Comparison and evaluation of Heap-Sort and Quick-Sort Algorithms

CS 5800 Project - Yuanzhi Xu, Kejian Tong, Ting Ji
https://www.youtube.com/watch?v=9LumphOcHkM Video

Agenda

- Introduction
- Analysis
    - Heap-Sort
    - Quick-Sort
- Comparison
- Enhancement
    - Heap-Sort
    - Quick-Sort
- Conclusion

Introduction

Introduction - Context

- Sorting is a fundamental operation in computer science
- Many sorting algorithms(quick sort, merge sort, heap sort, selection sort, insertion sort,bubble sort...)
- Quick-Sort and Heap-Sort are two representative sorting algorithms(guarantee O(nlogn) complexity if implemented properly, the best performance to achieve with comparison-based sorting)
- Most built-in sort algorithms these days are usually Quick-Sort

Introduction - Why this topic?

- Sorting is an important algorithm and fundamental operation in Computer Science(have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, etc)
- Quick-Sort and Heap-Sort are two representative sorting algorithms
- Most built-in sort algorithms are usually Quick-Sort
- Why Quick-Sort? (given that Quick-Sort’s performance can downgrade to $O(n^2)$ while Heap-Sort’s complexity is always $O(n\log n)$.)

Introduction - Defined Questions

1. There are differences between Quicksort and heap sort. How much are they different?
2. Does one of them beat the other in some factors?
3. How do we evaluate the two algorithms according to the used method, stability, memory usage, and complexity?
4. Are there any enhancements to these algorithms?

Analysis

Analysis - Quick-Sort

- How does it work?
- How about the stability?
- What is the Memory Usage?
- Coding Algorithm

```
QuickSort
1. If p < r then
2. q Partition (A, p, r)
3. Recursive call to Quick Sort (A, p, q)
4. Recursive call to Quick Sort (A, q + r, r)

PARTITION (A, p, r)
1. x ← A[p]
2. i ← p-1
3. j ← r+1
4. while TRUE do
5. Repeat j ← j-1
6. until A[j] ≤ x
7. Repeat i ← i+1
8. until A[i] ≥ x
9. if i < j
10. then exchange A[i] ↔ A[j]
11. else return j
```

- Complexity

Analysis - Heap-Sort

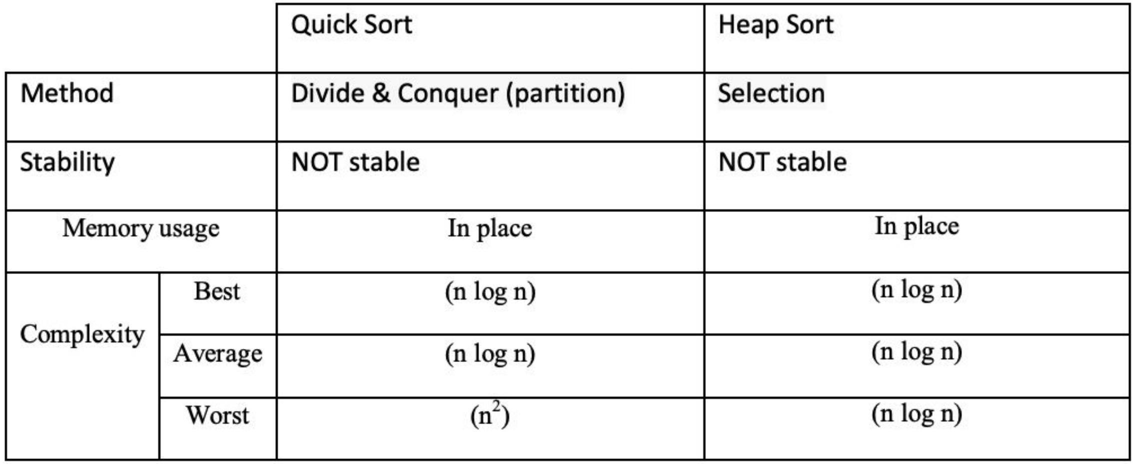
- How does it work?
- How about the stability?
- What is the Memory Usage?
- Coding Algorithm

```
QuickSort
HEAPSORT (A)
1. BUILD_HEAP (A)
2. for i ← length (A) down to 2 do exchange A[1] ↔ A[i] heap-size [A] ← heap-size [A] - 1 Heapify (A, 1)

BUILD_HEAP (A)
1. heap-size (A) ← length [A]
2. For i ← floor(length[A]/2) down to 1 do
3. Heapify (A, i)

Heapify (A, i)
1. l ← left [i]
2. r ← right [i]
3. if l ≤ heap-size [A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heap-size [A] and A[i] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. Heapify (A, largest)
```

- Complexity

Comparison

Comparison

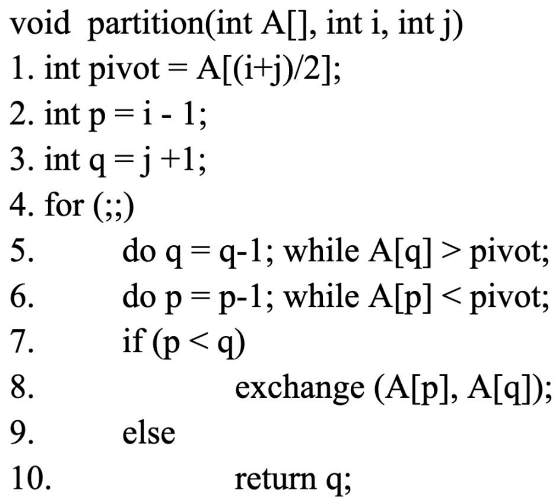


		Quick Sort	Heap Sort
Method		Divide & Conquer (partition)	Selection
Stability		NOT stable	NOT stable
Memory usage		In place	In place
Complexity	Best	(n log n)	(n log n)
	Average	(n log n)	(n log n)
	Worst	(n^2)	(n log n)

