# Sorting Algorithms

## Quicksort

**Why is quicksort often the best practical choice for sorting (two reasons)?**

* It is very efficient on average as its expected running time is
* It also sorts in place.

**What is the worst-case and average case running time of quicksort?**

in worst case, average is

**What is the space complexity of QuickSort?**

Recursive implementation:

**Does quicksort sort in place?**

Yes.

**What is the stability of Quicksort?**

It is not stable

**What strategy does quicksort use?**

Divide and conquer

**Describe the three step process that quicksort uses for sorting a typical subarray A[p..r]**

1. **Divide:** divide the Array A[p..r] into two (possibly empty) arrays consisting of A[p..q-1] and A[p+1..r] where each item in A[p..q-1] is less than equal to A[q] which is less than or equal to each item in A[q+1..r]. Compute the index q as part of this procedure.
2. **Conquer:** Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort
3. **Combine:** Because the subarrays are already sorted in-place there is no extra work required to combine the results the whole array A[p..r] is now sorted.

**What is the pseudocode for quicksort and what does the initial call look like to sort an entire array A?**

QUICKSORT (A, p, r)

if p < r

q = PARTITION (A, p, r)

QUICKSORT (A, p, q-1)

QUICKSORT (A, q+1, r)

Initial call to sort an entire array A is QUICKSORT (A, 1, A.length).

**What is the pseudocode for PARTITION?**

PARTITION (A, p, r)

x = A[r] *// select the pivot element*

i = p – 1 *// upper boundary of the less than region*

**for** j = p to r – 1 *// j defines the upper boundary of the greater than region*

**if** A[j] x *// compare the current element with the pivot*

i = i + 1 *// if current element is less than pivot, expand the less than region by incrementing i*

exchange A[i] with A[j] *// and exchange the values at i and j*

exchange A[i+1] with A[r] *// at the end, exchange the pivot with the value ahead of the less than region*

**return** i+1

**What are the four regions at the beginning of each iteration of the loop for any array index k?**

1. If (A[k] is less than the pivot element, **less than region**)
2. If (A[k] is greater than the pivot element, **greater than region**)
3. If (A[k] is equal to the pivot element, **equal to region**)
4. The indices between j and r-1 are not covered and have no relationship to the pivot. **Undefined region**.

**Identify the four regions in figure (d) and explain how PARTITION transforms the array from (d) to (e) and from (e) to (f)**

**Diagram

Description automatically generated**

The lightly shaded region is the elements which are less than the pivot (4). The heavily shaded region contain those elements which are greater than the pivot. The non-shaded region is the undefined region. j is incremented each time we loop and defines the upper boundary of the greater than region while i is only incremented when we add an item to the less than region. i defines the upper boundary of the less than region.

Partition works likes this. Select the pivot element to be the last element in the array. Then loop through each element in the array from p to r-1 and compare the currently selected element (j) to the pivot.

* If the selected element is less than or equal to the pivot, expand the less than region by incrementing i and exchange the current item with the item that is in the i slot after i has been incremented.
* If the selected element is greater than the pivot, don’t do anything. Move on to the next selected element.
* After we have iterated through the array from r to p-1, then swap the pivot with the element in the i+1 slot.

**What is the Java code implementation for quicksort and what language specific details do you need to remember?**

Particularly for the PARTITION method, remember to use a List<> for the array so that you can use the Collections.swap() method.

## MergeSort

**What is the running time of merge sort in the worst case and average case?**

Average and worst-case running time is

**What is the space complexity of merge sort?**

**What is the stability of Merge Sort?**

It is stable

**What strategy does Merge Sort use?**

Divide and conquer

**Does merge sort work in place?**

No.

**Describe the three-step process that merge sort uses for sort**

1. **Divide:** divide the n-element sequence into two sequences of length n/2
2. **Conquer:** Sort the two subsequences recursively using merge sort
3. **Combine:** merge the two sorted subsequences to produce the sorted answer.

**Describe how the MERGE operation works in terms of two decks of sorted cards**

We call MERGE (A, p, q, r) where A is an array to be sorted and p, q, r are indices into the array such at The procedure assumes that the subarrays A[p..q] and A[q+1..r] are already sorted and combines them into a single sorted array.

Suppose there are two decks of cards faced up on the table. The basic step in merging the two decks of cards is to take the smaller card of the two decks and place it in the output pile, exposing a new card on the deck the smallest card was removed from. We repeat this step until one pile is empty and simply put all the other cards on top of the output pile since they are already sorted.

