**BubbleSort (Stable)**

for (int i = n - 1; i >= 0; i--)   
 for (int j = 0; j < i; j++)  
 if A[j] > A[j + 1]  
 swap(A[j], A[j + 1])

Best case: Sorted array, O(n)

Average: Random array, O(n2)

Worst: Descending array, O(n2)

Invariant: At the end of jth iteration, the largest j elements are sorted in the last j positions of array

Correctness: After n iterations, sorted

Space: O(1)

**SelectionSort (Unstable)**

for (int i = 0; i < n; i++)   
 min = indexOfMinElement(A, i, n - 1)  
 swap(A[j], A[min])

Best case: Any array, O(n2)

Average: Any array, O(n2)

Worst: Any array, O(n2)

Invariant: At the end of jth iteration, the smallest j elements are sorted in the first j positions of array

Correctness: After n iterations, sorted

Space: O(1)

**InsertionSort (Stable)**

for (int i = 1; i < n; i++)   
 key = A[i]  
 j = i – 1  
 while (j >= 0 && A[j] > key) **[strictly larger than for stability]** A[j + 1] = A[j]  
 j = j – 1  
 A[j + 1] = key

Best case: Sorted, O(n)

Average: Random array, O(n2)

Worst: Descending array, O(n2)

Invariant: At the end of jth iteration, the first j elements in the array are sorted

Correctness: After n iterations, sorted

Space:

**MergeSort (Stable – depends on Merge)**

if (n == 1)  
 return  
else   
 left = MergeSort(A[0 … n/2], n/2)  
 right = MergeSort(A[n/2 + 1 … n - 1], n/2)  
 return Merge(left, right, n)

Best case: any array, O(n log n)

Average: any array, O(n log n)

Worst: any array, O(n log n)

Invariant: At the end of jth iteration, the first j elements in the array are sorted

Correctness: After n iterations, sorted

Recurrence: T(n) = 2T(n/2) + O(n)

Space: O(n log n)

**QuickSort (Unstable)**

if (n == 1)  
 return  
else   
 pivot = partition(A[0 … n – 1], n)  
 small = QuickSort(A[0 … p – 1, p - 1)  
 large = QuickSort(A[p + 1 … n - 1], n - p)  
 return Merge(left, right, n)

partition(A[1… n - 1], n, pivotIndex) [all partitioning is unstable]

pivot = A[pivotIndex];  
swap(A[1], A[pivotIndex]);  
low = 1;  
high = n - 1;  
while (low < high)  
 while (A[low] < pivot && low < high)  
 low++;  
 while (A[high] > pivot && low < high)  
 high--;  
 if (low < high)   
 swap(A[low], A[high]);  
swap(A[1], A[low - 1]);  
return low – 1;

Best case: Sorted, O(n log n)

Average: Random array, O(n log n)

Worst: Constant array, O(n2)

Partition invariant: For every 1 < i < low, A[i] < pivot. For every j >= high, A[j] > pivot.

Quicksort invariant: A[high] > pivot at the end of each loop.

Correctness: After n iterations, sorted

binSearch(A, key, n)  
begin = 0  
end = n – 1  
while (begin < end)  
 mid = begin + (end - begin) / 2  
 if key <= A[mid]  
 end = mid  
 else   
 begin = mid + 1  
return A[begin]

CountingSort: O(n+k)

Create array C to count key occurrences in A: O(n)

construct sorted array by iterating through C and insert each element j a total of C[j] times: O(k)

To make it stable, modify C into cumulative count. Iterate through A backward; for each element i, we check C[i] to know how many elements before it, so we know the position to slot in. Decrement C[i] after inserting.

Select(A[1…n], k)

If (n == 1) return A[1]  
pIndex = …  
p = partition(A[1…n, n, pIndex])  
if (k == p) return A[p]  
else if (k < p) return Select(A[1… p-1], k)  
else if (k > p) return Select(A[p+1…n], k)

AVL trees

Invariant: if u and v are siblings, then |height(u) – height(v)| < 2

If u is parent of v, then |heigh(u) – height(v)| > 0

Monotonic stacks

Stack contains index, not the value

Outer for loop, inner while loop

While condition is stack not empty and OPERATOR(<,>, <=, >=)

Next/prev greater use decreasing, next/prev smaller use increasing