Enhancement

Enhancement - Quick-Sort

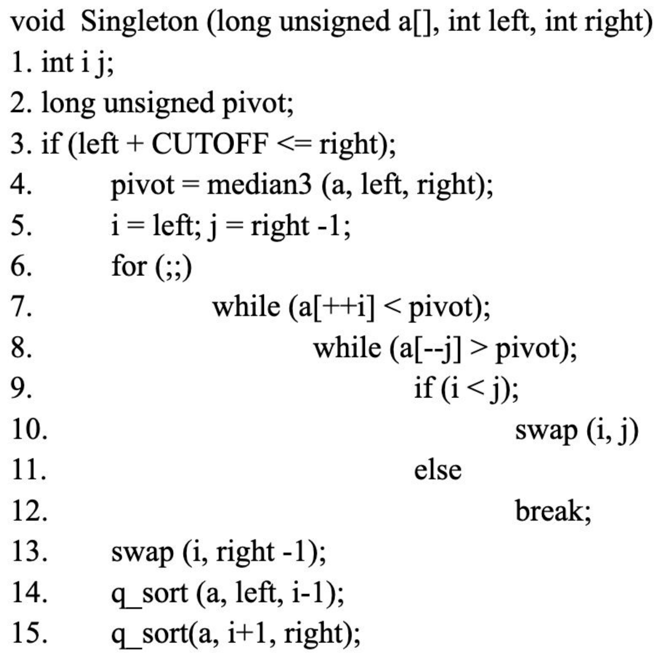
1. Quickersort
   a. The pivot is chosen as the middle key in the array of keys to be sorted.



```
void  partition(int A[], int i, int j)
1. int pivot = A[(i+j)/2];
2. int p = i - 1;
3. int q = j +1;
4. for (;;)
5.        do q = q-1; while A[q] > pivot;
6.        do p = p-1; while A[p] < pivot;
7.        if (p < q)
8.                exchange (A[p], A[q]);
9.        else
10.               return q;
```

Enhancement - Quick-Sort

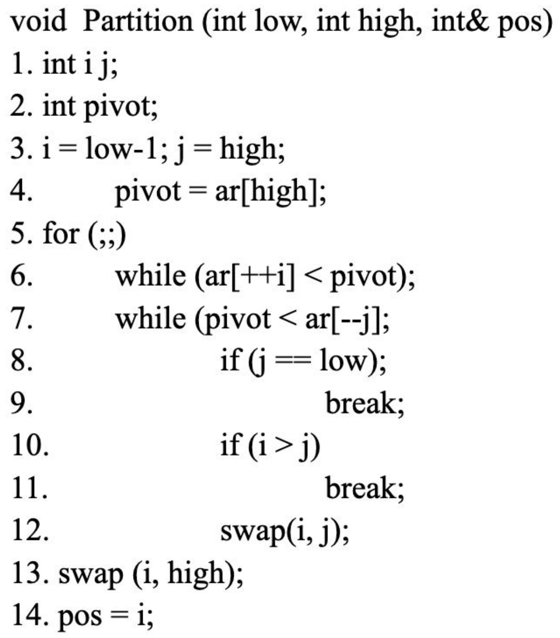
2. Singleton
    a. Median-of-three method for choosing the pivot.
        i. Select the three elements from the left, middle and right of the array.
        ii. The three elements sorted and placed back into the same positions in the array.
        iii. The pivot is the median of these three elements.



```
void  Singleton (long unsigned a[], int left, int right)
int i j;
long unsigned pivot;
if (left + CUTOFF <= right);
        pivot = median3 (a, left, right);
        i = left; j = right -1;
        for (;;)
                while (a[++i] < pivot);
                        while (a[--j] > pivot);
                                if (i < j);
                                        swap (i, j)
                                else
                                        break;
        swap (i, right -1);
        q_sort (a, left, i-1);
        q_sort(a, i+1, right);
```

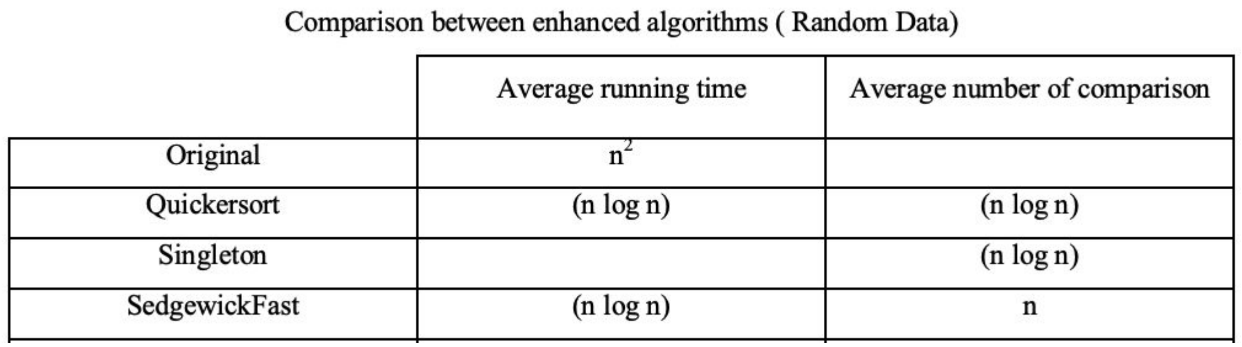
Enhancement - Quick-Sort

3. SedgewickFast
    a. Minimizes the number of swaps
        i. Scanning in from the left of the array
        ii. Scanning in from the right of the array
        iii. Swapping the two numbers found to be out of position.