**What is the pseudocode for the MERGE operation?**

MERGE(A, p, q, r)

= q – p + 1 *//length of left pile*

= r – q *//length of right pile*

Let L[1..+1] and R[1.. +1] be new arrays

For i = 1 to

L[i] = A[p+i-1] *//Copy values from A to left array*

For j = 1 to

R[j] = A[q + j] *//Copy values from A to right array*

L[+1] = *// Set sentinel value*

R[] = *// Set sentinel value*

i = 1

j = 1

for k = p to r *// p to r is the total number of items to merge*

if L[i] R[j] *// if the item in the left pile is smaller than in the right*

A[k] = L[i] *//add the left item to the output pile*

i = i + 1 *// and increment the left pile index*

else A[k] = R[j] *// else add the right item to the output pile*

j = j + 1 *//and increment the right pile index*

**What is the pseudocode for the whole MERGE-SORT algorithm?**

MERGE-SORT(A, p, r)

if p < r

q =

MERGE-SORT(A, p, q)

MERGE-SORT(A, q+1, r)

MERGE(A, p, q, r)

To sort the entire array A, we make the initial call MERGE-SORT(A, 1, A.length)

**Compare and contrast MergeSort with QuickSort. Why is QuickSort generally preferable to MergeSort? In what scenarios would MergeSort be a good choice?**

Quicksort sorts in place and thus requires less space. It is also very easy to avoid QuickSort’s worse case running time of by choosing the pivot randomly. Quicksort also has a small hidden constant compared to MergeSort. If data has to be sorted on disk, you really want to use some variation of MergeSort. MergeSort is worth considering if speed is important, bad worst-case performance cannot be tolerated, and extra space is available. Studies show that QuickSort is better for smaller datasets while MergeSort is better on larger datasets.

## Insertion Sort

**What is the worst-case and average case running time?**

Worst case:

Average case:

Best case:

**What is the space complexity?**

Insertion Sort uses no extra space and therefore has a space complexity of .

**Does quicksort sort in place?**

Yes

**What is the stability of InsertionSort?**

It is stable

**Describe how insertion sort works with a deck of cards.**

Our left hand is empty while the unsorted cards are facedown on the table. We sort the cards by drawing one card at a time and inserting it into the correct place in the left hand. To find the correct place to insert the card, we compare it with the other cards currently in the hand, from right to left.

**Explain how insertion sort in the following example works.**

**Diagram

Description automatically generated**

**What is the pseudocode of insertion sort?**

INSERTION-SORT(A)

for j = 2 to A.length *//The current card being inserted into the hand*

key = A[j]

i = j -1 *// Insert A[j] into the sorted sequence A[1.. j-1]*

while i > 0 and A[i] > key

A[i+1] = A[i]

i = i – 1

A[i +1] = key

**When is insertion sort better than QuickSort or MergeSort? When is it worse?**

Insertion sort is preferred with a **small set** to sort. It is also preferred when **data is sorted or nearly sorted** because it skips sorted values. Insertion sort is preferable to MergeSort when **space is a concern** as it sorts in place and has a time complexity of while MergeSort has a space complexity of .

Insertion sort is faster for small n when compared to QuickSort because QuickSort has extra overhead from the recursive function calls.

Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

## Heapsort & Heaps

**What is the worst-case, best case, and average case running time of quicksort?**

Average, best case, and worst case are all

**What is the space complexity?**

**Does it sort in place?**

Yes

**What is the stability of this algorithm?**

Not stable.

**What is a heap?**

A binary heap data structure is an array object that we can view as a nearly complete binary tree. The tree is completely filled at all levels except possibly the lowest, which is filled from left to right.

**What two attributes does an Array A have that represents a heap?**

A.length which gives the number of elements in the array and A.heap-size, which represents how many elements int eh heap are stored within array A. A.heap-size contains the valid elements of the heap. The root of the tree is A[0].

**What is the pseudocode for finding the parent and left and right children of a node i?**

PARENT(i)

return floor(i/2)

LEFT(i)

return 2i

RIGHT(i)

return 2i+1

**What are the two kinds of binary heap?**

Max heap and min heap

**What is the heap property for a max heap? For a min heap?**

Max heap property : A[PARENT(i)] >= A[i]. In other words, the value of a node can be no greater than its parent. The subtree rooted at a node contains values no larger than that contained at the node itself.

