dominated by MAX-HEAPIFY(A). And during the MAX-HEAPIFY(A) we know the height of the 3-ary max-heap is $\Theta(\log_3 n)$, at most 3 comparisons is needed for each level. Therefore the worst-case running time for HEAP-EXTRACT-MAX is $O(3*\log_3 n)$, $O(\log_3 n)$ is also acceptable. The following is the code implementation:

```
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)
    largest = i
   for j = 1 to 3
        child = CHILDREN(i,j)
        if child <= A.heap_size and A[child] > largest
           largest = child
    if largest != i:
       exchange A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
```

(d)

We can take the MAX-HEAP-INSERT from the book to insert a new element. The procedure first expands the max-heap by adding to the tree a new leaf whose key is -∞. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\log_3 n)$. The worst-case scenario is the key might have to traverse to the root and the length is its height.

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

(e)

We can also implement HEAP-INCREASE-KEY from the book. The worst-case scenario is the key might have traverse from the bottom to the root and the length is its height. The running time of HEAP-INCREASE-KEY on an n-element heap is $O(\log_3 n)$.

```
HEAP-INCREASE-KEY.(A, i, key)
1 if key < A[i]
2      error "new key is smaller than current key"
3 A[i]=key
4 while i > 1 and A[PARENT(i)]< A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

# Q3 → Prove the leaves' number in a heap

3. Show that the number of leaves in a heap is $\lceil \frac{n}{2} \rceil$. (See if you can find a simple answer based on the lecture.)

Answer: In a heap, the last leaf node is always the child of the last parent, i.e., the parent node with the highest index. Every node following this last parent node is a leaf node. So if we find the index of this last parent and subtract it from the total number of nodes, we can find the total number of leaf nodes in a heap. We can find the parent index of a node $i$ by $parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$. Similarly, the last parent is the parent of the last node, and in a heap with $n$ nodes, it is $\left\lfloor \frac{n}{2} \right\rfloor$. So the number of leaf nodes in any heap is $n - \left\lfloor \frac{n}{2} \right\rfloor$ which is $\left\lceil \frac{n}{2} \right\rceil$.

# Q4 → Using loop invariant to prove HEAP-DECREASE-KEY

4. Prove the correctness of `HEAP-DECREASE-KEY` using a loop invariant.

Answer: Loop invariant: At the start of each iteration of the *while* loop indexed by $i$, the node $i$ is the root of its MIN-HEAP without any changes in the position of the nodes outside this mini MIN-HEAP.

$HEAP - DECREASE - KEY(A, i, key)$
    **if** $key > A[i]$
        **error** "$new\ key\ is\ larger$"
    $A[i] = key$
    **while** $i > 1$ **and** $A[PARENT(i)] > A[i]$
        **exchange** $A[i]$ **with** $A[PARENT(i)]$
        $i = PARENT(i)$

We go through 3 steps to show that the loop invariant proves the algorithm correct; the initialization, maintenance, and termination.

Initialization: Before the loop begins, we have $A[i] = key$. Because key has replaced a value larger than itself, we know that the order of MIN-HEAP is still maintained and that the children of $i$ are still larger than key. Hence a mini MIN-HEAP is formed with $i$ as the root, and the rest of the structure is unaffected.

Maintenance: The purpose of the loop is to keep traversing the heap in a bottom-up manner, i.e., from leaf to root, and in the process, exchange smaller children with their parents, thus preserving heap structure. In every iteration, we see that after the child is exchanged with the parent, and $i$ is updated to the new parent, the mini MIN-HEAP is maintained.

Termination: The loop terminates if one of two things happens. Either $i$ has reached the root, in which case the smallest element has been pushed to the top while maintaining heap structure in the children nodes. This means that the MIN-HEAP has formed with the newly decreased value. The other instance where the loop stops is

when the parent's value is lesser than the child's value, which is also a case where MIN-HEAP is formed as $i$ with its own mini MIN-HEAP and the rest of the unaffected structure together make a new MIN-HEAP.

# Q5 → Recurrence

5. For the following algorithm:

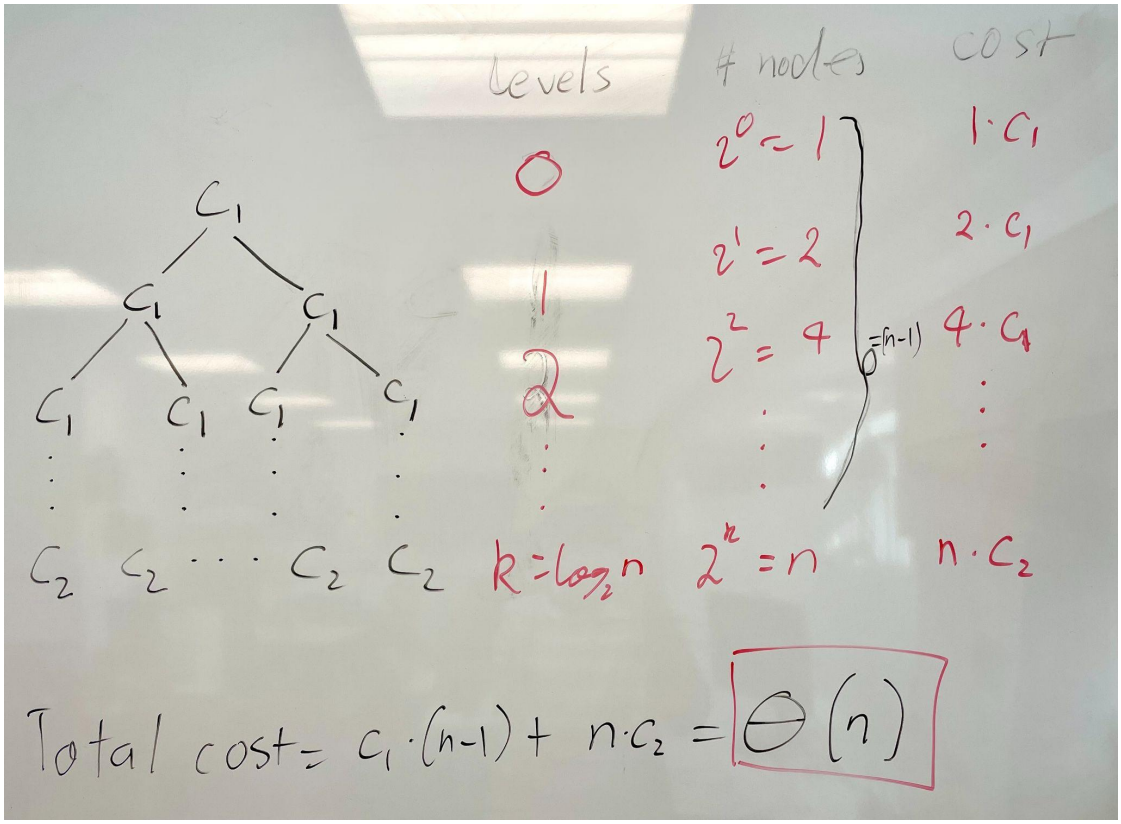
   (a) Write the recurrence formula for the following function. Each node should hold the cost of that recursive call when you don't consider the cost to compute the recursive subproblems.[1]

   (b) Draw the recursion tree for the following function and show the running time for each level. In your recursion tree, show the top 3 levels of the recursion tree and the bottom level of the recursion tree. Put "..." (dots) to represent the levels you didn't show.