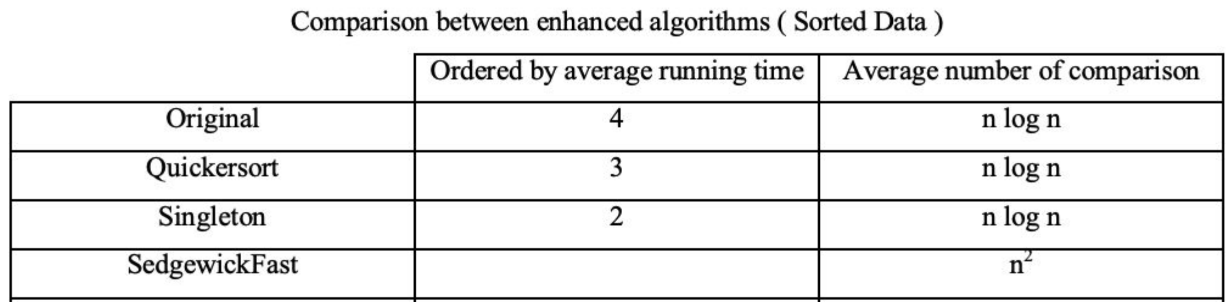
```
void  Partition (int low, int high, int& pos)
int i j;
int pivot;
i = low-1; j = high;
        pivot = ar[high];
for (;;)
        while (ar[++i] < pivot);
        while (pivot < ar[--j];
                if (j == low);
                        break;
                if (i > j)
                        break;
                swap(i, j);
swap (i, high);
pos = i;
```

Enhancement - Quick-Sort



Comparison between enhanced algorithms (Random Data)

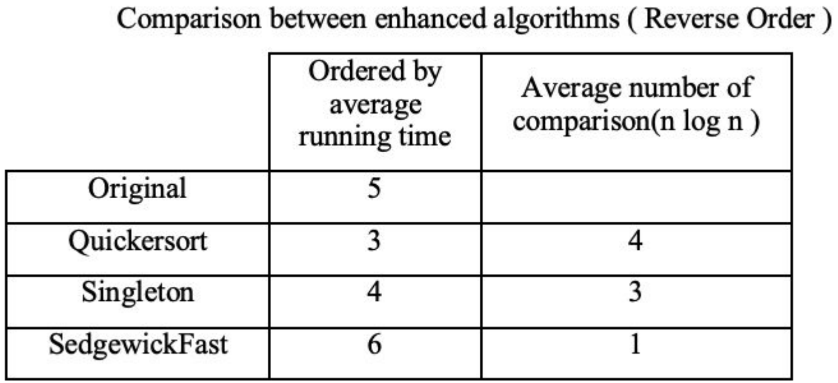
	Average running time	Average number of comparison
Original	n^2	
Quickersort	(n log n)	(n log n)
Singleton		(n log n)
SedgewickFast	(n log n)	n



Comparison between enhanced algorithms (Sorted Data)

	Ordered by average running time	Average number of comparison
Original	4	n log n
Quickersort	3	n log n
Singleton	2	n log n
SedgewickFast		n^2

Enhancement - Quick-Sort



Comparison between enhanced algorithms (Reverse Order)

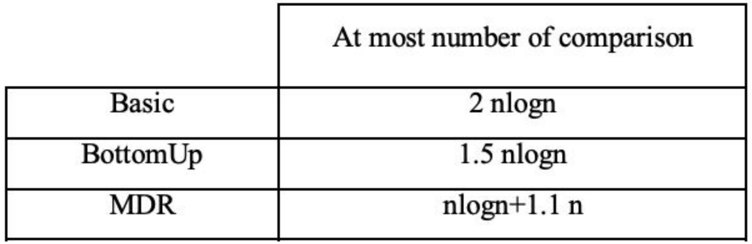
	Ordered by average running time	Average number of comparison(n log n)
Original	5	
Quickersort	3	4
Singleton	4	3
SedgewickFast	6	1

Enhancement - Heap-Sort

1. Bottom-up Heap-Sort
   a. No comparisons are made between a node and its children
   b. Binary insertion to find the path of maximum sons
      i. Let the larger of the sons of the root be in that list
      ii. Connect it with the path of maximum sons for that node
   c. The length of that path is at most log k.
2. MDR Heap-Sort
   a. Save even more comparisons by using old information.
   b. The possible values of info(j) are:
      i. Unknown, Left, Right (coded by 2 bits)
   c. We need info(j) only for j= 1,.., |_(n - 1)/2_|, since the other nodes do not have two sons. Hence, 2|_(n - 1)/2_| < n extra bits are sufficient.

Enhancement - Heap-Sort

Comparison between enhanced Heap-Sort algorithms



	At most number of comparison
Basic	2 nlogn
BottomUp	1.5 nlogn
MDR	nlogn+1.1 n

Conclusion

Conclusion - Summary

- Heap-Sort is typically somewhat slower than Quick-Sort, but its worst-case running time is always Θ(nlogn)
- Quick-Sort is usually faster, though its worst case performance can downgrade to O(n^2)
- There is no "ideal" solution about sorting
- Heap-Sort is a good choice if:
  1. you want to predict precisely when your algorithm will have finished
  2. your array is almost sorted

Conclusion - Weakness & Limitations

- Heap Sort is considered unstable, expensive, and not very efficient. Although it guarantees Θ(nlogn) complexity, in real-world implementation, there are constant factors that the theoretical analysis doesn't take into account.
- Quicksort will execute faster because its constant factors are smaller than the constant factors for Heapsort. (Partitioning is faster than maintaining the heap)
- Quick-Sort is recursive, fragile and unstable. It won’t preserve the order of elements while reordering. And it requires O(n^2) in the worst case, which drastically affects the performance of Quick-Sort.

