Chad Hirsch

CS4310 Assignment 2

1. Given an array A of n integers, find the longest subarray of A such that all the numbers in that subarray are in sorted order. Give your pseudo code. What is the running time of your method?

```
Longest_Subarray(array):
    i = 0
    j = 0

    start_longest = 0
    end_longest = 0

    start_curr = 0
    end_curr = 0

    longest  = 1
    curr = 1

    while i < array.length and j < array.length
        if array[i] < array[j]
            curr += 1
            end_curr += 1
            if curr > longest
                longest = curr
                start_longest = start_curr
                end_longest = end_curr
        else
            curr = 1
            start_curr = j
            end_curr = j
        i += 1
        j += 1

    return array[start_longest : end_longest]
```

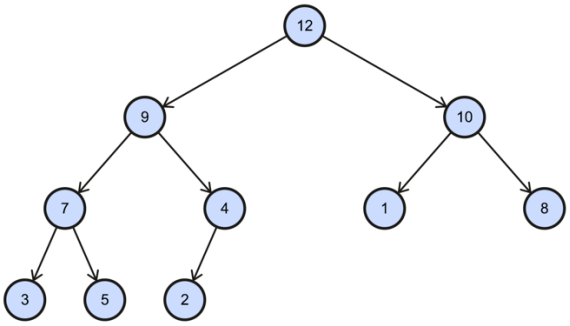
This algorithm runs in O(n) time since we only traverse the array once.

Chad Hirsch

2. (10 pts) What are the minimum and maximum number of elements in a heap of height h?

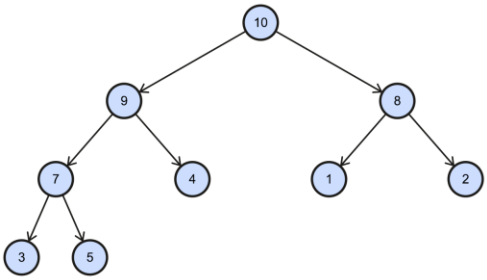
At most there are $2^{h+1} - 1$ elements in a heap of height h because in this case all levels of the heap are filled. Each level of the heap has 2^i elements, where i is the current level. Therefore, $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$. The minimum number of elements are 2^h elements in a heap of height h. This is because there are only h-1 full levels in the heap, and the last level contains only 1 element. Since there are h-1 full levels then there are $2^{(h-1)+1} - 1 = 2^h - 1$ elements plus the 1 element on the last level, so we get that there are $(2^h - 1) + 1 = 2^h$ elements.

3. (15 pts) Show the max-heap that results from building the heap directly on the following integer values in an array: 10, 5, 12, 3, 2, 1, 8, 7, 9, 4.



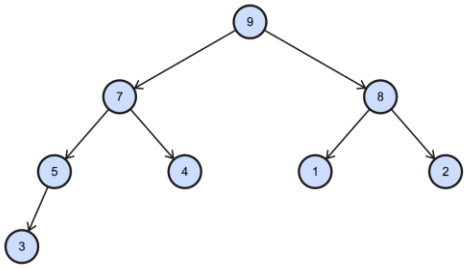
12, 9, 10, 7, 4, 1, 8, 3, 5, 2

4. (15 pts) Show how heapsort works in-place using the max-heap built in the previous question to sort the values in ascending order.