```
MINIMUM(A, l, r)
1) if (r − l == 0)
2)     return A[r]
3)
4) lmin = MINIMUM(A, l, ⌊(l + r)/2⌋)
5) rmin = MINIMUM(A, ⌊(l + r)/2⌋ + 1, r)
6) PRINT(rmin, lmin)
7) if rmin < lmin
8)     return lmin
9) else
10)    return rmin
```

(a) $T(n) = c_2 \; if \; n = 1 \; else \; 2T(n/2) + c_1$

(b)




# Q6 → Heap application - Shortest Job First

6. Now that your a graduate student, you are bombarded with things to do! You have to make sure you play FFXIV, re-clear Skyrim for the 300th time, and do your homework assignment for CS6033. Of course, as you are trying to finish all your tasks a new task might arise (your roommate might hit you up for a game of League, or you might remember you promised to call your parents last weekend) and you have to make sure it gets done.

   You decide to keep working until all the tasks are done because you want to enjoy your weekend.

   In a attempt to build momentum, you decide to complete the tasks in order of the time you estimated they would take.

   Devise an algorithm that will tell you the next task to do in $O(\log n)$ time, as well as integrate new tasks in $O(\log n)$.

   Justify the running time of your algorithm.

This is similar to the Shortest Job First algorithm. We use a min-heap to store all the tasks and use the time to complete the task as the key. Whenever a new task comes up, we can insert it into the heap in O(log n). Whenever we need a new task to do, we can pop it out of the heap in O(log n).

