Chapter 6

Sorting

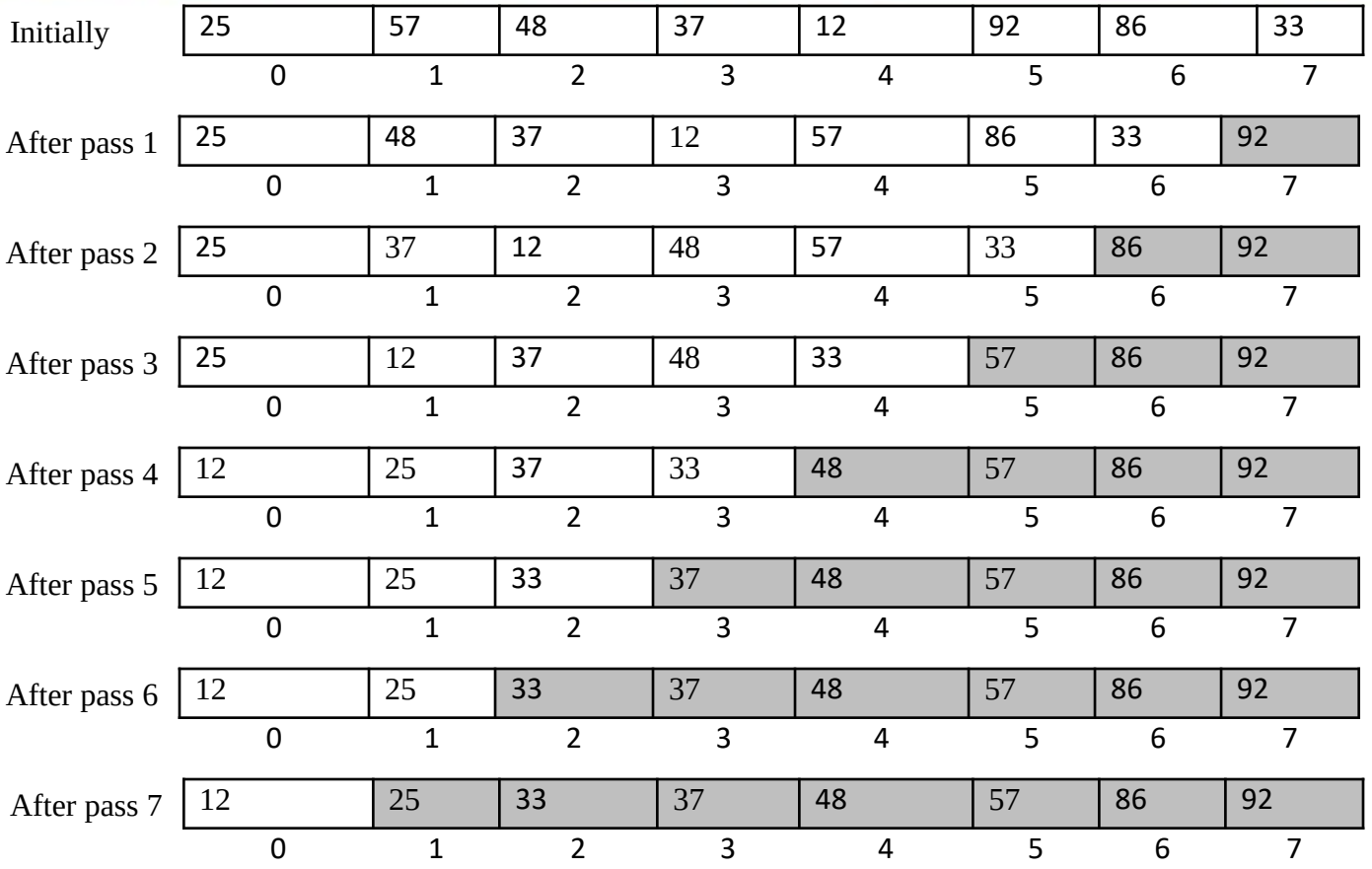
Introduction

- The process of rearranging data into ascending (smallest to largest) or descending (largest to smallest) order is called sorting
- Sorting is important because sorted data can be searched and merged efficiently.
- A sort can be classified as being **internal** if the records that it is sorting are in main memory, or **external** if some of the records that it is sorting are in auxiliary storage. Here, we restrict our attention to internal sorts.
- **Choice of algorithms:** *bubble sort*, *selection sort*, *insertion sort*, *shell sort*, *merge sort*, *quick sort*, *heap sort*

Bubble Sort

- The basic idea of this sort is to pass the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example a[i] with a[i + 1]) and interchanging the two elements if they are not in the proper order.
- After each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, n – 1 passes are sufficient to sort n elements.
- This method is called the bubble sort because each number slowly "bubbles" up to its proper position.

Bubble Sort



- In the above example, the array was sorted after 5 iterations making last two iterations unnecessary. To eliminate unnecessary passes we can keep a record of whether or not any interchanges are made in a given pass.

Bubble Sort

- **Analysis:**
  - The number of comparisons in first iteration is n-1, second iteration is n-2 and so on.
  - There are total n-1 iterations.
  - Total number of comparisons is (n-1) + (n-2) + … + 2 + 1 = $(n^2 - n)/2$.
  - Hence, time complexity is $O(n^2)$.
  - This sorting technique takes little additional space and hence its space complexity is O(1).

Bubble Sort

- **C - Function:**

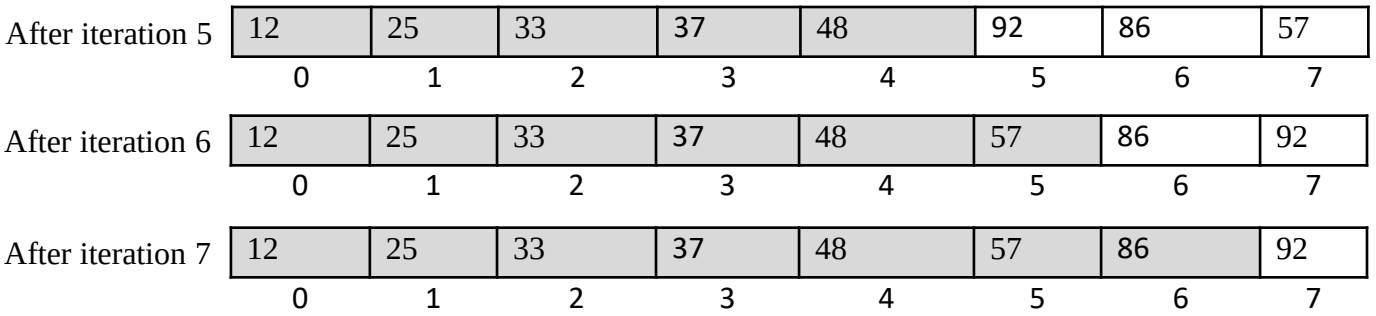
```
void bubblesort(int a[], int n) {
        int pass, i, temp, flag = 1;
        for(pass = 0; pass < n-1 && flag == 1; pass++) {
                flag = 0;
                for(i = 0; i < n-1-pass; i++) {
                        if(a[i] > a[i+1]) {
                                flag = 1;
                                temp = a[i];
                                a[i] = a[i+1];
                                a[i+1] = temp;
                        }
                }
        }
}
```

Selection Sort

- The basic idea of this sort is to find the least (for ascending order) value in the array and swap it with the leftmost value and then forget the leftmost value. Repeat the same process with the remaining components to its right.
- When only one component remains to be sorted, we are finished.

	0	1	2	3	4	5	6	7
Initially	25	57	48	37	12	92	86	33
After iteration 1	12	57	48	37	25	92	86	33
After iteration 2	12	25	48	37	57	92	86	33
After iteration 3	12	25	33	37	57	92	86	48
After iteration 4	12	25	33	37	57	92	86	48

Selection Sort



- **Analysis:**
  - The number of comparisons in first iteration is n-1, second iteration is n-2 and so on.
  - There are total n-1 iterations.
  - Total number of comparisons is (n-1) + (n-2) + … + 2 + 1 = $(n^2 - n)/2$.
  - Hence, time complexity is $O(n^2)$.
  - This sorting technique takes little additional space and hence its space complexity is O(1).

Selection Sort

C - Function:

```
void selectionsort(int a[], int n) {
        int i, j, p, least;
        for(i = 0; i < n-1; i++) {
                p = i;
                least = a[p];
                for(j = i+1; j < n; j++) {
                        if(a[j] < least) {
                                p = j;
                                least = a[p];
                        }
                }
                if(p != i) {
                        a[p] = a[i];
                        a[i] = least;
                }
        }
}
```

Insertion Sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Initially, the first element in the array is assumed to be sorted. Pick the next element in the array and insert it into its correct sorted position in the sorted part of the array. Repeat this process until the array is sorted.
- For an array of size n, n-1 iterations are needed to sort the array.

Insertion Sort

Initially	25	57	48	37	12	92	86	33
	0	1	2	3	4	5	6	7
After iteration 1	25	57	48	37	12	92	86	33
	0	1	2	3	4	5	6	7
After iteration 2	25	48	57	37	12	92	86	33
	0	1	2	3	4	5	6	7
After iteration 3	25	37	48	57	12	92	86	33
	0	1	2	3	4	5	6	7
After iteration 4	12	25	37	48	57	92	86	33
	0	1	2	3	4	5	6	7
After iteration 5	12	25	37	48	57	92	86	33
	0	1	2	3	4	5	6	7
After iteration 6	12	25	37	48	57	86	92	33
	0	1	2	3	4	5	6	7
After iteration 6	12	25	33	37	48	57	86	92
	0	1	2	3	4	5	6	7

Insertion Sort

- **Analysis:**
  - The maximum number of comparisons in first iteration is 1, second iteration is 2 and so on.
  - There are total n-1 iterations.
  - Total number of comparisons is 1 + 2 + … + (n-1) = $(n^2 - n)/2$.
  - Hence, time complexity is $O(n^2)$.
  - If the initial array is sorted, only one comparison is made on each pass, so that the sort is O(n).
  - This sorting technique takes little additional space and hence its space complexity is O(1).

Insertion Sort

- **C - Function:**

```
void insertionsort(int a[], int n)
{
        int i, k, y;
        for (k = 1; k < n; k++)
        {
                y = a[k];
                for (i = k - 1; i >= 0 && y < a[i]; i--)
                        a[i+1] = a[i];
                a[i+1] = y;
        }
}
```

Shell Sort

- Significant improvement on simple insertion sort can be achieved by using shell sort (or diminishing increment sort).
- This method separates original file into subfiles. These subfiles contain every k^{th} element of the original file. The value of k is called an increment. For example, if k = 5, then subfile consisting of x[0], x[5], x[10],……. is first sorted. Other four subfiles each containing one fifth of the elements of the original file are also sorted.

  Subfile 2: x[1], x[6], x[11],…….

  Subfile 3: x[2], x[7], x[12],…….

  Subfile 4: x[3], x[8], x[13],…….

  Subfile 5: x[4], x[9], x[14],…….

Shell Sort

- After the first k subfiles are sorted (usually by simple insertion), a new smaller value of k is chosen and the file is again partitioned into new set of subfiles. Each of these larger subfiles is sorted and the process is repeated yet again, until eventually the value of k is set to 1.
- The interval between elements is reduced based on the sequence used. Here we use Shell's original sequence, that is, n/2, n/4, …, 1.
- Suppose we need to sort the following array.

Initially

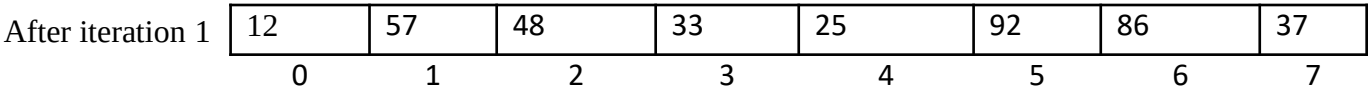
25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

- In the first loop, the elements lying at interval n/2 = 8/2 = 4 are sorted

Shell Sort

Subfiles	Sorted subfiles
25, 12	12, 25
57, 92	57, 92
48, 86	48, 86
37, 33	33, 37

- After rearranging all elements at n/2 interval the array becomes



- In the next iteration, the interval of n/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.

Subfiles	Sorted subfiles
12, 48, 25, 86	12, 25, 48, 86
57, 33, 92, 37	33, 37, 57, 92

Shell Sort

- After rearranging all elements at n/4 interval the array becomes

After iteration 2

12	33	25	37	48	57	86	92
0	1	2	3	4	5	6	7

- Finally, when interval is $n/8 = 8/8 = 1$, we use simple insertion to sort the array as given below.

12	33	25	37	48	57	86	92
0	1	2	3	4	5	6	7

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Shell Sort

- **Analysis:**
  - Worst case time complexity is $O(n^2)$.
  - Best case time complexity is $O(n\log_2 n)$.
  - This sorting technique takes little additional space and hence its space complexity is O(1).

