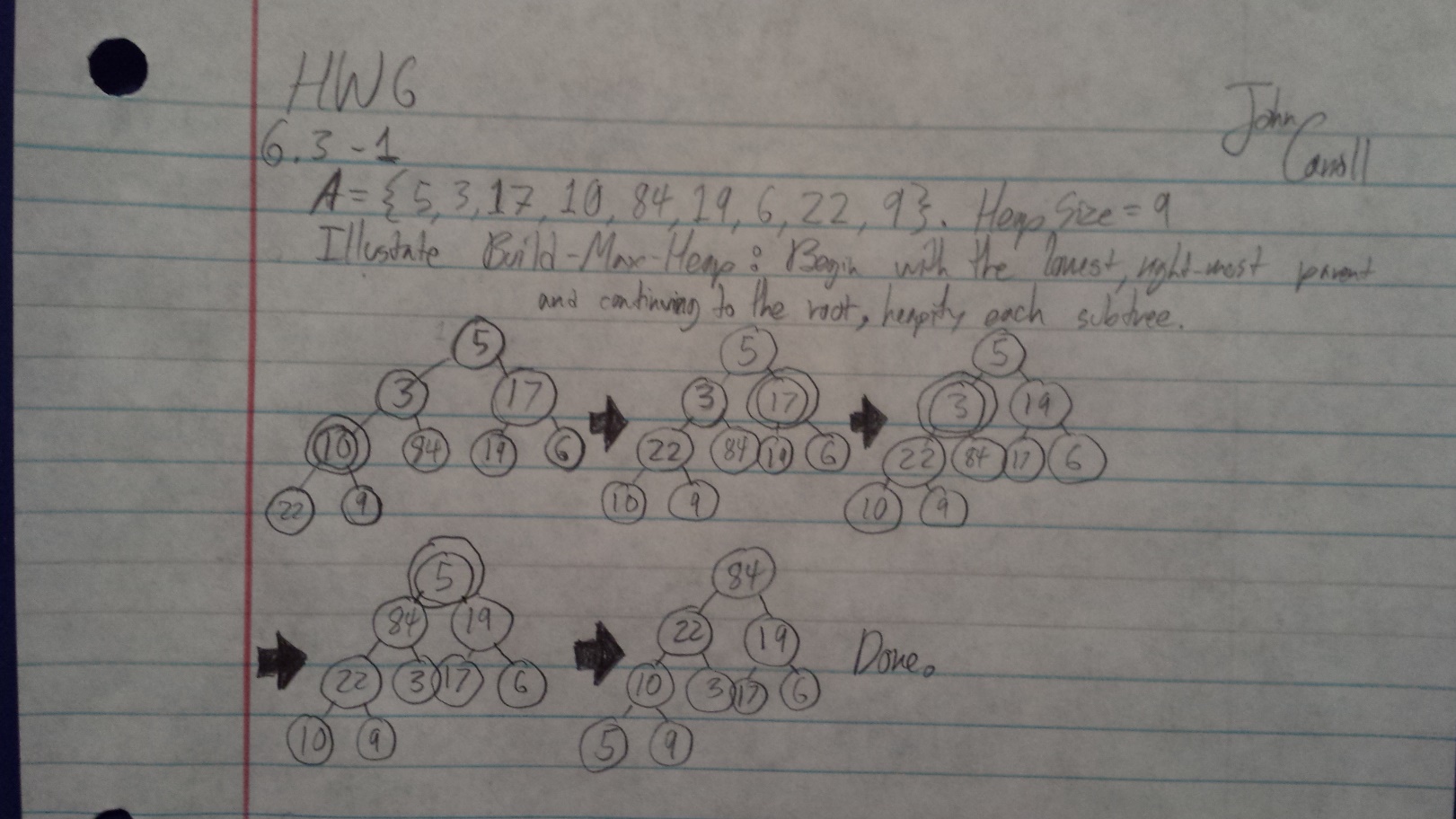
Homework 3

COMP 3270

Problems: **6.3-1, 6.3-2, 6.5-8, 7.1-3, 7.4-2, 8.1-1, 8.3-2, 7-1(a)** at the end of chapter 7.

**6.3-1** [Scanned image of hand-work]



**6.3-2**

We want it to decrease instead of increase because there can only be one maximum, with the exception of duplicates. Also, because it is accurate. If it increased, it would heapify from the top/root downwards. The root would check its two children, it could satisfy max-heapify here, but it cannot be guaranteed that it will satisfy the subtrees. If there is a bigger node lower than the root’s two children, then the max will definitely not be the root. The root *should* become that bigger node further down, of which can no longer be assessed and moved upwards due to the iteration method. There is no guarantee that the root’s two children are roots of max-heaps. This is why decreasing is used instead.

**6.5-8**

Heap-delete(A, i) {

heap-size[A] = heap-size[A] - 1;

while ( 2 \* i < heap-size[A]) {

if (A[ 2 \* i ] > A[ 2 \* I + 1 ]) {

A[i] = A[ 2 \* I ];

i= 2 \* i;

}

else {

A[i]=A[2\*i+1];

i=2\*i+1;

}

}

}

**7.1-3**

The running time of PARTITION on a sub-array of size **n** is **Θ(n)**.