What we learned from this project

- There is no perfect sorting algorithm, but there might be a most suitable sorting algorithm.
- The study and comparative study of different sorting algorithms is very worthwhile. It helps us understand different sorting algorithms better and apply them properly and accordingly.
- The research on sorting algorithms is always ongoing. New algorithms may arise from time to time, which combine the advantages of different algorithms to achieve better performance.
- Working on improving such sorting algorithms is needed in order to facilitate its applications work in a more efficient way to reduce both time and resources

Thanks for listening

Final Project: Comparison and evaluation of Heap-Sort and Quick-Sort Algorithms

CS 5800 Project
Yuanzhi Xu, Kejian Tong, Ting Ji

Introduction

I. Context

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step). A large number of sorting algorithms have been made in order to have best performance in terms of computational complexity (best, average and worst), memory usage, stability and method. One sorting algorithm can perform much better than another. The common sorting algorithms include quick sort, merge sort, heap sort, selection sort, insertion sort, bubble sort, etc. These sorting algorithms mentioned above are all comparison based, which are lower bounded by O(nlogn). Quick-Sort and Heap-Sort are two representative sorting algorithms which are widely used in computer science since they guarantee O(nlogn) complexity if implemented properly, which is the best performance we can achieve with comparison-based sorting. However, we got to know that most built-in sort algorithms these days are usually Quick-Sort.

II. Why we choose this topic

Since sorting can often reduce the complexity of a problem, it is an important algorithm and fundamental operation in Computer Science. Sorting algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.A large number of sorting algorithms have been made in order to have the best performance in terms of computational complexity (best, average, and worst), memory usage, stability and method. Quick-Sort and Heap-Sort are two representative sorting algorithms that are widely used in computer science since they guarantee O(nlogn) complexity if implemented properly, which is the best performance we can achieve with comparison-based sorting. However, we got to know that most built-in sorting algorithms these days are usually Quick-Sort. Why is Quick-Sort more popular than Heap-Sort, given that Quick-Sort's performance can downgrade to O(n^2) while Heap-Sort's complexity is always O(nlogn). We hope to look into this and see if we can find out the reason.

III. Defined questions

1. There are differences between Quicksort and heap sort. How much are they different?
2. Does one of them beat the other in some factors?
3. How do we evaluate the two algorithms according to the used method, stability, memory usage, and complexity?
4. Are there any enhancements to these algorithms?

IV. Why are each of you personally interested in those questions?

1. Yuanzhi Xu:

Learning arithmetic and mathematics since I was a child, I find that numbers are an indispensable part of our daily life. And people are never tired of inventing new algorithms for numbers, such as Fibonacci numbers, prime numbers, and so on. As we can collect a larger number of data, it is important to understand the connection and relations of the data. And sorting is one of the well-applicable algorithms in our daily life. It is easier and faster to locate items in a sorted list than unsorted. Sorting algorithms can be used in a program to sort an array for later searching or writing out to an ordered file or report. Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and more in the data-driven world. This concludes why I'm personally interested in sorting and prompts me to come up with these questions.

2. Kejian Tong

Before I started this course, algorithms were a black box to me and I didn't really realize how useful algorithms were in life. I was always curious about some things in daily life. For example, in real life, we tend to break things down along useful lines. If we were to sort the change, we would first divide the coins by denomination, then add each denomination, and then add them together. For another example, Commercial Computing is used in various government and private organizations for the purpose of sorting various data like sorting files by name/date/price, sorting of students by their roll no., sorting of account profile by given id, etc. Algorithms are also invisible codes that tell a computer how to accomplish a specific task. We can think of it as a recipe for a computer: an algorithm tells the computer what to do to produce a specific result. Every time we do a Google search or check the Facebook feed or navigate with GPS in the car, and we are interacting with the algorithm. With these questions and doubts, I am interested in understanding the algorithmic logic behind these life scenarios, why it is Quick Sort, and why it is sometimes Heap Sort.

3. Ting Ji

As a computer science major student, I have been introduced to many sorting algorithms such as selection sort, bubble sort, insertion sort, merge sort, quick sort and heap sort. However, I sometimes feel a bit confused about which sorting algorithm to use since there are so many sorting algorithms to choose from. Especially for Heap-Sort and Quick-Sort, although their average performances are both O(nlogn), why do commercial applications stick with Quick-Sort instead of Heap-Sort while Quick-Sort's worst-case performance is significantly worse than Heap-Sort? There must be some reason behind this. I think this is a very interesting problem to investigate. And I can learn a lot from this investigation, such as what Heap-Sort and Quick-Sort are, how they work, what their complexities are, how they are different from each other and when we favor one over the other.

Analysis

IV. QuickSort

1. *How it works*

   The idea is to choose a pivot element, and to compare each element to this pivot element. If an element is inferior or equal to the pivot element, this element is put on the left of the pivot element. If the element is strictly superior to the pivot element, it is put on the right of the pivot element. This operation is called partitioning and is recursively called until all the elements are sorted.

2. *Stability*
   Quicksort is not stable, which means Quicksort may change the relative order of records with equal keys.

3. *Memory Usage*
   Quicksort is an in-place algorithm (don't need an auxiliary array).

