Data Structures and Algorithms

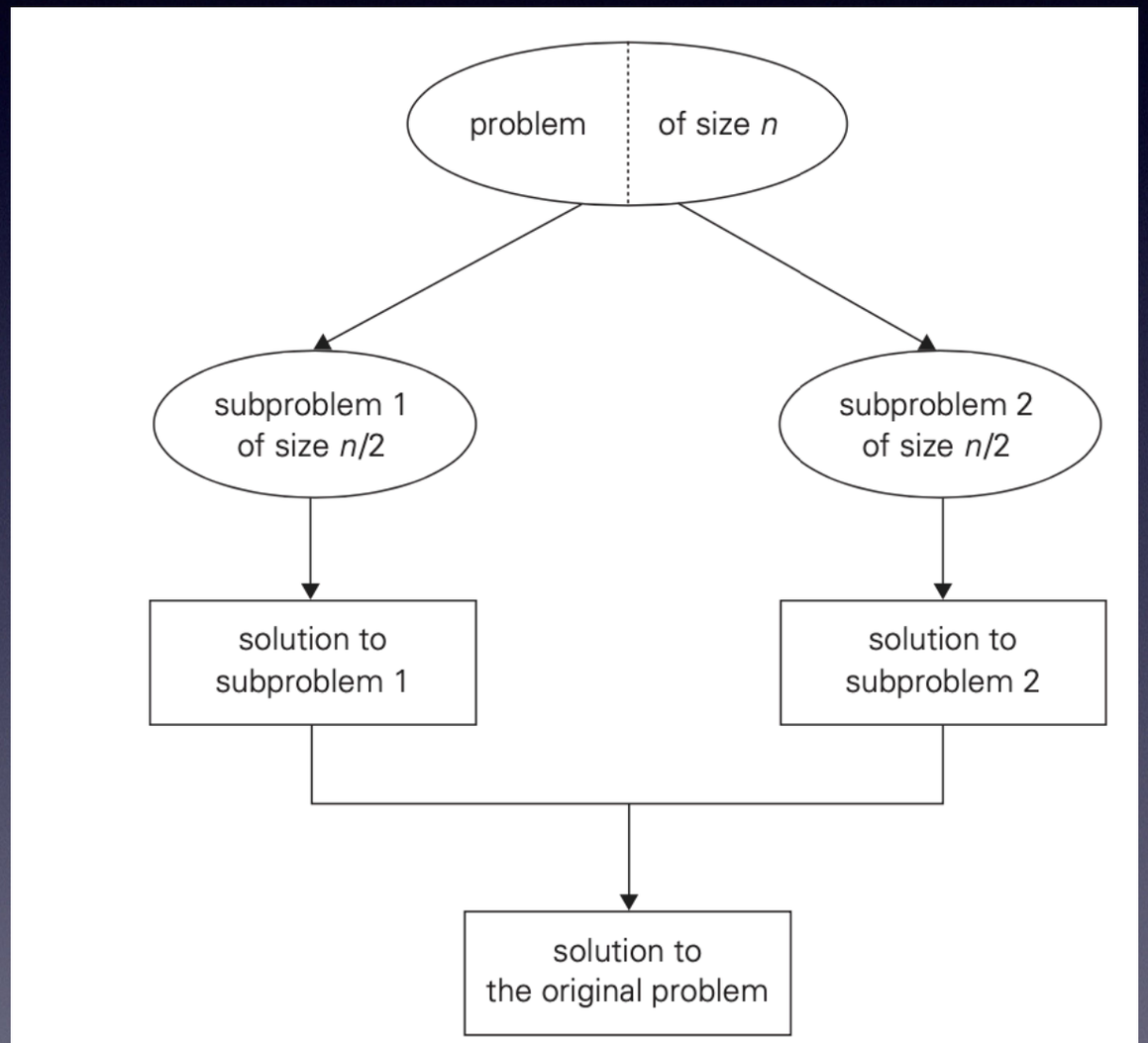
Divide-and-Conquer Approach / MergeSort / QuickSort / Comparison of Sorting Algorithms