Min heap property: A[PARENT(i)] <= A[i]. The value of a node can be no smaller than its parent.

**Which of the two heap kinds do we use for heapsort?**

Max heap.

**What are min heaps typically used for?**

Min-priority queues.

**How is the height of a node in a heap defined? What about the height of the whole heap?**

The height of a node in a heap is defined as the number of edges along the longest simple path from a node to a leaf. We define the height of the heap to be the height of the root node.

**What is the height of a heap with n elements? Based on this, what can you say about the time operations take to complete on a heap?**

The height of an n-element binary tree is . The basic operations on a heap run in time proportional to the height of the tree and thus take time.

**What are the 7 common max heap procedures, what are their runtimes, and what do they do?**

1. MAX-HEAPIFY runs in time and is used to maintain the max-heap property
2. BUILD-MAX-HEAP runs in time and produces a max-heap from an unordered input array.
3. HEAP-SORT procedure runs in time sorts an array in place.
4. MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM which all run in time allows the heap data structure to implement a priority queue

**What is the pseudocode for MAX-HEAPIFY? Explain how it works. What assumption do we make about the index i’s children?**

**Text, letter

Description automatically generated**

Assumption is that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] might be smaller than its children, violating the max-heap property. MAX-HEAPIFY lets the value A[i] float down so that that the subtree rooted at index i obeys the max-heap property. MAX-HEAPIFY works by first determining if the node i or one of its two children are largest. If node i is the largest then the max-heap property is satisfied and no work is needed. However, if the left or right child is larger, then the node indexed by i is swapped with the larger of the two children. However, we must now recursively call MAX-HEAPIFY on the new subtree rooted at largest because it might not obey the max-heap property.

**Explain what the following diagram is doing when MAX-HEAPIFY(A, 2) is called.**

**Diagram, schematic

Description automatically generated**

**What is the recurrence relation for the running time of MAX-HEAPIFy and what is its solution?**

By case 2 of the master theorem . Alternatively, we can characterize the running time in terms of the height h as .

**What is the pseudocode for BUILD-MAX-HEAP? What is the running time?**

**Text, letter

Description automatically generated**

MAX-HEAPIFY costs and we make such calls so the total time is

**Explain how the following images are generated using the BUILD-MAX-HEAP procedure.**

**Diagram

Description automatically generated**

**What is the pseudocode for HEAPSORT? What is the runtime?**Text, letter

Description automatically generated

There are n-1 calls to MAX-HEAPIFY which each take time so the total time is .

**Explain how the following diagram demonstrates the HEAPSORT algorithm.**

**Shape

Description automatically generated**

## Bubble Sort

**What is the worst-case, best case, and average case running time? What is the space complexity?**

Average and worst case: . Because of this quadratic time complexity bubble sort is a poor sorting algorithm. Best complexity is . Space complexity is .

**What is the pseudocode for bubble sort?**

BUBBLESORT(A)

for j = 0 to A.length-1

for i = 0 to A.length-1

if A[i] > A[i + 1]

swap(A[i], A[i+1]

**How does bubble sort work?**

Bubble sort is a comparison-based algorithm where each pair of adjacent items are compared and are swapped if they are not in order. It requires n passes, each of which examine n elements, hence the time complexity.

**Is radix sort stable?**

Yes

**What class is radix sort?**

Comparison sort.

## Other sorting Algorithms

### Radix sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst Case** | **Average Case** | **Best Case** | **Space Complexity** | **Stability** | **In Place** | **Class** |
|  |  |  |  | Stable | No | Non-comparison |
| **Strengths** | | | | **Weaknesses** | | |
| * Linear time * Worked well for sorting punch cards | | | | Restricted inputs – radix sort only works when sorting numbers with a fixed number of digits. | | |

**What does radix mean?**

Radix is the base of a system of numeration. It is the number of unique digits, including zero, used to represent number in a positional numeral system.

