

Sorting Homework

Homework #3:

1. Collections.sort(list) – Merge Sort :

First things first, I started to run the program original sorting method (Collections.sort(list)), to be the base of the algorithm better performance comparison and it uses **merge sort** algorithm. Basically, I run the original method(Collections.sort(list)), and implement the following code separately, just for curiosity.

a. Main function that sorts arr[l...r]:

```
void mergeSort( int l, int r)
{
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort( l, m);
        mergeSort(m + 1, r);
        merge( l, m, r);
    }
```

Find the middle point

Sort first and second halves

Merge the sorted halves

b. Merges two subarrays of arr[], first subarray is arr[l..m], second subarray is arr[m+1..r] :

```
void merge( int l, int m, int r)
{
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[] = new int[n1];
    int R[] = new int[n2];

    for (int i = 0; i < n1; ++i)
        L[i] = (int) list.get(l+i);
    for (int j = 0; j < n2; ++j)
        R[j] = (int) list.get(m+1+j);

    int i = 0, j = 0;
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            list.set(k, L[i]);
            i++;
        }
        else {
            list.set(k, R[j]);
            j++;
        }
        k++;
    }
    while (i < n1) {
        list.set(k, L[i]);
        i++;
        k++;
    }
    while (j < n2) {
        list.set(k, R[j]);
        j++;
        k++;
    }
}
```

Find sizes of two subarrays to be merged

Create temp arrays

Copy elements to temp arrays

Checking elements in L&R arrays and take the smallest to get the merged subarray

Copy remaining elements of L [] if there is any, and the remaining elements of R [] if there is any

Results:

Merge sort doesn't having best or worst cases like quick sort, it is just average time($n \log n$), so as results :

When I run the original method (Collections.sort(list)) : when I run the code above :

```
Run Configuration x  sorting [compile] x
Iteration 1: 30.079
30.248 s/op
Iteration 2: 36.002
36.249 s/op
Iteration 3: 44.38
44.593 s/op
Iteration 4: 34.669
34.913 s/op
Iteration 5: 32.292
32.487 s/op
Iteration 6: 32.311
32.518 s/op
Iteration 7: 34.901
35.114 s/op
Iteration 8: 33.275
33.345 s/op
Iteration 9: 41.511
41.597 s/op
Iteration 10: 35.25
35.480 s/op

Result: 35.654 ±(99.9%) 6.588 s/op [Average]
Statistics: (min, avg, max) = (30.248, 35.654, 44.593), stdev = 4.357
Confidence interval (99.9%): [29.067, 42.242]

# Run complete. Total time 00:07:12

Benchmark      Mode  Samples  Score  Score error  Units
c.b.MyBenchmark.compete  avgt    10    35.654    6.588    s/op
```

```
sorting [compile] x  Run Configuration x
Iteration 1: 33.936
34.046 s/op
Iteration 2: 34.902
35.008 s/op
Iteration 3: 33.669
33.739 s/op
Iteration 4: 33.75
33.857 s/op
Iteration 5: 32.841
32.962 s/op
Iteration 6: 33.971
34.088 s/op
Iteration 7: 34.212
34.324 s/op
Iteration 8: 35.798
35.926 s/op
Iteration 9: 34.398
34.514 s/op
Iteration 10: 35.416
35.509 s/op

Result: 34.397 ±(99.9%) 1.331 s/op [Average]
Statistics: (min, avg, max) = (32.962, 34.397, 35.926), stdev = 0.880
Confidence interval (99.9%): [33.067, 35.728]

# Run complete. Total time 00:07:29

Benchmark      Mode  Samples  Score  Score error  Units
c.b.MyBenchmark.compete  avgt    10    34.397    1.331    s/op
```

That's not a big difference, but just to make sure that the average of seconds per iteration is accurate, I took the average of those two averages.

Now, we can say that Merge Sort Algorithm took **35.06 avg seconds per iteration**, and that's a long time compared to the other sorting algorithms I run later.