The size of the sub-array being assessed by PARTITION is given by:

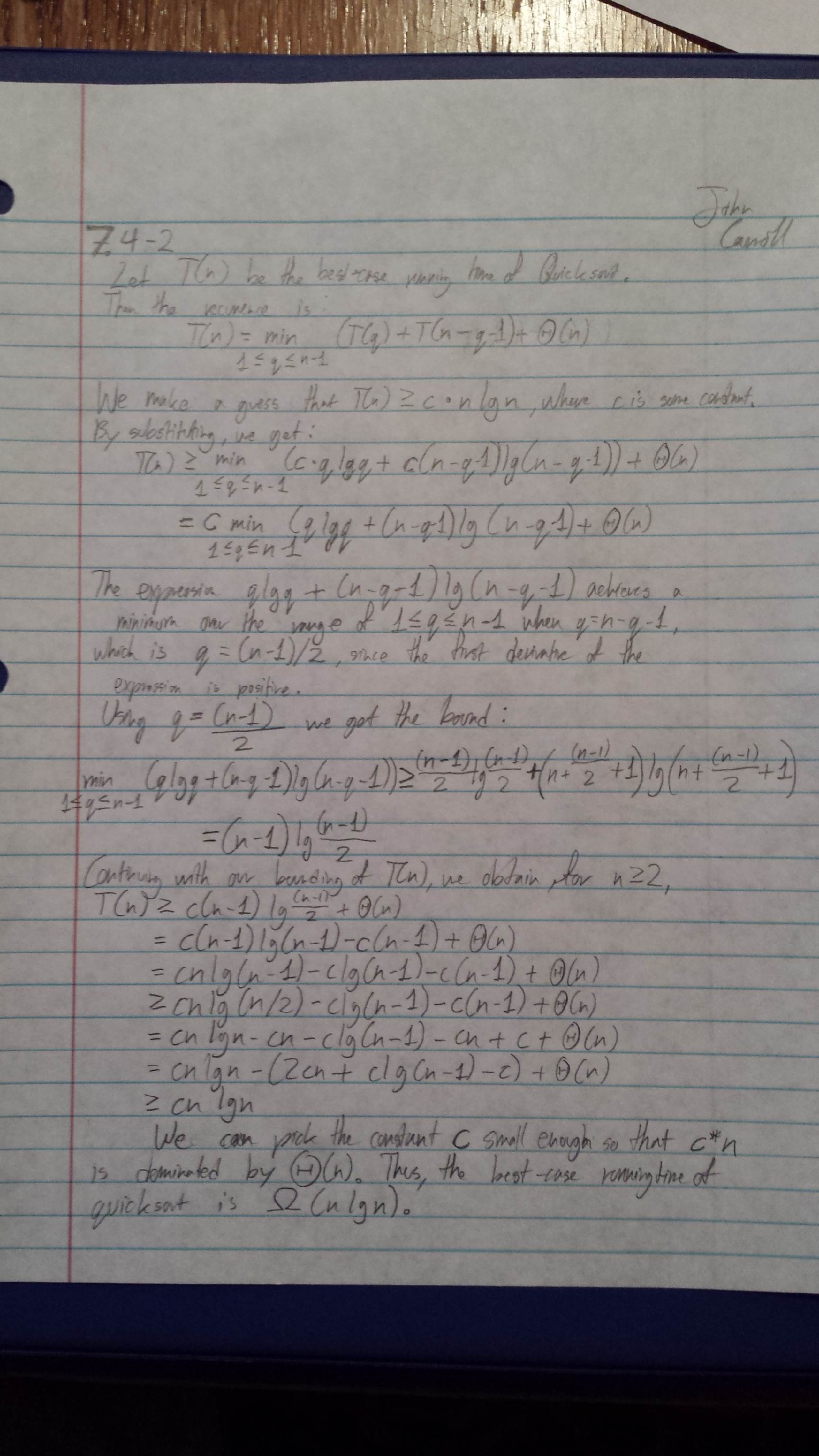
**n = r - p + 1**, where **p** is the first index in the array, and **r** is the last index

Inside of the for-loop, **i** and **j** assess each index of the sub-array **n** times. **i** moves from left to right and **j** moves from right to left. The total number of moves they make inside of the for-loop is **n**, since they go the length of the array.

The running time of everything inside of PARTITION is constant.

Thus, the running time of PARTITION on a sub-array of size **n** is **Θ(n)**.

**7.4-2** [Scanned image of hand-work]



**8.1-1**

The absolute best-case happens when every element of an array of size **n** has been compared with and the data has already been sorted. From the best-case, comparison sort will result in **n - 1** comparisons; and thus, the leaf will have a depth of **n - 1.**

**8.3-2**

Stable: insertion sort, merge sort

Not Stable: heapsort, quicksort

To make any sorting algorithm stable, the array can be placed into a map or Key-Value structured collection – pairs. The first element, K (key), in each pair is the original element and the second, V (value), is its index. Sort lexicographically for this scheme.

This scheme takes additional Θ(n) space and same for time.

**7-1(a) Problem**

**\*Note**: An array generally goes from 0….n -1, and in pseudocode it may be perceived that an array goes from 1….n. Either way, it is but a unit of measurement and I and J will be off by 1.

Initial conditions

A = {13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21}

Size\_of\_A = 12

p = 0, r = 11

Beginning, prior to entering loop, before iteration 1

|  |  |  |
| --- | --- | --- |
| x | i | j |
| 13 | -1 | 12 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 2 | 6 | 21 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

After Iteration 1

|  |  |  |
| --- | --- | --- |
| x | i | j |
| 13 | 0 | 10 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 19 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 2 | 13 | 21 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

After Iteration 2

|  |  |  |
| --- | --- | --- |
| x | i | j |
| 13 | 1 | 9 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 2 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 19 | 13 | 21 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

After Iteration 3

|  |  |  |
| --- | --- | --- |
| x | i | j |
| 13 | 9 | 8 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 2 | 9 | 5 | 12 | 8 | 7 | 4 | 11 | 19 | 13 | 21 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |