Assignment 3: Sorting: Putting your affairs in order

By: Derfel Terciano

Background

In this assignment, we are comparing and implementing the types of sorting algorithms. More specifically, we will be comparing and implementing *Insertion Sort*, *Shell Sort*, *Heap Sort*, and a recursive *Quick Sort*. Once implemented, we also have to implement a test harness that will show us how effective each sorting algorithm is.

Our test harness should report the following:

- The type sort that is being tested
- The number of elements being sorted
- The number of moves the algorithm used to sort an array
- The number of times the algorithm compared two elements.

Lastly, the test harness should print out the sorted array.

Insertion Sort

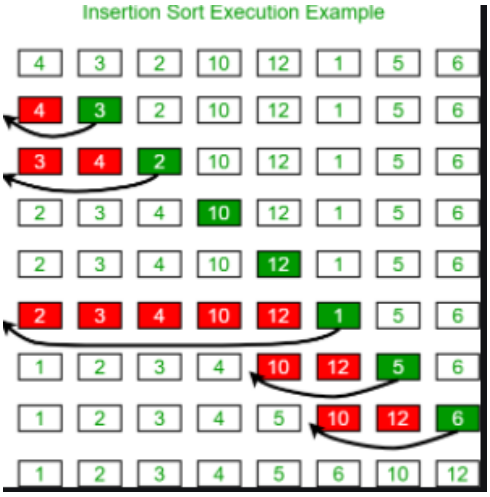
Insertion sort is pretty straightforward however, it is pretty inefficient. With insertion sort, you go through each element and then you compare it with each previous element until it finds the correct order or position. If we are looking at an unsorted array *A*, then we can look at *A*[k] and determine if:

1) *A*[k] is in the right position which means that it is bigger than *A*[k-1] **OR**

2) *A*[k] is in the wrong position which means that *A*[k-1] is bigger than *A*[k].

At this point, *A*[k-1] would be moved to where *A*[k] currently is and then *A*[k] would be compared with *A*[k-2] and so on until the element is in a sorted order. (This means it will keep moving back until *A*[k] is bigger than the previous element it's comparing.)

The following image from GeeksForGeeks shows us how the insertion sort works:



(NOTE: This graph has made me learn alot about how the algorithm works.)

The following python pseudo code (taken front he assignment pdf) can be seen here:

Insertion Sort in Python

```
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

Shell Sort

Shell Sort is like insertion sort except instead of going through each element and comparing that to the previous element, you are comparing two pairs of elements that are some distance apart. The distance between the two elements is called a gap. Each iteration of the sort will decrease the gap size by one and will keep decreasing the gap size until the gap size is 1. When the gap sequence is 1, then the array will be sorted when it reaches the end.

The size of the gap sequence is considered to be a balancing act. If the gap size is either too big or too slow, then this will inhibit the performance of the sorting algorithm. Thus, it may become very slow. With that being said, this assignment will use Knuth's gap sequence that has a complexity of $O(n^{\frac{3}{2}})$.

The following python pseudocode for Shell Sort can be found in assignments pdf:

Shell Sort in Python