2. QuickSort Algorithm :

I run quicksort algorithm code through these main steps:

- Select the pivot** (which divides the arraylist into two halves in such a way that all the elements smaller than the pivot will be on the left side of it and all the elements greater than the pivot will be on the right side of the pivot), and here I tried four methods of selecting the pivot:
 - The first element
 - The last element
 - Random element
 - The middle element
- After selecting an element as pivot, it divides the arraylist for the first time. In quick sort, we call this **partitioning**. Inside this phase, (i) will start from pivot point and (j) starting from last element , (i) will search for elements which are bigger than pivot and (j) will search for the element which are smaller than pivot. When both (i) find bigger than pivot and (j) finds smaller than pivot, swap them. Continue this

process until (i) become greater than (j), and here we stop swapping (i) and (j). Finally, put pivot in place of (j).

```
int partition(int start, int end) {
    int i = start;
    int j = end;

    Random r = new Random();

    //int pivotIndex = start;
    int pivotIndex = j-1;
    //int pivotIndex = nextIntInRange(i, j, r);
    // int pivotIndex = (i+j) /2;

    int pivot = (int) list.get(pivotIndex);

    while(true) {
        while((int) list.get(j) > pivot && j > i) {
            j--;
        }

        while((int) list.get(i) < pivot && i < end) {
            i++;
        }

        if(i < j) {
            //swap
            int temp;
            temp = (int) list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
            j--;
            i++;
        } else {
            return j;
        }
    }
}
```

The first element

The last element

Random element

The middle element

Decreasing (j) while it's bigger than pivot and it's bigger than (i)

Increase (i) while it's less than pivot and it's smaller than (j)

Swapping (j) and (i), when (i) finds bigger than pivot and (j) finds smaller than pivot, if (i < j)

```
// Below method is to just find random integer from given range
static int nextIntInRange(int min, int max, Random rng) {

    int diff = max - min;
    if (diff >= 0 && diff != Integer.MAX_VALUE) {
        return (min + rng.nextInt(diff + 1));
    }
    int i;
    do {
        i = rng.nextInt();
    } while (i < min || i > max);
    return i;
}
```

This function I called it in partitioning function when I select the pivot as a random number

- c. The pivot element will be at its **final sorted position**.
- d. Now, all the elements are **smaller** than the pivot will be **on the left side** of the pivot and all the elements **greater** than the pivot will be **on the right side** of it. Note that, the elements to the left and

right may not be sorted. Then pick subarraylists, from the left of pivot and the right of pivot, and **perform quicksort** recursively on either side.

```
public void quickSort(int start, int end) {
    int j;
    if (start < end) {
        j = partition(start, end);
        quickSort(start, j);
        quickSort(j+1, end);
    }
}
```

Pivot takes its final position

The left subarraylist

The right subarraylist

Results:

At first, I thought that the method of selecting the pivot would determine the worst case and the best case of this algorithm. But I conclude that:

It doesn't matter which pivot to select. What matters is how good partition is done by **pivot's final position**. If the pivot in the final position is in **middle** of array means its **best case** and the array is splitted in two $n/2$ sub arrays and it's nearly balanced, nearly time complexity is $O(n \lg n)$. If the pivot's final position is **at left most of the arraylist** then it ends up having 0, $n-1$ sub problems which is **worst case** and it's not balanced, nearly time complexity is $O(n^2)$.

And here an example to clear my conclusion:

Like in (case2), before partitioning, selecting the last element as pivot but its last position after partitioning is in middle and it achieves the best case, time complexity $O(n \lg n)$

Like in (case4), before partitioning, selecting the middle element as pivot but its last position after partitioning is in the first of the arraylist and it achieves the worst case, time complexity $O(n^2)$.