Divide-and-Conquer Approach

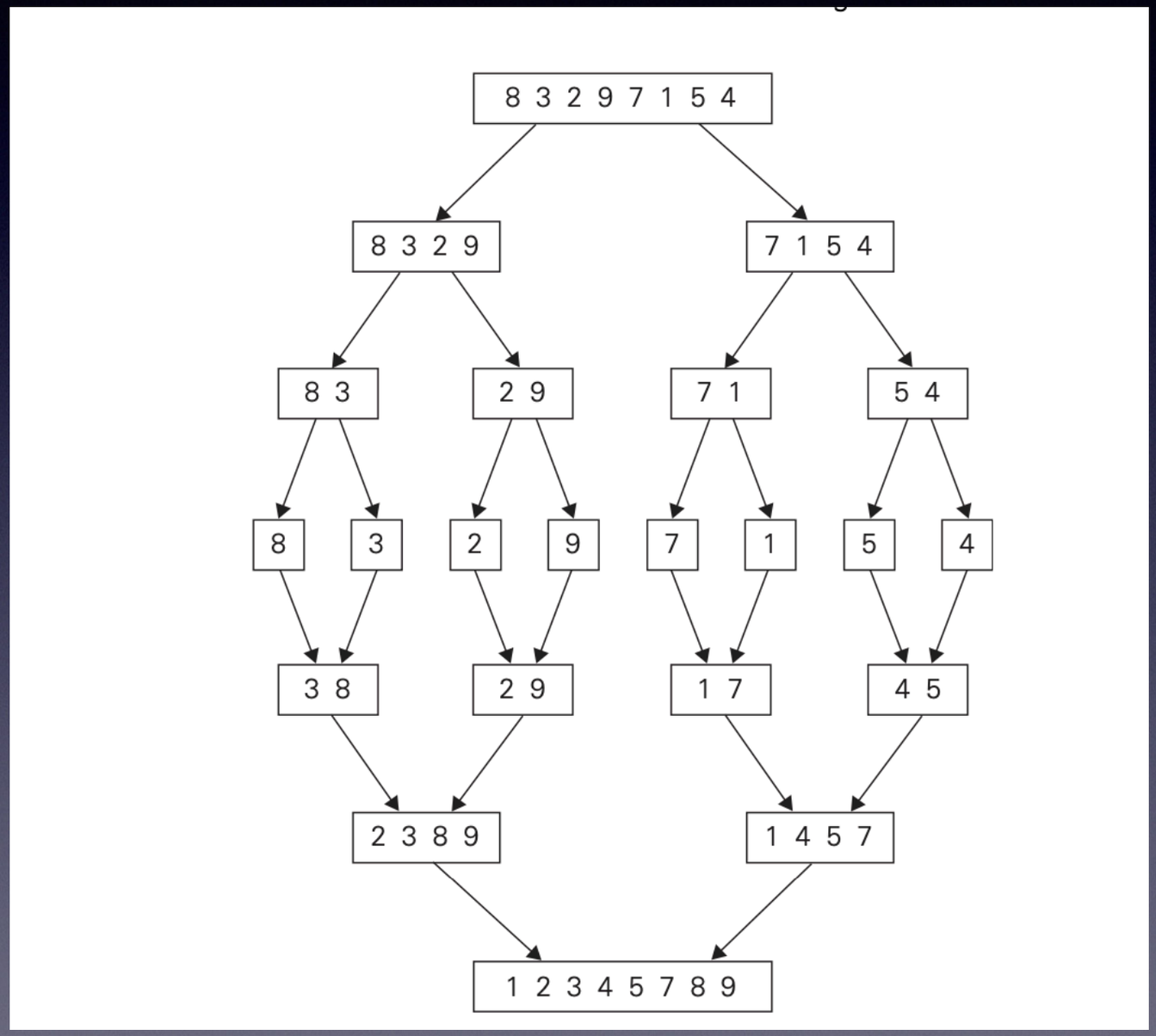
Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Divide-and-Conquer Approach



MergeSort


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

MergeSort - II

Mergesort has two steps: merging and sorting. The algorithm uses a divide-and-conquer approach to merge and sort a list.