Shell Sort

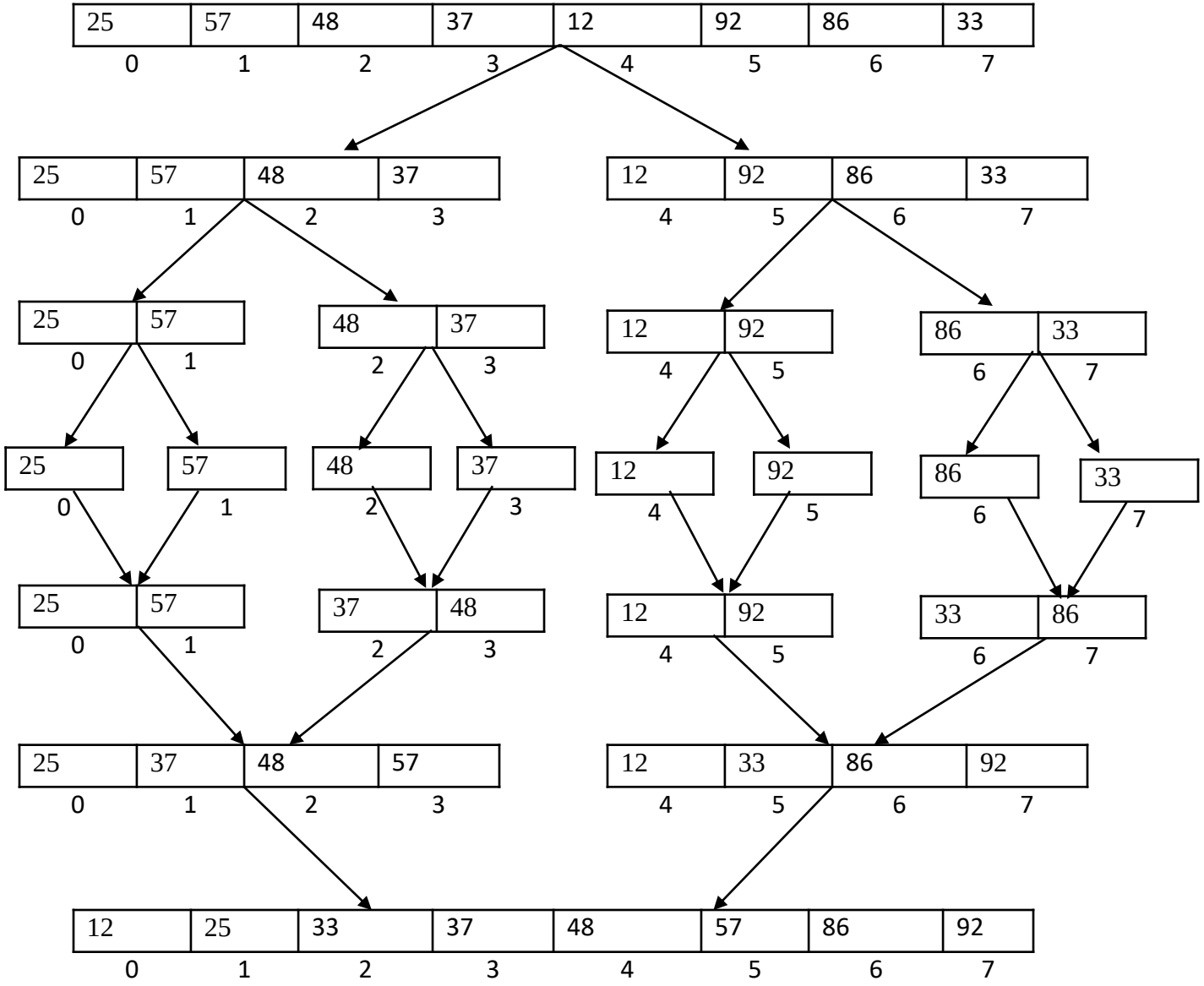
- **C - Function:**

```
void shellsort(int a[], int n)
{
        int interval, i, j, temp;
        for(interval = n/2; interval > 0; interval /= 2)
        {
                for(i = interval; i < n; i += 1)
                {
                        temp = a[i];
                        for(j = i; j >= interval && a[j-interval] > temp; j -= interval)
                                a[j] = a[j-interval];
                        a[j] = temp;
                }
        }
}
```

Merge Sort

- Merge sort is based on **divide-and-conquer** strategy.
- **Divide-and-conquer strategy**
  - To solve a 'hard' problem, break it down into two or more 'easier' subproblems, solve these subproblems separately, and combine their answers.
  - The divide-and-conquer strategy is not suitable for all problems, but it does work very well for sorting.
- **Merge sort:** If we are required sort an array of size greater than one, we can divide the array into two subarrays of about equal length, recursively sort each subarray separately, and finally merge these two subarrays.