Before partitioning

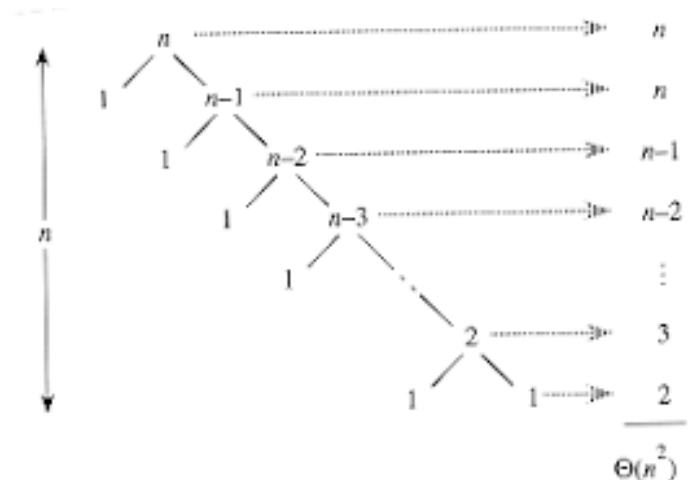
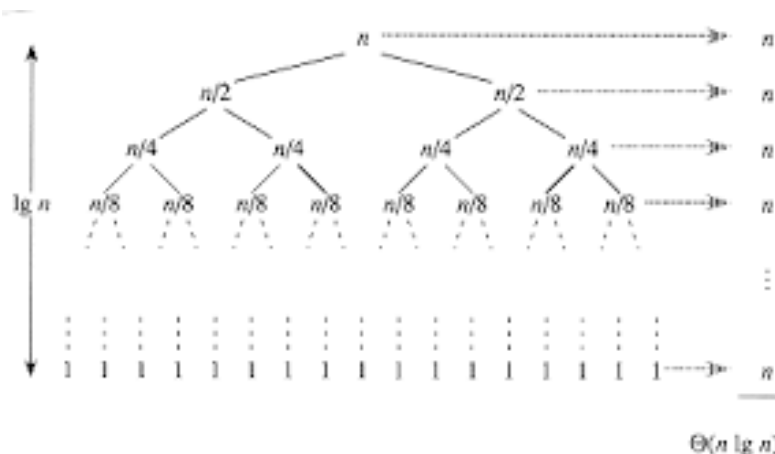
75	26	15	67	85	54	31	49
----	----	----	----	----	----	----	----

4	2	3	5	6	9
---	---	---	---	---	---

After partitioning

26	15	31	49	85	54	75	67
----	----	----	----	----	----	----	----

3	2	4	5	6	9
---	---	---	---	---	---



So as a result, when I selected the pivot at (case1+case2) it gives me less time complexity (avg sec. case(1): 27.788, avg sec. case(2): 28.464) than selecting it randomly or at middle (case3+case4) (avg sec. case(3): 33.036, avg sec. case(4): 32.401), which means that when it partitioned at (case1+case2), and took its last position it was maybe closest to middle but surly more balanced, than the last two cases.

And this situation is just for the arraylist I experimenting on, so if I run quick sort algorithm on another arraylist , maybe it would be better to choose case4 as general case of selecting pivot as middle element or case3 (randomly), as long as the file that is randomly generated it will has a probability of $1/n!$. So, if the arraylist is **already sorted** it will achieve the best case $O(n \log n)$ when selecting it randomly or at the middle. So, selecting pivot randomly maybe benefit for some arrays and it'll achieves the best case $O(n \log n)$.

Case 1 (selecting the first element as pivot):

```
Iteration 1: 27.562
27.635 s/op
Iteration 2: 27.749
27.823 s/op
Iteration 3: 27.463
27.533 s/op
Iteration 4: 27.735
27.805 s/op
Iteration 5: 27.643
27.727 s/op
Iteration 6: 27.664
27.800 s/op
Iteration 7: 28.496
28.578 s/op
Iteration 8: 27.557
27.626 s/op
Iteration 9: 27.551
27.619 s/op
Iteration 10: 27.633
27.738 s/op

Result: 27.788 ±(99.9%) 0.444 s/op [Average]
Statistics: (min, avg, max) = (27.533, 27.788, 28.578), stdev = 0.293
Confidence interval (99.9%): [27.345, 28.252]

# Run complete. Total time: 00:06:03

Benchmark      Mode  Samples  Score  Score error  Units
c.b.MyBenchmark.compete  avgt    10    27.788      0.444    s/op
```

Case 2 (selecting the last element as pivot):

```
Iteration 1: 29.051
29.125 s/op
Iteration 2: 28.823
28.909 s/op
Iteration 3: 28.234
28.306 s/op
Iteration 4: 28.132
28.226 s/op
Iteration 5: 28.473
28.541 s/op
Iteration 6: 28.16
28.245 s/op
Iteration 7: 28.477
28.548 s/op
Iteration 8: 28.252
28.334 s/op
Iteration 9: 28.109
28.180 s/op
Iteration 10: 28.127
28.224 s/op

Result: 28.464 ±(99.9%) 0.487 s/op [Average]
Statistics: (min, avg, max) = (28.180, 28.464, 29.125), stdev = 0.322
Confidence interval (99.9%): [27.977, 28.950]

# Run complete. Total time: 00:06:12

Benchmark      Mode  Samples  Score  Score error  Units
c.b.MyBenchmark.compete  avgt    10    28.464      0.487    s/op
```