Divide and conquer is a technique used for breaking algorithms down into subproblems, solving the subproblems, and then combining the results back together to solve the original problem. It can be helpful to think of this method as divide, conquer, and combine.

The mergesort algorithm focuses on how to merge together two pre-sorted arrays such that the resulting array is also sorted. Mergesort can be implemented either recursively or iteratively.

Here is the recursive mergesort algorithm:

1. If the list has only one element, return the list and terminate. (Base case)
2. Split the list into two halves that are as equal in length as possible. (Divide)
3. Using recursion, sort both lists using mergesort. (Conquer)
4. Merge the two sorted lists and return the result. (Combine)

Merge Step

The Merge Algorithm

Here is one way to implement merge:

1. Create an empty list called the result list.
2. Do the following until one of the input lists is empty: Remove the first element of the list that has a lesser first element and append it to the result list.
3. When one of the lists is empty, append all elements of the other list to the result.

Complexity of Merge

The merge function does a $O(1)$ (constant) number of operations for each element in the list. The list size is $O(n)$ since there are n elements. Merge does a constant amount of work $O(n)$ times, so merge runs in $O(n)$ time.

Radix Sort

- Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same **place value**. Then, sort the elements according to their increasing/decreasing order.

Complexity of MergeSort

Complexity of Mergesort

Division:
It takes $O(1)$ time to divide the problem into two parts. To divide the problem, the algorithm computes the middle of the list by taking the length of the list and dividing by two, which takes constant time.

Solving the subproblems:
Two equally large subproblems are produced. Each half takes $T\left(\frac{n}{2}\right)$ time, so solving the subproblems takes a total of $2\,T\left(\frac{n}{2}\right)$ time.

Combining the subproblems:
This is the merge step of mergesort. This step takes $O(n)$ time, as shown in the analysis of the merge algorithm.

Therefore,

$$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n).$$

The master theorem tells us that the solution to this recurrence is

$$T(n) = O(n \log n).$$

Mergesort runs in $O(n \log n)$ time in its best case, worst case, and average case. That means that no matter what the input, mergesort will operate in $O(n \log n)$ time.

(Dis)Advantages of MergeSort

Pros and Cons

Pros

- Time-efficient with time complexity of $O(n \log n)$
- Can be used for external sorting
- Highly parallelizable
- Stable sort

Cons

- Marginally slower than quicksort in practice
- Not as space-efficient as other sorting algorithms, e.g. block sort

Quick Sort

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \geq A[s]}$$

Quicksort

Quicksort uses divide and conquer to sort an array. Divide and conquer is a technique used for breaking algorithms down into subproblems, solving the subproblems, and then combining the results back together to solve the original problem. It can be helpful to think of this method as divide, conquer, and combine.

Here are the divide, conquer, and combine steps that quicksort uses:

Divide:

1. Pick a pivot element, $A[q]$.
2. Partition, or rearrange, the array into two subarrays: $A[p, \ldots, q-1]$ such that all elements are less than $A[q]$, and $A[q+1, \ldots, r]$ such that all elements are greater than or equal to $A[q]$.

Conquer: Sort the subarrays $A[p, \ldots, q-1]$ and $A[q+1, \ldots, r]$ recursively with quicksort.