```python
from math import log

def gaps(n: int):
    for i in range(int(log(3 + 2 * n) / log(3)), 0, -1):
        yield (3**i - 1) // 2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap, len(A)):
            j = i
            temp = A[i]
            while j >= gap and temp < A[j - gap]:
                A[j] = A[j - gap]
                j -= gap
            A[j] = temp
```

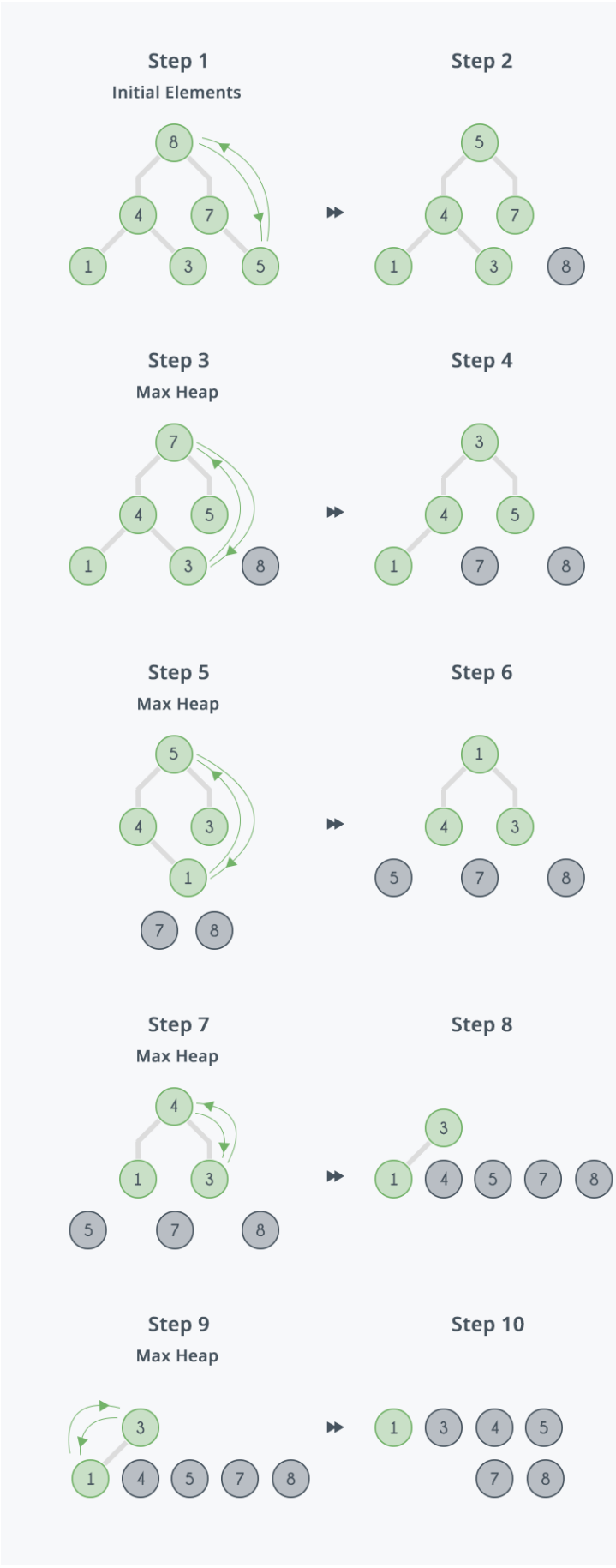
The difficult part here would be trying to figure out how to make an alternative for the yield function.

Heapsort

Heapsort is a kind of sort that utilizes a binary tree representation of the array. These binary trees are called heaps that have nodes which represent each element of the array. There are generally two kinds of heaps: min and max heaps. In this assignment, we will be using a max heap. In a max heap, the parent node usually has the value that is bigger than its children. In the array representation, the parent node is usually placed in the beginning of the array. The children (*2k*) are then placed in the next two spaces and the children's children (*2k+1*) would be placed in the spaces after.

To begin sorting the heaps, you must first *build the heap*. To build the heap, it must obey the rule of the max heap which means the biggest element should be the parent node. Then, you must *fix the heap*. To fix the heap, we must now sort the array. In short, the largest elements (or the elements at the top of the tree) are removed from the top of the heap and placed at the end of the array. Then, you would fix the heap again and repeat the process until the array is sorted.

Below is a diagram from hackerearth.com that shows how heap sort works. This diagram truly made me understand the algorithm.


Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

Below is also the python pseudocode from the assignment pdf that shows us how the algorithm is implemented.

Heap maintenance in Python