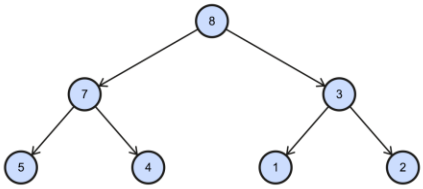


10, 9, 8, 7, 4, 1, 2, 3, 5, 12

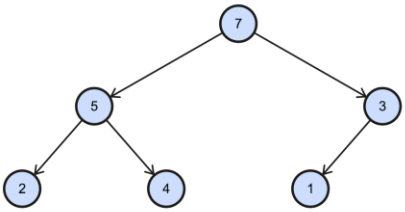
Chad Hirsch



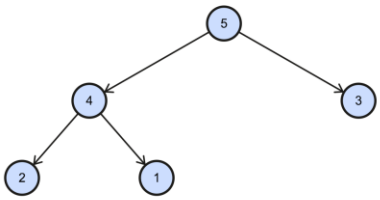
9, 7, 8, 5, 4, 1, 2, 3, 10, 12



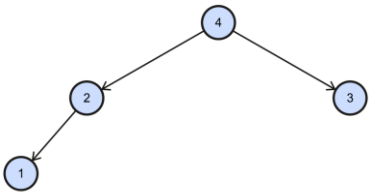
8, 7, 3, 5, 4, 1, 2, 9, 10, 12



7, 5, 3, 2, 4, 1, 8, 9, 10, 12

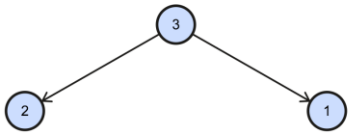


5, 4, 3, 2, 1, 7, 8, 9, 10, 12

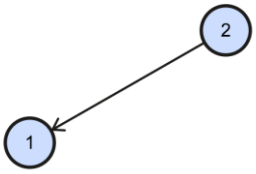


4, 2, 3, 1, 5, 7, 8, 9, 10, 12

Chad Hirsch



3, 2, 1, 4, 5, 7, 8, 9, 10, 12



2, 1, 3, 4, 5, 7, 8, 9, 10, 12

1, 2, 3, 4, 5, 7, 8, 9, 10, 12

5. (20pts) Assuming that the pivot is always the 2nd element of a list, simulate the execution of Quicksort on the following input: 32, 23, 18, 27, 8, 30, 98, 14, 45, 0, 99, 47, 43, 23, 123, 75, 6, 19, 85.

Pivot, In correct position

32, 23, 18, 27, 8, 30, 98, 14, 45, 0, 99, 47, 43, 23, 123, 75, 6, 19, 85

18, 8, 14, 0, 6, 19, 23, 32, 27, 30, 98, 45, 99, 47, 43, 23, 123, 75, 85

0, 6, 8, 18, 14, 19, 23, 23, 27, 32, 30, 98, 45, 99, 47, 43, 123, 75, 85

0, 6, 8, 14, 18, 19, 23, 23, 27, 30, 32, 98, 45, 99, 47, 43, 123, 75, 85

0, 6, 8, 14, 18, 19, 23, 23, 27, 30, 32, 45, 47, 43, 75, 85, 98, 99, 123

0, 6, 8, 14, 18, 19, 23, 23, 27, 30, 32, 43, 45, 47, 75, 85, 98, 99, 123

0, 6, 8, 14, 18, 19, 23, 23, 27, 30, 32, 43, 45, 47, 75, 85, 98, 99, 123

0, 6, 8, 14, 18, 19, 23, 23, 27, 30, 32, 43, 45, 47, 75, 85, 98, 99, 123

6. (20pts) Consider the following extension to binary search on a sorted array A for an item x: Let A[p] and A[q] be the partitioning elements of A so that A is divided into three roughly equal-sized lists (i.e., a difference of 1 or 2 is allowed between the sizes of these sublists but no more). Based on A[p] and A[q], decide the sublist where x may exist. Repeat this process until the sublist is small enough that the answer can be directly

obtained.

• Write pseudo-code for the above TernarySearch method, specifying all the

parameters correctly.

TernarySearch(array, target):

```
if array.length < 1
    return -1
else if array.length <= 3
    for i =0 to array.length -1
        if array[i] == target
            return i

p = array.length / 3
q = array.length * 2 / 3

if target == array[p]
    return p
else if target == array[q]
    return q
else if target < array[p]
    return TernarySearch(array[0: p-1], target)
else if target > array[p] and target < array[q]
    return 1 + p + TernarySearch(array[p+1 : q-1 ], target)
else if target > array[q]
    return 1 + q + TernarySearch(array[q+1 : array.length -1] , target)
return -1
```

• Derive the recurrence relation for the running time of the TernarySearch algorithm.

$T(n) = T(n/3) + O(1)$

We recursively call TernanrySearch on 1/3 of our array, and on each call Ternary Search does a constant amount of operations

• Solve the recurrence relation, find a close-bound for the running time of

TernarySearch and then express this close-bound using asymptotic notation. Justify

your answer.

$O(\log n)$

Proof By Induction:

Chad Hirsch

Inductive Hypothesis: T(n) < c*log(n), let c =100
Base Case:
T(2) = 2 < 100 log(2)
Inductive Step:
Assume T(x) < 100 log(x), for x < n
T(n) <= T(n/3) + O(1)
< (100 log(n/3)) + O(1)
=100 log(n) + O(1) – 100 log(3) < 100 log(n)

• Derive an expression and its asymptotic bound for the space complexity of

TernarySearch.

O(log n)

S(n) = $\sum_{i=0}^{log_3 n} O(1) = O(log_3 n) = O(\log n)$

Each level takes O(1) space and there are $log_3 n$ levels so the space complexity is O(log n)