Case 3 (selecting the middle element as pivot):

```
Iteration 1: 32.067
32.157 s/op
Iteration 2: 31.826
31.895 s/op
Iteration 3: 33.994
34.070 s/op
Iteration 4: 33.265
33.340 s/op
Iteration 5: 32.402
32.486 s/op
Iteration 6: 33.605
33.683 s/op
Iteration 7: 33.031
33.121 s/op
Iteration 8: 32.735
32.867 s/op
Iteration 9: 34.584
34.652 s/op
Iteration 10: 32.002
32.085 s/op

Result: 33.036 ±(99.9%) 1.381 s/op [Average]
Statistics: (min, avg, max) = (31.895, 33.036, 34.652), stdev = 0.913
Confidence interval (99.9%): [31.655, 34.417]

# Run complete. Total time: 00:07:10

Benchmark      Mode  Samples  Score  Score error  Units
c.b.MyBenchmark.compete  avgt    10    33.036     1.381    s/op
```

Case 4 (selecting randomly the pivot):

```
# Warmup Iteration 1: 33.082
33.322 s/op
# Warmup Iteration 2: 32.64
32.723 s/op
# Warmup Iteration 3: 32.935
33.038 s/op
Iteration 1: 32.231
32.316 s/op
Iteration 2: 32.265
32.334 s/op
Iteration 3: 32.223
32.294 s/op
Iteration 4: 32.102
32.174 s/op
Iteration 5: 32.058
32.128 s/op
Iteration 6: 32.465
32.536 s/op
Iteration 7: 32.299
32.388 s/op
Iteration 8: 32.045
32.922 s/op
Iteration 9: 32.561
32.630 s/op
Iteration 10: 32.214
32.287 s/op

Result: 32.401 ±(99.9%) 0.358 s/op [Average]
Statistics: (min, avg, max) = (32.128, 32.401, 32.922), stdev = 0.237
Confidence interval (99.9%): [32.043, 32.759]
```

Also I've tried three Way Partition. In simple QuickSort algorithm, we select an element as pivot, partition the array around a pivot and recur for subarrays on the left and right of the pivot.

The idea of 3 way Quick Sort is to process all occurrences of the pivot.

In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:

- a) arr[l..i] elements less than pivot.
- b) arr[i+1..j-1] elements equal to pivot.
- c) arr[j..r] elements greater than pivot.

```
public void threeWayPartition( int lowVal, int highVal)
{
    int n =(int) list.size();
    int start = 0, end = n-1;
    for(int i = 0; i <= end; i++)
    {
        if((int)list.get(i)< lowVal)
        {
            int temp = (int)list.get(start);
            list.set(start, (int)list.get(i));
            list.set(i,temp);
            start++;
            i++;
        }

        else if((int)list.get(i) > highVal)
        {
            int temp = (int)list.get(end);
            list.set(end, (int)list.get(i));
            list.set(i,temp);
            end--;
        }
        else
            i++;
    }
}
```

Results:

It gave me **4.165 avg seconds per iteration**. The reason behind this big difference between normal partitioning and three way partitioning is the way of partitioning where the partition is of elements smaller, equal, and larger than the pivot. Only the smaller and larger partitions have to be sorted recursively.

Run Configuration					
Iteration	1:	3.774			
		3.816 s/op			
Iteration	2:	4.732			
		4.775 s/op			
Iteration	3:	3.889			
		3.842 s/op			
Iteration	4:	3.752			
		3.789 s/op			
Iteration	5:	3.798			
		3.843 s/op			
Iteration	6:	3.747			
		3.780 s/op			
Iteration	7:	4.325			
		4.367 s/op			
Iteration	8:	4.799			
		4.852 s/op			
Iteration	9:	3.77			
		3.809 s/op			
Iteration	10:	4.741			
		4.774 s/op			
Result: 4.165 ±(99.9%) 0.713 s/op [Average]					
Statistics: (min, avg, max) = (3.780, 4.165, 4.852), stdev = 0.471					
Confidence interval (99.9%): [3.452, 4.868]					
# Run complete. Total time: 00:00:56					
Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark compete	avgt	10	4.165	0.713	s/op

