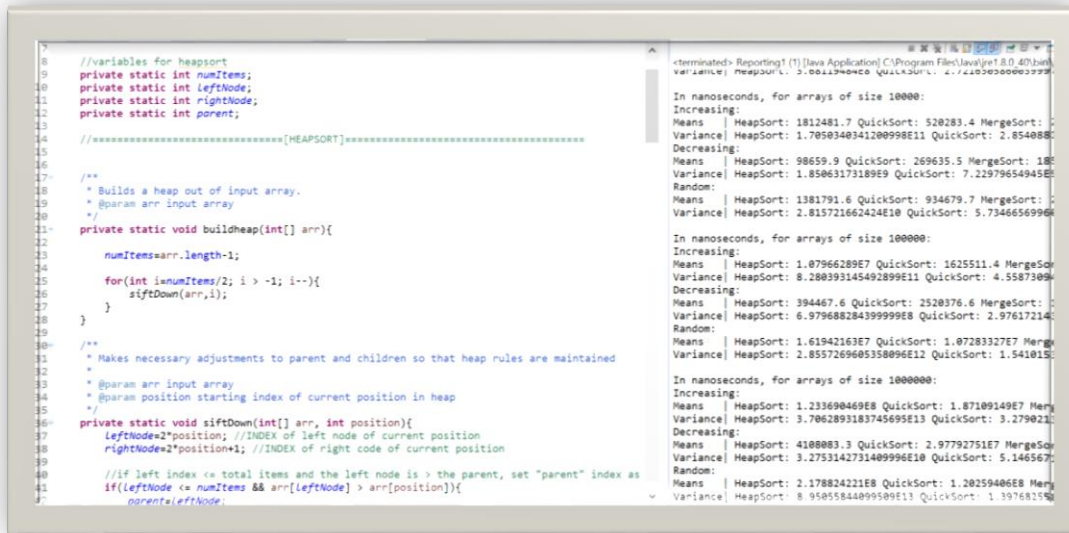


Analysis of Heapsort, Quicksort and Mergesort

Performance of three sorting algorithms on Arrays of varying size



```
7 //variables for heapsort
8 private static int numItems;
9 private static int leftNode;
10 private static int rightNode;
11 private static int parent;
12
13 //=====HEAPSORT=====
14
15 /**
16  * Builds a heap out of input array.
17  * @param arr input array
18  */
19 private static void buildHeap(int[] arr){
20     numItems=arr.length-1;
21     for(int i=numItems/2; i > -1; i--){
22         siftDown(arr,i);
23     }
24 }
25
26 /**
27  * Makes necessary adjustments to parent and children so that heap rules are maintained
28  *
29  * @param arr input array
30  * @param position starting index of current position in heap
31  */
32 private static void siftDown(int[] arr, int position){
33     leftNode=2*position; //INDEX of left node of current position
34     rightNode=2*position+1; //INDEX of right node of current position
35
36     //if left index <= total items and the left node is > the parent, set "parent" index as
37     if(leftNode <= numItems && arr[leftNode] > arr[position]){
38         parent=leftNode;
39     }
40     //if right index <= total items and the right node is > the parent, set "parent" index as
41     if(rightNode <= numItems && arr[rightNode] > arr[parent]){
42         parent=rightNode;
43     }
44     //swap parent with child
45     int temp=arr[parent];
46     arr[parent]=arr[position];
47     arr[position]=temp;
48     siftDown(arr,parent);
49 }
50
51 //=====MERGESORT=====
52
53 /**
54  * Splits the array into two halves and sorts each half recursively.
55  * @param arr input array
56  * @param left starting index of left half
57  * @param right ending index of right half
58  */
59 private static void mergeSort(int[] arr, int left, int right){
60     if(left < right){
61         int mid=(left+right)/2;
62         mergeSort(arr, left, mid);
63         mergeSort(arr, mid+1, right);
64         merge(arr, left, mid, right);
65     }
66 }
67
68 /**
69  * Merges two sorted arrays into a single sorted array.
70  * @param arr input array
71  * @param left starting index of left half
72  * @param mid ending index of left half
73  * @param right ending index of right half
74  */
75 private static void merge(int[] arr, int left, int mid, int right){
76     int n1=mid-left+1;
77     int n2=right-mid;
78     int[] L=new int[n1];
79     int[] R=new int[n2];
80     for(int i=0; i<n1; i++){
81         L[i]=arr[left+i];
82     }
83     for(int i=0; i<n2; i++){
84         R[i]=arr[mid+1+i];
85     }
86     int i=0, j=0, k=left;
87     while(i<n1 && j<n2){
88         if(L[i]<=R[j]){
89             arr[k]=L[i];
90             i++;
91         }
92         else{
93             arr[k]=R[j];
94             j++;
95         }
96         k++;
97     }
98     while(i<n1){
99         arr[k]=L[i];
100         i++;
101         k++;
102     }
103     while(j<n2){
104         arr[k]=R[j];
105         j++;
106         k++;
107     }
108 }
109
110 //=====QUICKSORT=====
111
112 /**
113  * Partitions the array around a pivot element and sorts the two resulting sub-arrays recursively.
114  * @param arr input array
115  * @param left starting index of left half
116  * @param right ending index of right half
117  */
118 private static void quickSort(int[] arr, int left, int right){
119     if(left < right){
120         int pivot=arr[right];
121         int i=left-1;
122         for(int j=left; j<right; j++){
123             if(arr[j]<=pivot){
124                 i++;
125                 swap(arr,i,j);
126             }
127         }
128         swap(arr,i+1,right);
129         quickSort(arr, left, i);
130         quickSort(arr, i+1, right);
131     }
132 }
133
134 /**
135  * Swaps two elements in an array.
136  * @param arr input array
137  * @param i index of first element
138  * @param j index of second element
139  */
140 private static void swap(int[] arr, int i, int j){
141     int temp=arr[i];
142     arr[i]=arr[j];
143     arr[j]=temp;
144 }
145
146 //=====REPORTING=====
147
148 /**
149  * Reports the performance of the three sorting algorithms.
150  * @param arr input array
151  * @param size size of array
152  */
153 private static void reportPerformance(int[] arr, int size){
154     System.out.println("In nanoseconds, for arrays of size "+size);
155     System.out.println("Increasing:");
156     Means | HeapSort: 1812481.7 QuickSort: 520283.4 MergeSort: 1812481.7
157     Variance | HeapSort: 1.7050340341200998E11 QuickSort: 2.854088E10 MergeSort: 1.7050340341200998E11
158     Decreasing:
159     Means | HeapSort: 98659.9 QuickSort: 269635.5 MergeSort: 1812481.7
160     Variance | HeapSort: 1.85063173189E9 QuickSort: 7.22979654945E9 MergeSort: 1.7050340341200998E11
161     Random:
162     Means | HeapSort: 1381791.6 QuickSort: 934679.7 MergeSort: 1812481.7
163     Variance | HeapSort: 2.815721662424E10 QuickSort: 5.7346656996E10 MergeSort: 1.7050340341200998E11
164
165     In nanoseconds, for arrays of size 100000:
166     Increasing:
167     Means | HeapSort: 1.07966289E7 QuickSort: 1625511.4 MergeSort: 1812481.7
168     Variance | HeapSort: 8.280393145492899E11 QuickSort: 4.5587309E10 MergeSort: 1.7050340341200998E11
169     Decreasing:
170     Means | HeapSort: 394467.6 QuickSort: 2520376.6 MergeSort: 1812481.7
171     Variance | HeapSort: 6.979688284399999E8 QuickSort: 2.97617214E10 MergeSort: 1.7050340341200998E11
172     Random:
173     Means | HeapSort: 1.61942163E7 QuickSort: 1.07283327E7 MergeSort: 1812481.7
174     Variance | HeapSort: 2.8557269605358096E12 QuickSort: 1.541015E10 MergeSort: 1.7050340341200998E11
175
176     In nanoseconds, for arrays of size 1000000:
177     Increasing:
178     Means | HeapSort: 1.233690469E8 QuickSort: 1.87109149E7 MergeSort: 1812481.7
179     Variance | HeapSort: 3.7062893183745695E13 QuickSort: 3.279021E10 MergeSort: 1.7050340341200998E11
180     Decreasing:
181     Means | HeapSort: 4108083.3 QuickSort: 2.97792751E7 MergeSort: 1812481.7
182     Variance | HeapSort: 3.2753142731409996E10 QuickSort: 5.146567E10 MergeSort: 1.7050340341200998E11
183     Random:
184     Means | HeapSort: 2.178824221E8 QuickSort: 1.20259406E8 MergeSort: 1812481.7
185     Variance | HeapSort: 8.950558440899509E13 QuickSort: 1.39768125E10 MergeSort: 1.7050340341200998E11
186 }
187
188 //=====MAIN=====
189
190 /**
191  * Main method to run the program.
192  */
193 public static void main(String[] args){
194     if(args.length < 1){
195         System.out.println("Usage: java HeapSortQuicksortMergeSort <size>");
196         return;
197     }
198     int size=Integer.parseInt(args[0]);
199     int[] arr=new int[size];
200     for(int i=0; i<size; i++){
201         arr[i]=(int)(Math.random()*size);
202     }
203     reportPerformance(arr, size);
204 }
```

Katherine Cass

Student ID: KRC53

Class: EECS 233, P4

21 April 2015



Fig. 1: performance of various sorting algorithms on arrays of size 1000, 10000, 100000, and 1000000, respectively.

Some observations based on data generated from my Reporting1 class, printed to the console and then manually entered into excel to generate these graphs (data table included below, and in "Reporting1results.txt"):

- **Heapsort** uses the most efficient number of comparisons, but requires data to be shifted around significantly, unless the data is already decreasing (then the siftdown method doesn't have to do much work). For this reason, it outperforms all other sorts on decreasing arrays of significant size. However, it performs poorly on large random arrays and small sorted arrays.
- **Quicksort** tends to have the overall lowest runtime because it requires a low number of comparisons as well. This algorithm is generally efficient for the array datastructure, with the exception of decreasing arrays, which Heapsort outperforms all in.
- **Mergesort** is generally slow, only outperforming Heapsort when performed on an increasing array and significantly large random arrays. This is understandable, because the data structure used for testing was an array. Mergesort works best on data structures that are not operated on in place, but instead moveable, such as linked list. For this reason, Mergesort would be a good option on very large amounts of data that require large amounts of memory.

Array Size	Increasing Array	Decreasing Array	Random Array
1000	<u>Means</u> HeapSort: 453826.1 QuickSort: 77533.7 MergeSort: 407596.8 <u>Variance</u> HeapSort: 1.1325483705749002E11 QuickSort: 4.177426266409999E9 MergeSort: 5.5712165204759995E10	<u>Means</u> HeapSort: 108110.9 QuickSort: 94169.3 MergeSort: 334553.5 <u>Variance</u> HeapSort: 3.8438126948999995E8 QuickSort: 1.52120861881E9 MergeSort: 4.15161135845E9	<u>Means</u> HeapSort: 249151.0 QuickSort: 234867.4 MergeSort: 395237.4 <u>Variance</u> HeapSort: 3.68119484E8 QuickSort: 2.7216305860039997E10 MergeSort: 6.546485387439999E9
	<u>Means</u> HeapSort: 1812481.7 QuickSort: 520283.4 MergeSort: 2078653.2 <u>Variance</u> HeapSort: 1.7050340341200998E11 QuickSort: 2.8540881568764E11 MergeSort: 3.9411250140095996E11	<u>Means</u> HeapSort: 98659.9 QuickSort: 269635.5 MergeSort: 1857641.9 <u>Variance</u> HeapSort: 1.85063173189E9 QuickSort: 7.22979654945E9 MergeSort: 2.114854258509E10	<u>Means</u> HeapSort: 1381791.6 QuickSort: 934679.7 MergeSort: 2717182.8 <u>Variance</u> HeapSort: 2.815721662424E10 QuickSort: 5.734665699609999E9 MergeSort: 4.382130224211599E11
100000	<u>Means</u> HeapSort: 1.07966289E7 QuickSort: 1625511.4 MergeSort: 8293059.1 <u>Variance</u> HeapSort: 8.280393145492899E11 QuickSort: 4.55873094124E9 MergeSort: 2.1933390109392898E12	<u>Means</u> HeapSort: 394467.6 QuickSort: 2520376.6 MergeSort: 1.18210734E7 <u>Variance</u> HeapSort: 6.979688284399999E8 QuickSort: 2.976172143904E10 MergeSort: 4.464553954496984E13	<u>Means</u> HeapSort: 1.61942163E7 QuickSort: 1.07283327E7 MergeSort: 2.07184495E7 <u>Variance</u> HeapSort: 2.8557269605358096E12 QuickSort: 1.5410153567681003E11 MergeSort: 3.777813188559425E13
	<u>Means</u> HeapSort: 1.233690469E8 QuickSort: 1.87109149E7 MergeSort: 1.045447179E8 <u>Variance</u> HeapSort: 3.7062893183745695E13 QuickSort: 3.2790213132889E11 MergeSort: 8.591474515040539E14	<u>Means</u> HeapSort: 4108083.3 QuickSort: 2.97792751E7 MergeSort: 1.051858562E8 <u>Variance</u> HeapSort: 3.2753142731409996E10 QuickSort: 5.14656717320089E12 MergeSort: 1.3596272642273755E15	<u>Means</u> HeapSort: 2.178824221E8 QuickSort: 1.20259406E8 MergeSort: 1.797325111E8 <u>Variance</u> HeapSort: 8.95055844099509E13 QuickSort: 1.3976825586954E12 MergeSort: 3.7404862719378484E13

Table 1. Output from Reporting1.java, in nanoseconds