4. *Coding Algorithm*

```
QuickSort
1. If p < r then
2. q Partition (A, p, r)
3. Recursive call to Quick Sort (A, p, q)
4. Recursive call to Quick Sort (A, q + r, r)

PARTITION (A, p, r)
1. x ← A[p]
2. i ← p-1
3. j ← r+1
4. while TRUE do
5. Repeat j ← j-1
6. until A[j] ≤ x
7. Repeat i ← i+1
8. until A[i] ≥ x
9. if i < j
10. then exchange A[i] ↔ A[j]
11. else return j
```

5. *Complexity*
   The complexity of the quick-sort algorithm, for sorting an N elements array, is not always O(n log2 n). In fact, it can vary between (n log2 n and n^2 O(n log2 n)). The complexity of the quick-sort algorithm is essentially O(n log2 n). when the array‘s elements are not almost all sorted (assuming the pivot element is the first one of the array). The complexity is the same (close to a factor) for the heapsort algorithm. Quicksort takes O(n log2 n) time on average when the input is a random permutation.

   A smart approach is to set up a recurrence relation for the T(n) factor, the time needed to sort a list of size. In the most unbalanced case, a single Quicksort call involves O(n) work plus two recursive calls on lists of size 0 and n-1, so the recurrence relation is:
   T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).

   This is the same relation as for insertion sort and selection sort, and it solves to worst case T(n) = O(n^2). In the most balanced case, a single quicksort call involves O(n) work plus two recursive calls on lists of size n/2, so the recurrence relation is T(n) = O(n) + 2T(n/2). The master theorem tells us that T(n) = O(nlogn).

V. HeapSort

1. How it works

The idea is to look at the array to sort as it was a binary tree. The first element is the root, the second is descending from the first element. The aim is to obtain a heap, thus a binary tree verifying the following properties:

a) The maximal difference between the two leaves is 1 (all the leaves are on the last, or on the penultimate line).

b) The leaves having the maximum depth are ―heaped‖ on the left.

c) Each node has a bigger value than its two children, for an ascending sort.

2.Stability

HeapSort is not stable, that means heapsort may change the relative order of records with equal keys.

3. Memory Usage

HeapSort is in-place algorithm (don't need an auxiliary array).

4. Coding Algorithm

```
HEAPSORT (A)
1. BUILD_HEAP (A)
2. for i ← length (A) down to 2 do exchange A[1] ↔ A[i] heap-size [A] ← heap-size [A] - 1
Heapify (A, 1)

BUILD_HEAP (A)
1. heap-size (A) ← length [A]
2. For i ← floor(length[A]/2) down to 1 do
3. Heapify (A, i)

Heapify (A, i)
1. l ← left [i]
2. r ← right [i]
3. if l ≤ heap-size [A] and A[l] > A[i]
4. then largest ← l
5. else largest ← i
6. if r ≤ heap-size [A] and A[i] > A[largest]
7. then largest ← r
8. if largest ≠ i
9. then exchange A[i] ↔ A[largest]
10. Heapify (A, largest)
```

5. Complexity

The complexity of the heap-sort algorithm, for sorting a n elements array, is O(n log2n).

Let T(n) be the time to run Heapsort on an array of size n. Examination of the algorithms leads to the following formulation for runtime:

$T(n) = TB(n) + \sum_{k=1}^{n-1} TH(k) + \theta(n-1)$

TB is the time complexity of BuildHeap.

TH is the time complexity of Heapify.

Since Heapify is also used in Buildheap, we will attack it first:

$TH(n) = \theta(1) + TH(size\ of\ subtree)$

So we need to know how big the subtrees of a heap with n elements can be. A first guess might be that a subtree can only be half as big as the tree, since a heap is a binary tree. However, a little reflection shows that a better guess is 2/3:

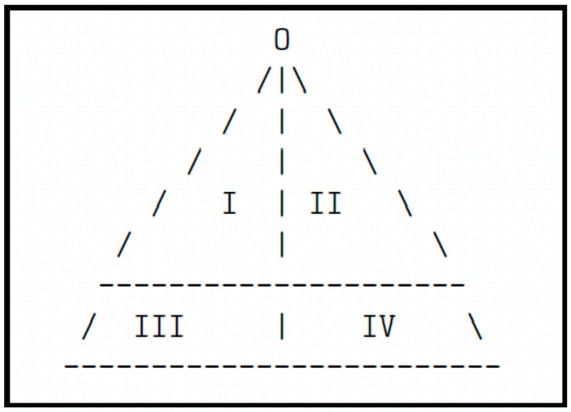


Figure 1. Binary tree

In a complete binary tree, each of the regions shown has approximately the same number of nodes. In a heap, region IV may be empty while region III is full. Since the left subtree consists of regions I and III, it has approximately 2/3 of the nodes in the heap.

This argument can be formalized. Let _ be the depth of heap A, and let n be the number of nodes in A.

$2^l <= n < 2^{l+1}$

The right subtree of A cannot be larger than the left subtree. A complete tree of height h has 2^{h+1} - 1 nodes. Regions I and II in Figure.1 each have 2^{l-1}-1 nodes. Region III has 2^{l-1} nodes, and size of the left subtree is $2^l - 1$

Let n = 2^l+k, where 0 <= k < 2^l. The left subtree contains 2^{l-1}+ j nodes and 0 <= j < 2^{l-1}.
If i = k - j, 0 <= i <= 2^{l-1}, then n = 2^l+ j + i and

$$\frac{size\ of\ left\ subtree}{n} = \frac{2^{i-1}+j}{2^i+j+i} \le \frac{2^{i-1}+j}{2*2^{i-1}+j} = f(j)$$

Let $\alpha = 2^{l-1}. Then$

f(j) = $\frac{\alpha+j}{2\alpha+j}$.