****Fianl Note :**

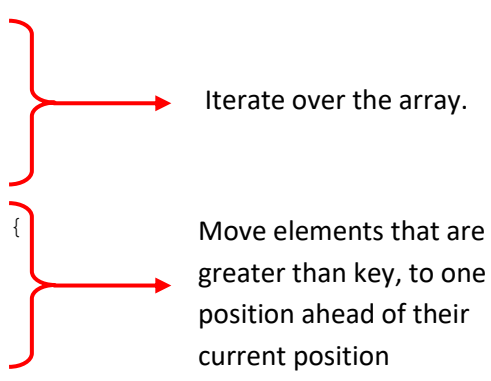
Quicksort has $O(n^2)$ worst-case runtime, and $O(n \log n)$ average case runtime. Surprisingly, quicksort runs faster than merge sort and that's because many factors affect an algorithm's runtime, and, when try them both, it seems like quicksort wins out. Also, the number of comparisons or the number of swaps necessary to perform to sort the data, affects runtime. Quicksort in particular requires little additional space, and this makes it **faster** than merge sort in **many cases**.

3. Insertion Sort:

In insertion sort algorithm, the array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

```
void insertionSort()
{
    int n = (int) list.size();
    for (int i = 1; i < n; ++i) {
        int key = (int) list.get(i);
        int j = i - 1;

        while (j >= 0 && (int) list.get(j) > key) {
            list.set(j+1, j);
            j = j - 1;
        }
        list.set(j+1, key);
    }
}
```



Iterate over the array.

Move elements that are greater than key, to one position ahead of their current position

It compares the current element(key) with the largest value in the sorted array. If the current element(key) is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Results:

Insertion sort will be slow on large lists because it runs in $O(n^2)$ so if (n) gets bigger, time complexity will increase too. Also, shifting each element 1 by 1 slowly until it finds the location. That's a big amount of elements moving around for every insert. So, when I run the code, it passed 30 mins and not completing the first warm up iteration. So, maybe if we try to use this algorithm with small arraylists(next step) it will be more effective.

4. Combine of QuickSort & InsertionSort Algorithm :

As we saw previously, the Quicksort is one of the fastest sorting algorithms for sorting **large lists** of data. The Insertion sort is a fast sorting algorithms for sorting very **small lists that are already somewhat sorted**. Since the insertion sort was so slow on my computer and because the arraylist is too large, why not to combine these two algorithms to come up with a very simple and effective method for sorting this large arraylist.

It started by using a Quicksort. The Quicksort on average runs $O(n \log(n))$ but it can be as slow as $O(n^2)$ if we try to sort a list that is already sorted and the final position of the pivot was the first element of the arraylist.

As the Quicksort works, it breaks down the arraylist into smaller and smaller arraylists. Once it becomes small enough that an **Insertion sort becomes more efficient than the Quicksort** it will switch to the Insertion sort that it runs in worst case $O(n^2)$.

the code below, shows where is the right place of calling the insertion sort algorithm :

```
public void quickSortwithInsertion( int start, int end) {  
    if (start < end) {  
        if ((end - start) < 9) {  
            insertionSort(start, end);  
        }  
        else {  
            int part = partition(start, end);  
            quickSortwithInsertion(start, part);  
            quickSortwithInsertion(part + 1, end);  
        }  
    }  
}
```

**** Note below this code**

This is where it'll switch to Insertion Sort

****Note :** this condition decide that the array is small enough to switch to the Insertion sort and the point for when the running time is the best in insertion sort than the quicksort is **around 9**.

Results:

For now, this combination of algos gives less total time for completing run than quicksort and mergesort of the whole iterations: 4:33 mins on **20.977 avg seconds per iteration**. So, it's way better than quick sort , merge sort and surly, insertion sort.

Run Configuration ×					
Iteration 1:	19.624				
	19.674 s/op				
Iteration 2:	19.631				
	19.715 s/op				
Iteration 3:	21.727				
	21.832 s/op				
Iteration 4:	23.815				
	23.883 s/op				
Iteration 5:	20.498				
	20.550 s/op				
Iteration 6:	21.098				
	21.220 s/op				
Iteration 7:	20.257				
	20.338 s/op				
Iteration 8:	20.141				
	20.261 s/op				
Iteration 9:	21.354				
	21.421 s/op				
Iteration 10:	20.773				
	20.875 s/op				
Result: 20.977 ±(99.9%) 1.875 s/op [Average]					
Statistics: (min, avg, max) = (19.674, 20.977, 23.883), stdev = 1.240					
Confidence interval (99.9%): [19.102, 22.851]					
# Run complete. Total time: 00:04:33					
Benchmark	Mode	Samples	Score	Score error	Units
c.b.MyBenchmark.compete	avgt	10	20.977	1.875	s/op