Merge Sort


25 57 48 37 12 92 86 33
0 1 2 3 4 5 6 7
25 57 48 37
0 1 2 3
12 92 86 33
4 5 6 7
25 57
0 1
48 37
2 3
12 92
4 5
86 33
6 7
25
0
57
1
48
2
37
3
12
4
92
5
86
6
33
7
25 57
0 1
37 48
2 3
12 92
4 5
33 86
6 7
25 37 48 57
0 1 2 3
12 33 86 92
4 5 6 7
12 25 33 37 48 57 86 92
0 1 2 3 4 5 6 7

Merge Sort

- **Analysis:**
  - Let the total no. of comparisons required to sort n values be $f(n)$.
  - After dividing the array, the left subarray's length is about $n/2$ and right subarray's length is also about $n/2$. Hence, we need about $f(n/2)$ comparisons to sort each subarray. To merge these two sorted subarrays, we need about n comparisons. Hence,

    $f(n) \approx 2\,f(n/2) + n$ if $n > 1$

    $f(n) = 0$ if $n \leq 1$

    Hence,

    $f(n) \approx n \log_2 n$

    Time complexity is $O(n \log n)$.
  - Space complexity is $O(n)$, since merging needs an auxiliary array of length n.

Merge Sort

- **C - Functions:**

```
void merge(int a[], int left, int mid, int right)
{
        int i, j, k;
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int leftarray[n1], rightarray[n2];
        for(i = 0; i < n1; i++)
                leftarray[i] = a[left + i];
        for(j = 0; j < n2; j++)
                rightarray[j] = a[mid + 1 + j];
        i = 0;
        j = 0;
        k = left;
        while(i < n1 && j < n2)
        {
                if(leftarray[i] <= rightarray[j])
                {
```

Merge Sort

```
                    a[k] = leftarray[i];
                    i++;
            }
            else
            {
                    a[k] = rightarray[j];
                    j++;
            }
            k++;
}
while (i < n1)
{
            a[k] = leftarray[i];
            i++;
            k++;
}
```

Merge Sort

```
        while (j < n2)
        {
                a[k] = rightarray[j];
                j++;
                k++;
        }
}
void mergesort(int a[], int left, int right)
{
        if (left < right)
        {
                int mid = (left + right) / 2;
                mergesort(a, left, mid);
                mergesort(a, mid + 1, right);
                merge(a, left, mid, right);
        }
}
```

Quick Sort

- Choose any value (generally the leftmost value) from the array (called the **pivot**) and then **partition** the array into three subarrays such that:
  - the left subarray contains only values less than (or equal to) the pivot;
  - the middle subarray contains only the pivot;
  - the right subarray contains only values greater than (or equal to) the pivot.
- Finally sort the left subarray and the right subarray separately using recursive call.
- This is another application of the divide-and-conquer strategy.