Combine: No work is needed to combine the arrays because they are already sorted. [1]

Quick Sort Recursive Solution

Here is a recursive algorithm for quicksort:

1. If the list is empty, return the list and terminate. (Base case)
2. Choose a pivot element in the list.
3. Take all of the elements that are less than or equal to the pivot and use quicksort on them.
4. Take all of the elements that are greater than the pivot and use quicksort on them.
5. Return the concatenation of the quicksorted list of elements that are less than or equal to the pivot, the pivot, and the quicksorted list of elements that are greater than the pivot.

Selection of the Pivot Value

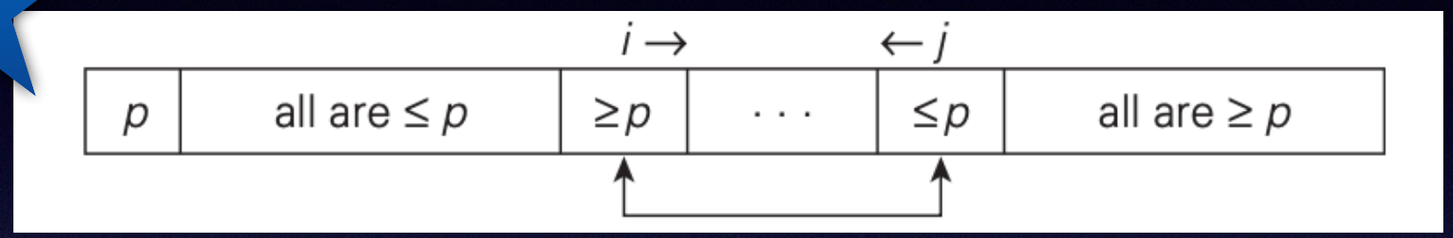
Choosing a Pivot

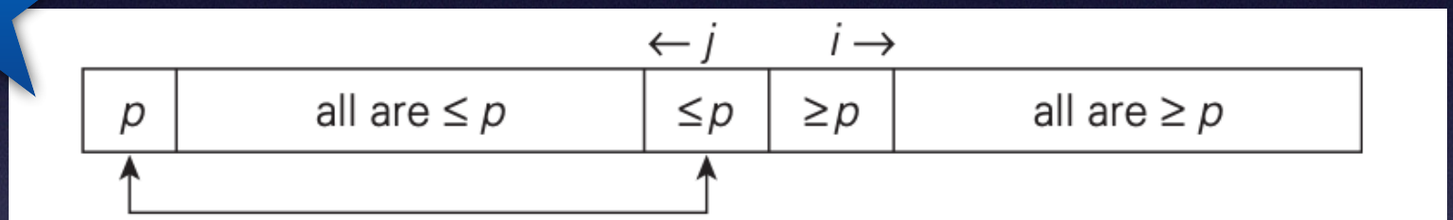
Picking a good pivot is the key for a fast implementation of quicksort; however, it is difficult to determine what a good pivot might be. The partitioning step takes time proportional to the number of elements being partitioned, so reducing the number of elements in each partition would give a faster runtime. The best-case pivot would divide the array into two equal parts, which would halve the problem size. However, this means that the pivot is the median of the elements, and in order to find the median, we would need an already sorted array. Since the goal of quicksort is *to sort* an array, we can't rely on having a pivot equal to the median of the elements.

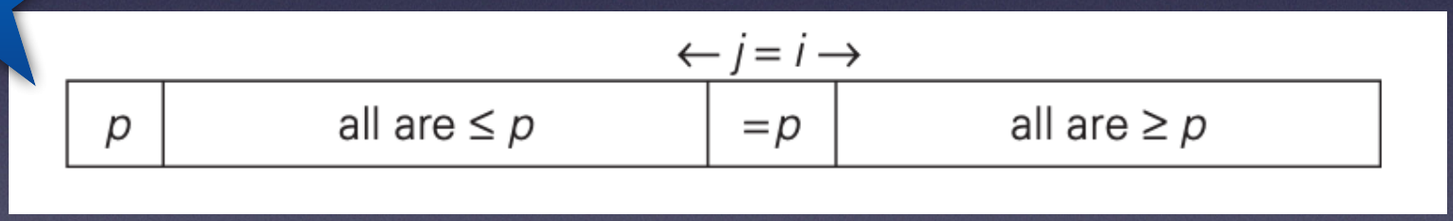
Here are some ways of choosing a pivot:

1. Select a random pivot.
2. Select the leftmost or rightmost element as the pivot.
3. Take the first, middle, and last value of the array, and choose the median of those three numbers as the pivot (median-of-three method).[2]
4. Use a median-finding algorithm such as the median-of-medians algorithm.

A Closer Look to QuickSort



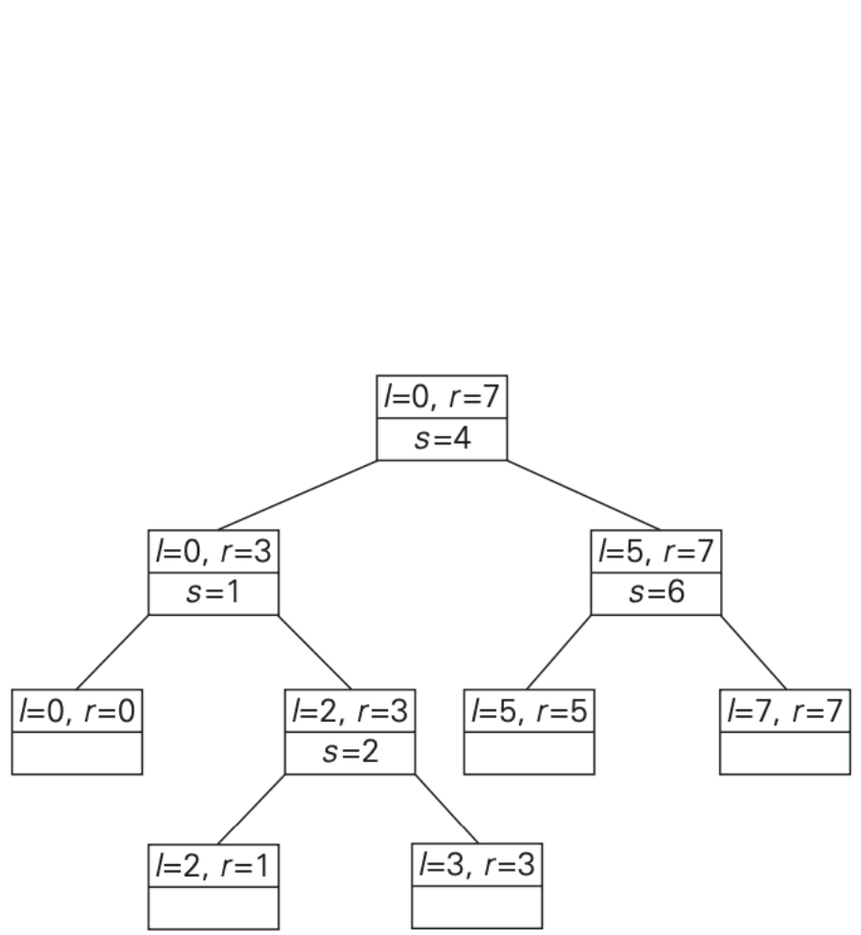




QuickSort Example

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
					i	*j*	
				8	9	7	
					i	*j*	
				8	7	9	
					j	*i*	
				8	7	9	
				7	**8**	9	
				7			
						9	

(a)



(b)

QuickSort

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$*Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Complexity of Quicksort

Complexity of Quicksort

Division: Dividing the list into parts less than and greater than the pivot takes $O(n)$ time because the algorithm needs to scan through the list, which has $O(n)$ elements. During this step, for each element, the algorithm performs a constant number of comparisons. In particular, it determines if the element is greater than or less than the pivot.