5. Heap Sort:

```
public void heap()
```

```
{  
    int n = list.size();
```

```
1 for (int i = n / 2 - 1; i >= 0; i--)  
    heapify( n, i);
```

Build a heap from largest non-leaves up to index 0 because leaves has no left or right child to compare with

```
2 for (int i=n-1; i>=0; i--)
```

One by one extract the final element from heap

```
{  
    int temp = (int) list.get(0);  
    list.set(0,i);  
    list.set(i,temp);  
    heapify( i, 0);
```

Swap root element with end element and apply the heap property

```
}
```

```
}
```

Heap sort have two steps:

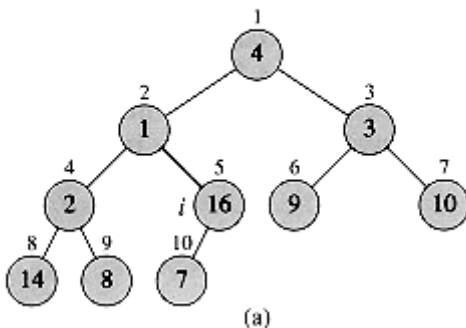
1- Build a max heap by inserting the elements one by one

Building a max heap step has two sub-steps. **First**, put all the elements in the tree by using tree indexing. **Second**, apply heapify(it'll be explained below) on all non leaves nodes up to index 0. But still the tree is not sorted as we want(some children in the same level are not sorted). This example shows, before applying heapify and after applying heapify on the new heap:

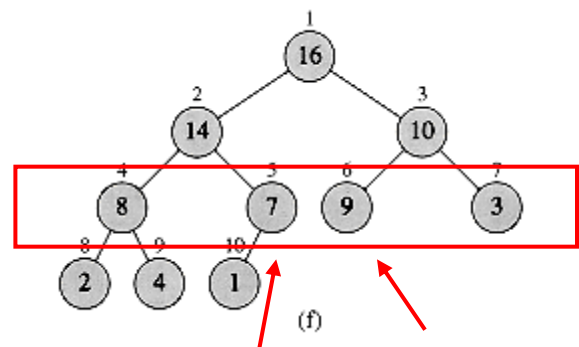
Before applying heapify

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



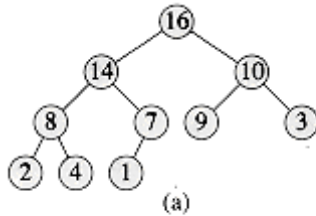
After applying heapify



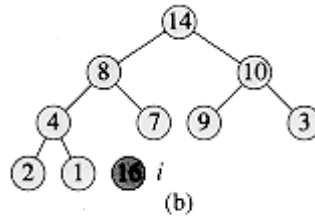
Note that, for example, 9, 7 at the same level and not in their right place, so the arraylist is not sorted yet!

2- Remove largest item and place it in sorted partition

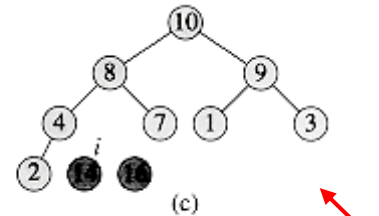
Now we have a max heap so we know the largest element, it swap with the last element, and then remove the last element from the tree and considered it's sorted, but wait, in this situation we've got a tree not a max heap so we call heapify to return it to a max heap.



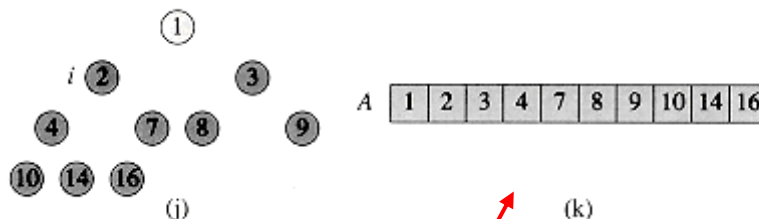
Here we have a max heap



it will take the last element
swap it, remove it from the tree
then, call heapify