Extend f to [0, α] $and\ analyze$:

f(0) = $\frac{\alpha+0}{2\alpha+0} = \frac{1}{2}$

f(α) $= \frac{\alpha+\alpha}{2\alpha+\alpha} = \frac{2}{3}$

f(j) = $\frac{\alpha}{(2\alpha+j)^2} \geq 0$

So f is an increasing function that takes on a maximum at the left endpoint, and the max is 2/3. We state our result as a theorem: If a heap A has size n, its subtrees have size less than or equal to 2n/3. Applying this theorem to Equation (2), we have:

TH(n) = TH(2n/3) + $\theta(1)$

Using Case 2 of the Master Theorem, we have
TH(n) = θ(logn)

Buildheap
The complexity of Buildheap appears to be $\theta(nlogn) -$
$n\ calls\ to\ Heapify\ at\ a\ cost\ of\ \theta(logn)\ per\ call, but\ this\ result\ can\ be\ improved\ to\ \theta(n)$.
The intuition is that most of the calls to heapify are on very short heaps.

Putting our results together with Equation (1) gives us the following run-time complexity for Heapsort:

T(n) = TB(n) + $\sum_{k=1}^{n-1} TH(k) + \theta(n-1)$
$= \theta(n) + \sum_{k=1}^{n-1} logk + \theta(n-1)$
$= \theta(nlogn)$

VI. Comparisons table

The main characteristics of Quick sort and Heap sort are listed in the table bellow

		Quick Sort	Heap Sort
Method		Divide & Conquer (partition)	Selection
Stability		NOT stable	NOT stable
Memory usage		In place	In place
Complexity	Best	(n log n)	(n log n)
	Average	(n log n)	(n log n)
	Worst	(n^2)	(n log n)

VII. Enhancement

1. QuickSort

A. Quickersort

In 1965 Scowen developed a Quicksort algorithm called Quickersort in which the pivot is chosen as the middle key in the array of keys to be sorted. If the array of keys is already sorted or nearly sorted, then the middle key will be an excellent choice since it will split the array into two subarrays of equal size. Choosing the pivot as the middle key, the running time on sorted arrays becomes:

```
void  partition(int A[], int i, int j)

1. int pivot = A[(i+j)/2];
2. int p = i - 1;
3. int q = j +1;
4. for (;;)
5.        do q = q-1; while A[q] > pivot;
6.        do p = p-1; while A[p] < pivot;
7.        if (p < q)
8.                exchange (A[p], A[q]);
9.        else
10.               return q;
```

B. Singleton

In 1969, Singleton suggested the median-of-three method for choosing the pivot. One way of choosing the pivot would be to select the three elements from the left, middle and right of the array. Then, the three elements sorted and placed back into the same positions in the array. The pivot is the median of these three elements.

The median-of-three method improves Quicksort in three ways:

1. the worst case is much more unlikely to occur.
2. It eliminates the need for a sentinel key for partitioning, since this function is served by one of the three elements that are examined before partitioning.
3. It reduces the average running time of the algorithm by about 5%.

```
void  Singleton (long unsigned a[], int left, int right)

int i j;
long unsigned pivot;
if (left + CUTOFF <= right);
        pivot = median3 (a, left, right);
        i = left; j = right -1;
        for (;;)
                while (a[++i] < pivot);
                        while (a[--j] > pivot);
                                if (i < j);
                                        swap (i, j)
                                else
                                        break;
        swap (i, right -1);
        q_sort (a, left, i-1);
        q_sort(a, i+1, right);
```

Quicksort with three way partitioning has been suggested as the method for handling arrays with duplicate keys . Basically, the array of keys is partitioned into three parts: one part contains keys smaller than the pivot, the second part contains keys equal to the pivot, and the third part contains all keys that are larger than the pivot.

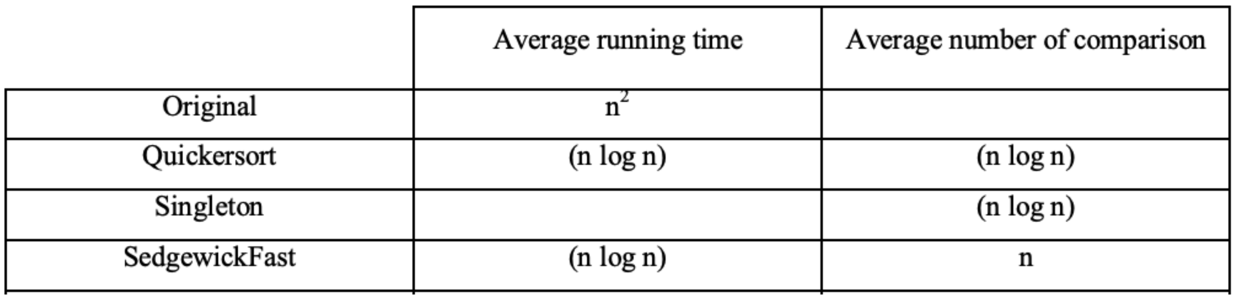
C. SedgewickFast

In 1978, Sedgewick suggested a version of Quicksort that minimizes the number of swaps by scanning in from the left of the array then scanning in from the right of the array then swapping the two numbers found to be out of position. This algorithm is called SedgewickFast which was declared to be a fast implementation of Quicksort. the algorithm makes comparisons, while for sorted data the number of comparisons is linear.

```
void  Partition (int low, int high, int& pos)

int i j;
int pivot;
i = low-1; j = high;
        pivot = ar[high];
for (;;)
        while (ar[++i] < pivot);
        while (pivot < ar[--j];
                if (j == low);
                        break;
                if (i > j)
                        break;
                swap(i, j);
swap (i, high);
pos = i;
```