```
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

Heapsort in Python

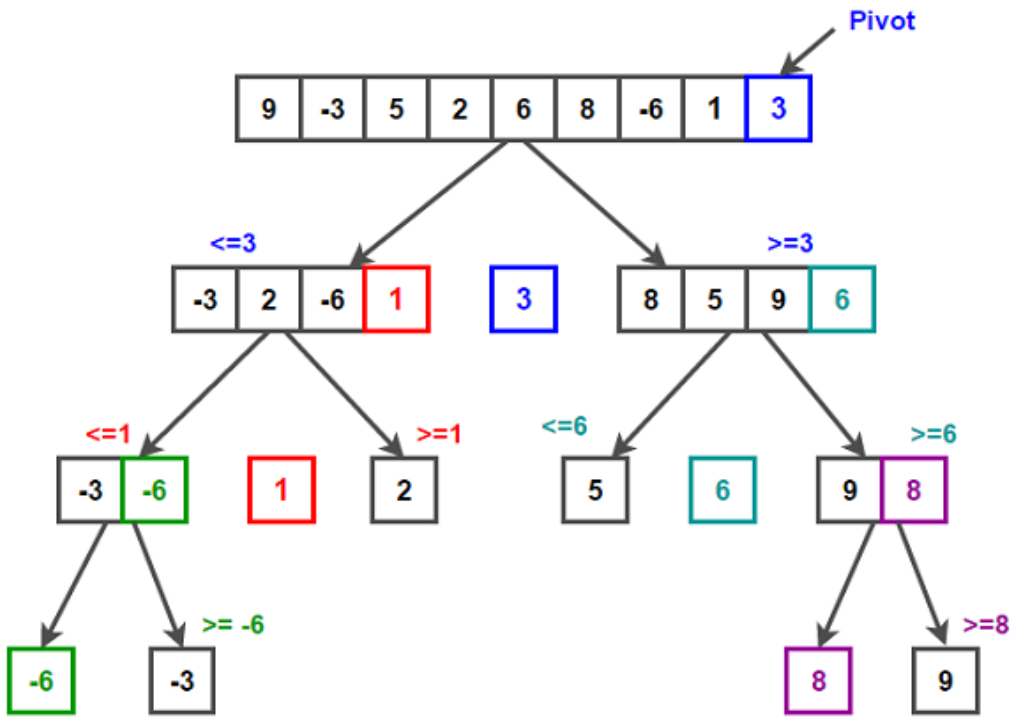
```
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

Quick Sort

Quick sort is an algorithm that uses a divide and conquer technique. In this algorithm, we partition arrays into 2 sub arrays and select a pivot element. If there are elements that are less than the pivot, then they go to the left array and if there are elements that are bigger than the pivot then they go to the right sub array.

The cool thing about this algorithm is that it is recursive, meaning that within each sub array, there will be another partition where eventually all the elements will be sorted. This means that this algorithm could be very fast.

Below is a diagram from techiedelight.com that explains how quicksort works. This diagram truly helped me understand how quicksort works.



Below is the python pseudo code for quick sort. This was taken from the assignment pdf.

Partition in Python

```
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

Recursive Quicksort in Python

```
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

Command Line options & main()

In this assignment, we need to utilize the command line in order to enable the different sorting algorithms. The command line should have the following options:

- -a : enables all sorting algorithms
- -e : Enables Heap Sort
- -i : Enables insertions sort
- -s : Enables shell sort
- -q : Enables quick sort
- -r seed: Sets the random seed. (13371453 is the default seed.)
- -n size: This sets the array size. (100 elements is the default)
- -p elements: This prints out the number of elements from the array (100 is the default value)
- -h : prints out the help message

We have seen in assignment 2 how to use the command line options in main() however, I will be using a different method to call each sort. Instead of using booleans to call each function, I will be using sets.

Move and Comparison tracking

In this assignment, we are required to keep track of the number of moves and compares our algorithm took to sort the array. In order to handle this, we are provided with a stats module that already keeps track of moves and compares. By calling one of its functions (such as cmp() and swap()), the module not only will swap or compare the elements, it will also keep track of its respective moves and compare fields.

By using sets and enumeration, I can iteratively check what command line option is called. The rough pseudo code below demonstrates how I can iteratively check what command line option is called.

```
for x in enumerations_of_algorithms:
    if x in set_of_algorithms:
        randomize_array

        if x is insertion:
            do insertion sort
        elif x is heap:
            do heap sort
        elif x is shell:
            do shell sort
        else:
            do quick sort

        print stats
        print arr
```

In my main(), whenever a certain command line option is called, I will insert the sorting algorithms enumeration in the set. When in the for loop, I will then check to see if that enumeration is in that set. If it is not in that set, then that algorithm will not be called.

Credits

- Eugene's 10/12 section taught me how to use sets rather than booleans in my code
- All pseudo code shown here was provided by Prof. Long in the assignment pdf
- The pseudo code for using enumerations and iterating through each enumeration was inspired from Euegne's 10/12 section