Repeating the process through
all elements



And this is the final sorted arraylist

**Explaining heapify method :

```
void heapify( int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && (int) list.get(l) > (int) list.get(largest))
        largest = l;

    if (r < n && (int) list.get(r) > (int) list.get(largest))
        largest = r;

    if (largest != i)
    {
        int swap = (int) list.get(i);
        list.set(i, largest);
        list.set(largest, swap);
        heapify( n, largest);
    }
}
```

Initialize largest as root

Left child

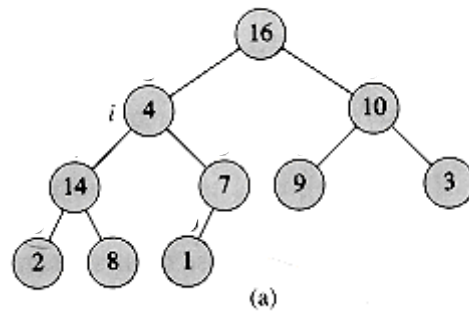
Right child

(l < n) checks if there is left child, and the other condition checks If left child is larger than root

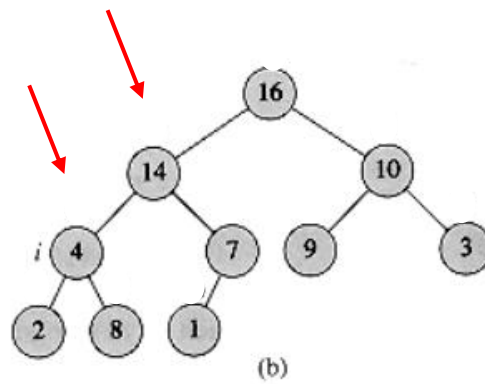
If largest is not root change it with the largest if changed before, and recursively heapify the affected sub-tree

(r < n) checks if there is right child, and the other condition checks If right child is larger than largest

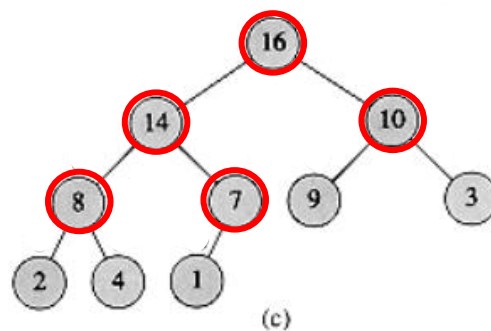
At this example, if we assume that (i) was passed in heapify function, since it is not larger than its both children,



The largest element will be in root will swap with its smallest children, here (14 and 4 will swap and produce this), and this affect the tree and not make it a max heap:



Here, the tree at element 4 not a heap.. the root is smaller than right child, so swap them and now it's a max heap:



Results:

Finally, this algorithm gives a less total time for completing run the whole iterations: 1:48 mins on **average time 8.080 seconds per iteration**. So, it's one of the best of all previous sorting algorithms I tested. And that's because HeapSort have guaranteed worst-case running time of $O(n \log n)$ as opposed to QuickSort's average running time of $O(n \log n)$. QuickSort is usually used in practice, because typically it is faster, but HeapSort is used for external sort when you need to sort huge files.

```
Iteration 1: 9.146
9.261 s/op
Iteration 2: 7.998
8.106 s/op
Iteration 3: 7.487
7.566 s/op
Iteration 4: 7.978
8.090 s/op
Iteration 5: 7.99
8.100 s/op
Iteration 6: 7.921
7.987 s/op
Iteration 7: 7.883
8.002 s/op
Iteration 8: 9.109
9.205 s/op
Iteration 9: 7.131
7.229 s/op
Iteration 10: 7.141
7.254 s/op

Result: 8.080 ±(99.9%) 1.051 s/op [Average]
Statistics: (min, avg, max) = (7.229, 8.080, 9.261), stdev = 0.695
Confidence interval (99.9%): [7.029, 9.131]

# Run complete. Total time: 00:01:48
```

Comparing all algorithms :

	Best case	Average time	Worst case	Total time (min)	Avg sec/iteration	Good for our arraylist?
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	7:20	35.06	No
Quick Sort	(three-way partition) , first element as pivot $O(n)$	$O(n \log n)$	$O(n^2)$	00:56	4.165	Yes!
	(simple partition) , first element as pivot $O(n \log n)$	$O(n \log n)$	$O(n^2)$	6:03	27.788	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	It doesn't complete its run		No
Quick + insertion				4:33	20.977	Maybe
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log n)$	1:48	8.080	Yes