```
tasks = [] // This is a min-heap
NEXT_TASK():
```

```
        if tasks.size > 0:
                next_task = HEAP-EXTRACT-MIN(tasks) // This takes O(log n)
                print(next_task)
ADD_TASK(new_task):
        MIN-HEAP-INSERT(tasks, new_task)  // This takes O(log n)
```

# Q7 → Heap application - Buy stocks

7. Last week when you asked for advice, you didn't think you would get so many suggestions. Due to not wanting to lose most of your money in broker fees,[2] you don't want to invest less than 100$ per stock, so you decide to buy the $B < N$ stocks that did the best in the past 6 months time. Since the advice keeps flowing in and you decide to invest your money next Monday, you want to quickly be able to determine the best stocks to buy in $O(B)$ time and insert a new stock suggestion in $O(\log N)$ time. Partial credit for determining the top $B$ stocks in $O(B \log N)$ time and inserting a new stock in $O(\log N)$ time.[3]

   For this question, you may assume the stock names you have already removed any duplicate names.

For this question we will accept both O(B) and O(B log N) solutions for picking top B stocks with full credits as the professor believes that the question is not clear enough. Here is an O(B) picking and O(log N) inserting solution using a min heap.

**Q 7.** Create a min-heap to keep B best performance stocks. When inserting, just insert the value to the min-heap if heap size is smaller than B. Otherwise, we compare the value with the root value. If the root value is smaller, replace it with the newly given value and call MIN-HEAPIFY. In this way, the min-heap always contains B best stocks. To determine what stocks to buy, we need to return all the values in the min-heap.

```
Q7-INSERT(A, key)
1: if A.heap-size< B then
2:    MIN-HEAP-INSERT(A, key)
3: else
4:    if A[1] < key then
5:       A[1] = key
6:       MIN-HEAPIFY(A, 1)
Q7-STOCKS-TO-BUY(A)
1: for i = 1 to A.heap-size
2:    print(A[i])
```

The running time of insertion is $O(logN)$. Both MIN-HEAP-INSERT and MIN-HEAPIFY take $O(logB)$ times and $O(logB) < O(logN)$ because $B < N$. The running time of getting stocks to buy is $O(B)$. Under the constraint of $B < N$, heap size is equal to B. Therefore, the for loop need $O(B)$ times to finish.

(Contributors: Atsushi Shimizu, Ge Chang)

# Q8 → Heap application - Find best stocks

8. Just out of curiosity, you would like to know the average return of the top 1/4 stock picks every week (based on how well they performed in the past 6 months as of last Friday). You are impatient and want the answer in $O(1)$ time. The time to insert a new stock suggestion must run in $O(\log N)$ time, in order to keep the costs of recomputing the existing picks in the system over the weekend and the costs of inserting new entries during the week low.

   For this question, you may assume the stock names you have already removed any duplicate names.

For this question, I use an array B to store top 1/4 stocks (min-heap), an array A to store other 3/4 stocks(max-heap), and integer `quartersum` to store the sum of top 1/4 stock. Since the stock suggestion keeps flowing in, the total number of stocks N is growing automatically. When inserting a new stock, we first see if the size of the min heap is N/4. If so, new stock will substitute for B[1] if it's greater than B[1], or new stock will be inserted into array A if it's smaller than B[1]; otherwise, we compare new stock and A[1], the larger one will be inserted into array B and the smaller one will be put in the array A. And the function AVERAGE-TOP-QUARTER is to return the average of the top 1/4 stock picks.

Hence, inserting a new stock runs in O(log N) time, and returning the average of the top 1/4 stock picks runs in O(1) time.

```
AVERAGE-TOP-QUARTER(quartersum)
      int num = N/ 4
      return quartersum / num;
MAX-HEAP-INSERT(A, key)
      A.heap-size = A.heap-size + 1
      int i = A.heap-size
      A[i] = key
      while i>1 and A[PARENT(i)] < A[i]
            exchange A[i]with A[PARENT(i)]
            i = PARENT(i)
MIN-HEAP-INSERT(B, key)
      if (B.heap-size < N/4)
            if(key > A[1])
                  quartersum += key
                  B.heap-size = B.heap-size + 1
                  int i = B.heap-size
                  B[i] = key
                  while i>1 and B[PARENT(i)] > B[i]
                        exchange B[i] with B[PARENT(i)]
                        i = PARENT(i)
      else
            quartersum += A[1]
            B.heap-size = B.heap-size + 1
            int j = B.heap-size
            B[i] = A[1]
            while i>1 and B[PARENT(i)] > B[i]
```

```
            exchange B[i] with B[PARENT(i)]
            i = PARENT(i)
      B[1]=key
      MAX-HEAPIFY(B,1)
else if( B.heap-size == N / 4)
      if(key > B[1])
            quartersum = quartersum + key - B[1]
            MAX-HEAP-INSERT(A, B[1])
            B[1] = key
            MIN-HEAPIFY(B, 1)
      else
            MAX-HEAP-INSERT(A, key)
```

(Contributor: Xinyu Zhang)