Subproblems:

- Worst Case
  In the worst case, all elements are either less than or greater than the pivot. In other words, if the pivot is the smallest or largest element of the array.
  In these cases, it takes $T(n-1)$ time to solve the subproblems because there will be $n-1$ recursive calls to the algorithm that creates subarrays of size 0 and $n-1$ and then at the next step creates subarrays of size 0 and $n-2$, and so on.
- Best Case
  The best case would be when both arrays are of the same length, in which case it would take $2T\left(\frac{n-1}{2}\right)$ to solve both of the subproblems.

Complexity of Quicksort

Combining: The subarrays are sorted in other steps of the algorithm, so there is no explicit, separate combining step.

Now, let's consider the following three analyses:

1. Best-case analysis
   The best case recurrence is

$$T(n) = 2\,T\left(\frac{n-1}{2}\right) + O(n).$$

   The master theorem tells us that the solution to this recurrence is

$$T(n) = O(n \log n).$$

   Quicksort will have a best-case running time when the pivot at each recursive call is equal to the median element of the subarray. This means that, at each step, the problem size is being halved, and the array can be sorted with $\log n$ nested calls. Each call takes $O(n)$ time (from the division step), so the total run time of the best-case quicksort is $O(n \log n)$.
2. Worst-case analysis
   The worst-case recurrence is

$$T(n) = T(n-1) + O(n).$$

   The master theorem tells us that the solution to this recurrence is

$$T(n) = O\left(n^2\right).$$

3. Average-case analysis

   This needs an explanation

   The expected run time of the algorithm is also

$$T(n) = O(n \log n).$$

Complexity of Quicksort

Recall that big O notation masks constant factors. While the average and best-case run time of quicksort is equal to that of other algorithms such as mergesort, a well-implemented quicksort will have much lower constant factors than other sorting algorithms. If two algorithms have the same asymptotic running time, the one with smaller constant factors will be faster. In practice, quicksort is often faster than mergesort.

Quicksort is usually implemented as an unstable sort with a best-case space complexity of $O(\log n)$ and an average-case space complexity of $O(n)$.

Comparison of Sorting Algorithms

Comparison Sorts

Comparison sorts compare elements at each step of the algorithm to determine if one element should be to the left or right of another element.

Comparison sorts are usually more straightforward to implement than integer sorts, but comparison sorts are limited by a lower bound of $O(n \log n)$, meaning that, on average, comparison sorts cannot be faster than $O(n \log n)$. A lower bound for an algorithm is the *worst-case* running time of the *best* possible algorithm for a given problem. The "on average" part here is important: there are many algorithms that run in very fast time if the inputted list is *already* sorted, or has some very particular (and overall unlikely) property. There is only one permutation of a list that is sorted, but $n!$ possible lists, so the chances that the input is already sorted is very unlikely, and on average, the list will not be very sorted.

Integer Sorts

Integer sorts are sometimes called counting sorts (though there is a specific integer sort algorithm called counting sort). Integer sorts do not make comparisons, so they are not bounded by $\Omega(n \log n)$. Integer sorts determine for each element x how many elements are less than x. If there are 14 elements that are less than x, then x will be placed in the 15^{th} slot. This information is used to place each element into the correct slot immediately—no need to rearrange lists.

Choosing a Sorting Algorithm

Algorithm	Best-case	Worst-case	Average-case	Space Complexity	Stable?
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$\log n$ best, n avg	Usually not*
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting Sort	$O(k + n)$	$O(k + n)$	$O(k + n)$	$O(k + n)$	Yes

Comparison of Sorting Algorithms

SORTING ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY
	BEST CASE	AVERAGE CASE	WORST CASE	WORST CASE
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N \log N)$
Radix Sort	$\Omega(N k)$	$\Theta(N k)$	$O(N k)$	$O(N + k)$
Count Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$	$O(k)$
Bucket Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$	$O(N)$