D. Comparison between enhanced Quicsort algorithms

Comparison between enhanced algorithms (Random Data)



	Average running time	Average number of comparison
Original	n^2	
Quickersort	(n log n)	(n log n)
Singleton		(n log n)
SedgewickFast	(n log n)	n

Comparison between enhanced algorithms (Sorted Data)

	Ordered by average running time	Average number of comparison
Original	4	n log n
Quickersort	3	n log n
Singleton	2	n log n
SedgewickFast		n^2

Comparison between enhanced algorithms (Reverse Order)

	Ordered by average running time	Average number of comparison(n log n)
Original	5	
Quickersort	3	4
Singleton	4	3
SedgewickFast	6	1

2. HeapSort

A. Bottem-up Heapsort

A variant of Heap sort proposed by Carlson in 1987. In this improved version, no comparisons are made between a node and its children. Instead, a path of maximum sons is found, by letting the larger of the sons of the root be in that list, and connecting it with the path of maximum sons for that node. A binary insertion in such a path can be performed, since any path from the root to a leaf in a heap is an ordered list. The length of that path is at most [log k]. The only part of the HEAPSORT algorithm that has to be changed is the rearrangement procedure.

First, a path of maximum sons is found. Choose the element to be inserted as the root of the heap to be rearranged during the creation of the heap, and the last leaf during the sorting.

Then, a binary search for the place of the element, which has to be inserted in the path, is performed. This can easily be done after observing that any node j has the ancestors j div 2 k, where k is the length of the path between the node and its ancestor (j div 2k can be computed by shifting j k steps to the right).

Finally, all elements that are larger than the last leaf are moved one step up, and the element is inserted. Note that the path of maximum sons depends only on the current configuration of the heap and not on the value of the element, which is to be inserted. In the procedure HEAPSORT, the auxiliary procedure REARRANGE is at first used for the creation of the heap and then for the necessary rearrangement after removing the current maximum element from the root of the heap.

```
REARRANGE (A, i, n)

1. define j, k, NrOfLevels, Bot, Top, Mid: integer;
2. j = 2 * i;
3. // * * * LINEAR SEARCH * * *
4. while j < n do:
5.      if a[j] < A[j+1]
6.              j = 2 * (j + i);
7.      else
8.              j = 2 * j;
9.  if j > n
10.     j = j / 2; // j is now a leaf
11.  // * * * BINARY SEARCH * * *
12. NrOfLevels = log (j / i) // Number of levels between i and j
13. ToP = NrOfLevels
14. BoT = 0;
15. while Top > Bot do
16.     Mid = (Top + Bot) / 2;
17.     if x < A[j / 2^Mid] then
18.             Top = Mid;
19.     else
20.             Bot = Mid + 1;
21. // * * * INSERTION * * *
22. for k = NrOfLevels -1 down to Top dp
23. A[j / 2^(k+1) ] = A[j / 2^k]
24. A[j / 2^Top] = x


HEAPSORT( A , N )

1. // This procedure is similar to the usual HEAPSORT
2. define i : integer
3. define Temp: elementtype
4. / * * * HEAP CREATION * * *
5.  for i = N/2 down to 1 do
6.      REARRANGE(A, A(i), i, N);
7.  // * * * SORTING * * *
8.  for i = N-1 down to i do
9.      Temp = A[i + 1]
10.     A[i + 1] := A[i]
11.     REARRANGE(A, Temp, 1, i)
```

B. MDR-Heapsort

A variant of bottom-up heapsort has been presented by McDiarmid and Reed in 1989 which we call MDR-HEAPSORT. In many cases procedure reheap needs almost 2d comparisons, if d is the depth of the heap. We like to get by with approximately d comparisons. Therefore we use only 1 comparison at each vertex. With 1 comparison we can decide which son of the vertex just considered contains the smaller object and we go to this son. By this procedure it is not possible to stop at the correct position, since we do not consider the object at the root. Hence, we walk down a path in the heap until we reach a leaf. This path will be called special path, and the last object on this path will be called special leaf: This part of the algorithm is done by the procedure leaf-search, which returns the special leaf. The procedure Reheap of heapsort places the root object at some position p of the special path and all objects on the special path from the root to position p are shifted _into their father vertices.

Bottom-up heapsort and MDR-HEAPSORT look for the same position p. But these algorithms start their search from the special leaf and search bottom-up by the procedure bottomup search. Finally, the procedure interchange performs the data transport. Hence, all heapsort variants construct the same heaps. Let d be the length of the special path and let j be the length of the path from the root of the special path to the vertex at position p. The procedure reheap of HEAPSQRT needs 2 min { d, j + 1 > comparisons while the procedure Bottom-up reheap needs d+ min(d, d-j+ 1 } comparisons (see below). For large j we save almost half of the comparisons.

With MDR-HEAPSORT we save even more comparisons by using old information. MDR-HEAPSORT works with an extra array called info. The possible values of info(j) are: Unknown, abbreviated by u, Left, abbreviated by l, Right, abbreviated by r and can be coded by 2 bits.

The interpretation is the following.

1. If info(j) = i (or r), then the left son contains a smaller object than the right one (or vice versa).
2. If info(j) = u, nothing is known about the smaller son.