Quick Sort

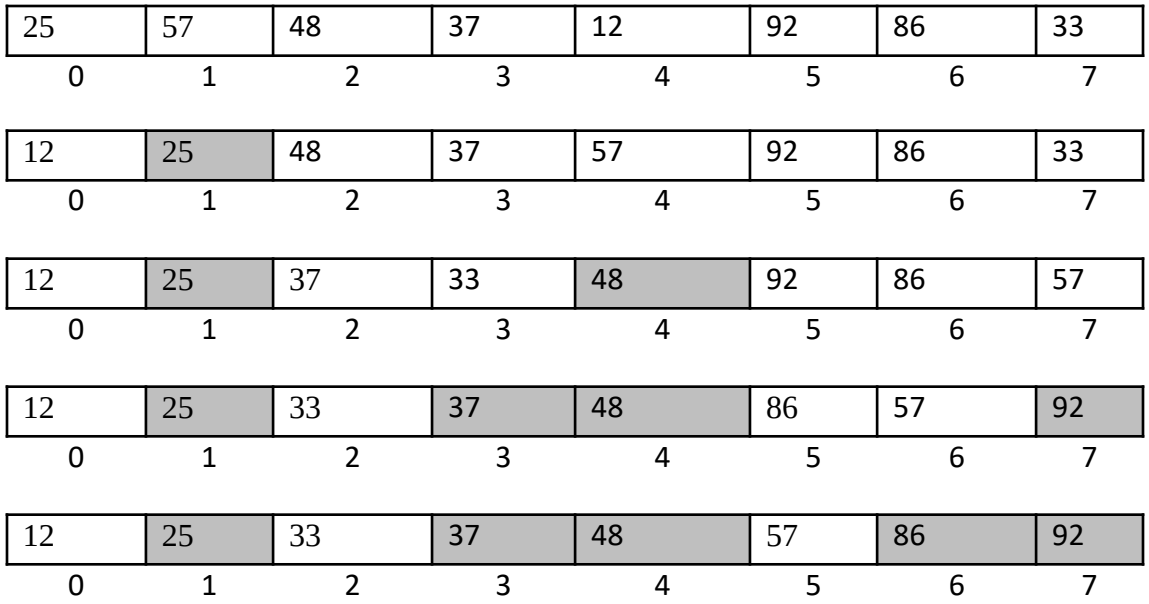
- **Partitioning algorithm**:

  To partition *a*[*left*…*right*] such that *a*[*left*…*p*–1] are all less than or equal to *a*[*p*], and *a*[*p*+1…*right*] are all greater than or equal to *a*[*p*]:

  1. Let *pivot* be the value of *a*[*left*], and set *p* = *left*.
  2. For *r* = *left*+1, …, *right*, repeat:
     2.1. If *a*[*r*] is less than *pivot*:
        2.1.1. Copy *a*[*r*] into *a*[*p*], *a*[*p*+1] into *a*[*r*], and *pivot* into *a*[*p*+1].
        2.1.2. Increment *p*.
  3. Terminate with answer *p*.

- Note that other (and better) partitioning algorithms exist.

Quick Sort



- **Analysis:**
  - Let *n* be the no. of values to be sorted. Let the total no. of comparisons required to sort *n* values be *f*(*n*). The partition algorithm takes about n comparisons to partition the array.

Quick Sort

- In the **best case**, the pivot turns out to be the median value in the array. So the left and right subarrays both have length about $n/2$. Then we need about $f(n/2)$ comparisons to each subarray. Therefore,

  $comps(n) \approx 2\, comps(n/2) + n \quad \text{if } n > 1$
  $comps(n) = 0 \quad \text{if } n \leq 1$
  Hence,

  $comps(n) \approx n \log_2 n$

  Best-case time complexity is $O(n \log n)$.

- In the **worst case**, the pivot turns out to be the smallest value. So the right subarray has length $n–1$ while the left subarray has length 0. Then we need $f(n-1)$ comparisons to sort right subarray and no comparisons to sort left subarray. Therefore,

  $comps(n) \approx comps(n–1) + n \quad \text{if } n > 1$
  $comps(n) = 0 \quad \text{if } n \leq 1$

Quick Sort

Hence,

$comps(n) \approx (n^2 + n - 2)/2$

Worst-case time complexity is $O(n^2)$.

The worst case arises if the array is already sorted and leftmost value is chosen as pivot.

- **Space complexity:** best case is $O(\log_2 n)$ because the algorithm calls itself approximately $\log_2 n$ times and worst case is O(n) because the algorithm calls itself approximately n times.

- To avoid the worst case situation, the partitioning algorithm should try to chose a pivot that is less likely to the least value in the array. For this, we can use the following ideas:
  - Choose the pivot from the middle of the array.
  - Choose the pivot the median of three values taken from the left, middle, and right.

Quick Sort

- **C - Functions:**

```
int partition(int a[], int left, int right)
{
        int pivot = a[left];
        int p = left;
        for(int r = left+1; r <= right; r++)
        {
                if(a[r] < pivot)
                {
                        a[p] = a[r];
                        a[r] = a[p+1];
                        a[p+1] = pivot;
                        p++;
                }
        }
        return p;
}
```

Quick Sort

```
void quicksort(int a[], int left, int right)
{
        if(left < right)
        {
                int p = partition(a, left, right);
                quicksort(a, left, p-1);
                quicksort(a, p+1, right);
        }
}
```