**How does radix sort work?**

Generate d number of bins where d is equal to the base of the number system you are using. So if we are sorting decimal numbers d would be 10. In the first pass through the array to be sorted, look at the least significant digit in each number and based on that number place it in the corresponding bin. Then pull out each of the numbers from the bins in a FIFO fashion. Repeat this procedure for the second most significant digit and then the third. So each pass examines n elements and there are up to d passes for a time complexity of = . The space complexity is

### Counting Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst Case** | **Average Case** | **Best Case** | **Space** | **Stability** | **In Place** | **Class** |
|  |  |  |  | Stable | No | Non-comparison sort |
| **Strengths** | | | | **Weaknesses** | | |
| Linear time | | | | * Restricted inputs – only works when the range of potential items in the input is known ahead of time. * Space cost: if the range of potential values is big, then it requires a lot of space. | | |

**How does counting sort work?**

Counting sort works by iterating through the input, counting the number of times each item occurs, and using those counts to compute an item’s index in the final, sorted array.

### Bucket Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst Case** | **Average Case** | **Best Case** | **Space** | **Stability** | **In Place** | **Class** |
|  |  |  |  | Stable | No | Non comparison |
| **Strengths** | | | | **Weaknesses** | | |
| * Useful when input is uniformly distributed over a range * Useful if there are floating point values | | | | * Need to have a way of obtaining an index for each item to be sorted. Many objects do not have these. * Sensitive to the distribution of input values, so if you have tightly-clustered values, its not worth it. * Need to tune number of buckets. | | |

**How does bucket sort work?**

Bucket sort divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm. The process of bucket sort can be understood as scatter-gather approach, the elements are first scattered into buckets then the elements in each bucket are sorted. Finally the elements are gathered in order.

### Selection Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst Case** | **Average Case** | **Best Case** | **Space** | **Stability** | **In Place** | **Class** |
|  |  |  |  | Unstable | Yes | Comparison based |
| **Strengths** | | | | **Weaknesses** | | |
| Use selection sort in the following cases:   1. When the array is not partially sorted 2. Memory usage constraints 3. Simple implementation is desired 4. Array to be sorted is relatively small | | | | Avoid in these cases:   1. Array to be sorted is large 2. Array is nearly sorted 3. You want a faster runtime and memory is not a concern. | | |

**How does selection sort work?**

It selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

## Comparing Sorting algorithms

* **Quicksort** is a good default choice. It tends to be fast in practice, and with some small tweaks its dreaded O(n^2)*O*(*n*2) worst-case time complexity becomes very unlikely. A tried and true favorite.
* **Heapsort** is a good choice if you can't tolerate a worst-case time complexity of O(n^2)*O*(*n*2) or need low space costs. The Linux kernel [uses heapsort instead of quicksort](https://github.com/torvalds/linux/blob/master/lib/sort.c#L194)for both of those reasons.
* **Merge sort** is a good choice if you want [a stable sorting algorithm](https://www.interviewcake.com/concept/stable-sort). Also, merge sort can easily be extended to handle data sets that can't fit in RAM, where the bottleneck cost is reading and writing the input on disk, not comparing and swapping individual items.
* **Radix sort** *looks* fast, with its O(n)*O*(*n*) worst-case time complexity. But, if you're using it to sort binary numbers, then there's a hidden constant factor that's usually 32 or 64 (depending on how many bits your numbers are). That's often *way* bigger than O(\lg(n))*O*(lg(*n*)), meaning radix sort tends to be slow in practice.
* **Counting sort** is a good choice in scenarios where there are small number of distinct values to be sorted. This is pretty rare in practice, and counting sort doesn't get much use.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst Case** | **Average Case** | **Best Case** | **Space** | **Stability** | **In Place** | **Class** |
|  |  |  |  |  |  |  |
| **Strengths** | | | | **Weaknesses** | | |
|  | | | |  | | |

## Recursion

**What is the form of recurrence that the master theorem can be used on?**

Where we interpret n/b to mean either floor(n/b) or ceiling(n/b).

**What are the three cases for the master theorem?**

# Java Implementation

## QuickSort



## MergeSort



# Notes

* What does stability mean for sorting algorithms? Why is it important?
  + A sorting algorithm is stable if it preserves the order of equal items. It is important because you can’t stack unstable sorts.
* Know the following sorting algorithms:
  + Quicksort
  + **Merge sort**
  + **Insertion Sort**
  + **Heapsort**
  + **Radix/counting/bucket sort**
  + **Selection sort**
* For each algorithm know the following:
  + Conceptually how it works
  + Code implementation
  + Time complexity
  + Space complexity
  + Stability of results
* What are the use cases for insertion, bucket, counting, and radix sort?
* How does the heap sort work?
* Why shouldn’t you use bubble sort or selection sort?