Counting sort

The algorithm works only for positive integers.

1. Find the max in array **a** (*call it k*). **(Θ(n))**
2. Make a zero array with the size of k+1, and call it **hist** (*it shows the histogram of the elements in a*)
3. For each element *e* in **a**, increment hist[e] by 1. **(Θ(n))**
4. Set hist[i] = hist[i] + hist[i-1] for all 1<i<k+1 **(Θ(k))**
5. Make another array with the size of a and call it **sortedA**
6. Take each element *e* in **a:**
    a. Decrement hist[e] by 1.
    b. Put *e* in index hist[e] of sortedA ➔ sortedA[hist[e]]=e. **(Θ(n))**

The running time is Θ(n+k)

Radix sort

1. Start from the least significant digit (the right most digit) to the most significant digit (the left most digit), and sort the numbers based on their i^{th} digit.

The running time is Θ(kn)

Insertion sort

1. Start from the first element in the array.
2. Go to the next element (*i*).
3. Start moving the element to the left to its appropriate location (k) (*where element i is bigger than the k-1 element and smaller than k+1 element*)
4. Go back to step 2 until i=n (*n=size of the array*)

The best case running time is O(n) (The input is an array that is already sorted.)

The worst case running time is O(n^2) (The input is an array sorted in reverse order.)
The average running time is O(n^2)

Bubble sort

1. Compare each pair of adjacent elements from the beginning of the array and swap them if $a_i > a_{i+1}$. (*to find the largest element*)
2. Go back to step 1 until no swaps are needed (*It means the array is sorted*)

The best case running time is O(n) (The input is an array that is already sorted.)

The worst case running time is O(n^2) (The input is an array sorted in reverse order.)
The average running time is O(n^2)

Selection sort

1. Find the lowest element in the array (*it requires scanning n elements (n= size of the array)*)
2. Swap it into the k^{th} position (*k=0,1,2,…,n-1*)
Step k. Go back to step 1 until the array is sorted.

The running time is $\Theta(n^2)$

Merge sort

Read the divide and conquer algorithm first.

Divide: Divide the array into two halves **($\Theta(1)$)**
Conquer: Recursively sort the two sub-problems, each of size n/2, (contributes **2T(n/2)** to the running time.) (Recursively = repeat this step until the size of the sub-arrays are 1)
Combine: Combine the sub-problems by merging the two sub-arrays into a sorted array. **($\Theta(n)$)**

Running time of the algorithm: $T(n)= 2T(n/2)+\Theta(n)$

The best case running time is O(n logn)

The worst case running time is O(n logn)
The average running time is O(n logn)

Quicksort

Read the divide and conquer algorithm first.

Most efficient sorting algorithm for arrays of data stored in local memory

Divide:

1. Find the pivot (use median-of-three algorithm = **$\Theta(1)$**, **or** use the fast median search algorithm= **O(n)**)

2. Swap the pivot with the last element of the array **($\Theta(1)$)**

3. For the remaining elements of the array a[0]..a[n-2], define 2 markers: Left and Right. Left and Right markers start from the left side (a[0]) and the right side (a[n-2]) of the array respectively and move toward the center.

4. Marker Left stops if element a[i]>pivot and Marker Right stops if element a[i]<pivot.

5. When both stop, they swap the elements.

6. The process is done when both cross one another. **(Steps 3 to 6= $\Theta(n)$)**

7. Now divide the array into two sub-arrays $\mathbf{a}_{\mathbf{left}}$ (a[0], a[1], . . . , a[k-1]), and $\mathbf{a}_{\mathbf{right}}$ (a[k+1], . . . , a[n−1]) (where aj ≤ pivot for every j ≤ k − 1, and aj ≥ pivot for every j ≥ i) **(Θ(1))**
   (**Note:** now you have a_{left}, pivot, a_{right})

Conquer: Recursively solve two sub-problems (each of size n/2 *if pivot=median of the array*) (contributes **2T(n/2)** to the running time.) (Recursively = repeat this step until the size of the sub-arrays are 5 or fewer, then sort array using insertion sort)
Combine: Combine the sub-problems by concatenating the aleft, pivot, aright. **(Θ(1))**

Running time of the algorithm: T(n)= 2T(n/2)+Θ(n)

The best case running time is O(n logn)

The worst case running time is $O(n^2)$ (This rarely happens**)**
The average running time is O(n logn)

Divide and Conquer algorithms

Many problems can be solved recursively like Binary Search, Mergesort, Quicksort, Maximum Subsequence Sum (finding the maximum sum of any subsequence in a sequence of integers), Order Statistics (finding the k th least or greatest element of an array), Matrix Operations: (matrix inversion, Fast-Fourier Transform, matrix multiplication).

- **Divide**. Divide the original problem into one or more sub-problems that are smaller size.
- **Conquer**. Recursively solve the sub-problems until their sizes are small enough, and then just solve them in a straightforward manner
- **Combine.** Combine the solution to the sub-problems into a final solution for the original problem.

The running time of the divide and conquer algorithms is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \ , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \ . \end{cases}$$

T(n) = the running time on a problem of size n
a = the number of sub-problems
n/b = the size of each sub-problem
D(n) = the running time to divide the problem into sub-problems (the running time= the number of steps)
C(n) = the running time to combine the solutions to the sub-problems into the solution to the original problem
Note: If the size of the problem is c (for some small constant c), it takes a constant time to solve it in a straightforward manner. **(Θ(1))**

Note: Instead of writing **C(n)+D(n),** you can write **f(n)** as the running time to divide and combine the problems