We need info(j) only for j= 1,.., |_(n - 1)/2_|, since the other nodes do not have two sons. Hence, 2|_(n - 1)/2_| < n extra bits are sufficient. The parameters are initialized as u. Now we are prepared to describe the procedure.

```
MDR-REHEAP (m , i)

1. LEAF-SEARCH (m, i)
2. BOTTOM-UP search (i, j).
3. INTERCHANGE (i, j)
```

LEAF-SEARCH(m , i)

```
1. j=i
2. while 2j < m do begin
3. if info(j) = l then
4.      J = 2j
5.      else if info(j) = r then
6.           j = 2j + 1
7.      else if a(2j) < a (2j+1) then
8.           info(j) = i
9.           J = 2j + 1,
10.          else info(j) = r
11.               J = rj + 1
12. if 2j = m then
13.     J = m
14. return j
```

BOTTOM-UP SEARCH(i, j) // j is the output of leaf-search

```
1. while (i<j and a(i)<a(j)) do
2.      j= |_ j/2 _|
3.  return
```

INTERCHANGE(i, j) // j is the output of bottom-up-search

```
1. l = bin(j) - bin(i)
2. x = a(i)|
3. for k= l – 1 down to 0 do
4.      a|_ j/$2^{k+1}$ _| = a|_ j/$2^k$ _|
5.      info(a|_ j/$2^{k+1}$ _|) = u
6. a(j): = x
```

C. Comparison between enhanced Heapsort algorithms

	At most number of comparison
Basic	2 nlogn
BottomUp	1.5 nlogn
MDR	nlogn+1.1 n

Conclusion

I. Summary of what we found

Heap-Sort is typically somewhat slower than Quick-Sort, but the worst-case running time is always Θ(nlogn). Quick-Sort is usually faster, though there remains the chance of worst case performance which can downgrade to O(n^2). There is no “ideal” solution about arrays’ sorting. If you want to predict precisely when your algorithm will have finished, it is better to use the heap-sort algorithm, because the complexity of this one is always the same. It is also the case if you know in advance that your array is almost sorted.

II. Weakness & Limitations

Heap Sort is considered unstable, expensive, and not very efficient when working with highly complex data.Although it guarantees O(nlogn) complexity, in real-world implementation, there are constant factors that the theoretical analysis doesn't take into account. Also, maintaining a heap is not free.

Given an array with a normal distribution, Quicksort and Heapsort will both run in O(nlogn). But Quicksort will execute faster because its constant factors are smaller than the constant factors for Heapsort. To put it simply, partitioning is faster than maintaining the heap.

Quick-Sort is recursive. If recursion is not available, the implementation is extremely complicated. Quick-Sort is fragile, a simple mistake in the implementation can go unnoticed and cause it to perform badly. Quick-Sort is also considered unstable as Heap-Sort. It won’t preserve the order of elements while reordering. Therefore, there is a possibility that two same elements in the unsorted list can be reversed when displaying the sorted list.And it requires quadratic time(O(n^2)) in the worst case, which drastically affects the performance of the Quick-Sort.

III. What we learned from this project

There is no perfect sorting algorithm. But there might be a most suitable sorting algorithm under a particular circumstance. The study and comparative study of different sorting algorithms is very worthwhile. It helps us understand different sorting algorithms better and apply them properly and accordingly.

The research on sorting algorithms (also other algorithms) is always ongoing. As a result, new algorithms may arise from time to time, which combine the advantages of different algorithms to achieve better performance. For example, QuickHeapSort is made by combining Quick-Sort and Heap-Sort. It sorts n elements in-place in O (n log n) average-case time and performs less moves as Heap-Sort algorithm does. Working on improving such sorting algorithm is needed in order to facilitate its applications work in a more efficient way to reduce both time and resources.

Reference:

[1] Thomas H. cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. (2001) Introduction to Algorithms. London: McGraw-Hill Book Company International Journal of Computing Academic Research (IJCAR), Volume 3, Number 2, April 2014 57

[2] Sharma, V. and Singh, S. (2008) Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms. IJCSNS International Journal of Computer Science and Network Security, VOL 8, No 4

[3] Muthusundari, S. Baboo, S. (2011) A Divide and Conquer Strategy for Improving The Efficiency and Probability Success In Sorting. International Conference on Advanced Computing, Communication and Networks.

[4] C. A. R. Hoare, "Quicksort," Computer Journal, 5, pp 10 – 15 1962.

[5] R.S. Scowen, "Algorithm 271: Quickersort," Comm. ACM 8, 11, pp 669-670, Nov. 1965..

[6] R. Sedgewick, Algorithms in C++, 3rd edition, Addison Wesley, 1998.

[7] R. L Wainright, ―Quicksort algorithms with an early exit for sorted subfiles,‖ Comm. ACM, 1987.

[8] W. Dobosiewicz, "Sorting by distributive partitioning" Information Processing Letters 7, 1 – 5., 1978.

[9] A. LaMarca and E. Ladner, "The Influence of Caches on the Performance of Sorting," Journal of Algorithms 31, 66-104 ., 1999.

[10] Williams, J.W.J., 1964. ACM algorithm 232: Heap sort. Commun. ACM., 7: 347-348.

[11] A. Levitin. Introduction to the design & Analysis of Algorithms. Pearson Education, 1st edition, 2008.

[12] S.Carlsson: A variant of HEAPSORT with almost optimal number of comparisons. Information Processing Letters, 24: 247-250, 1987.

[13] I.Wegner: BOTTOM-UP-HEAPSORT beating on average QUICKSORT (if n is not very small). Proceedings of the MFCS90, LNCS 452: 516-522, 1990

[14] I.Wegner: The worst case complexity of Mc diarmid and Reed's variant of BOTTOM-UP-HEAP SORT. Proceedings of the STACS91, LNCS 480: 137-147, 1991.

[15] Dutton R D. Weak Heap Sort. BIT, 1993, 33(3): 372-381.

[16] D. Cantone and G. Cincotti, "QuickHeapsort, an Efficient Mix of Classical Sorting Algorithms," presented at the Proceedings of the 4th Italian Conference on Algorithms and Complexity