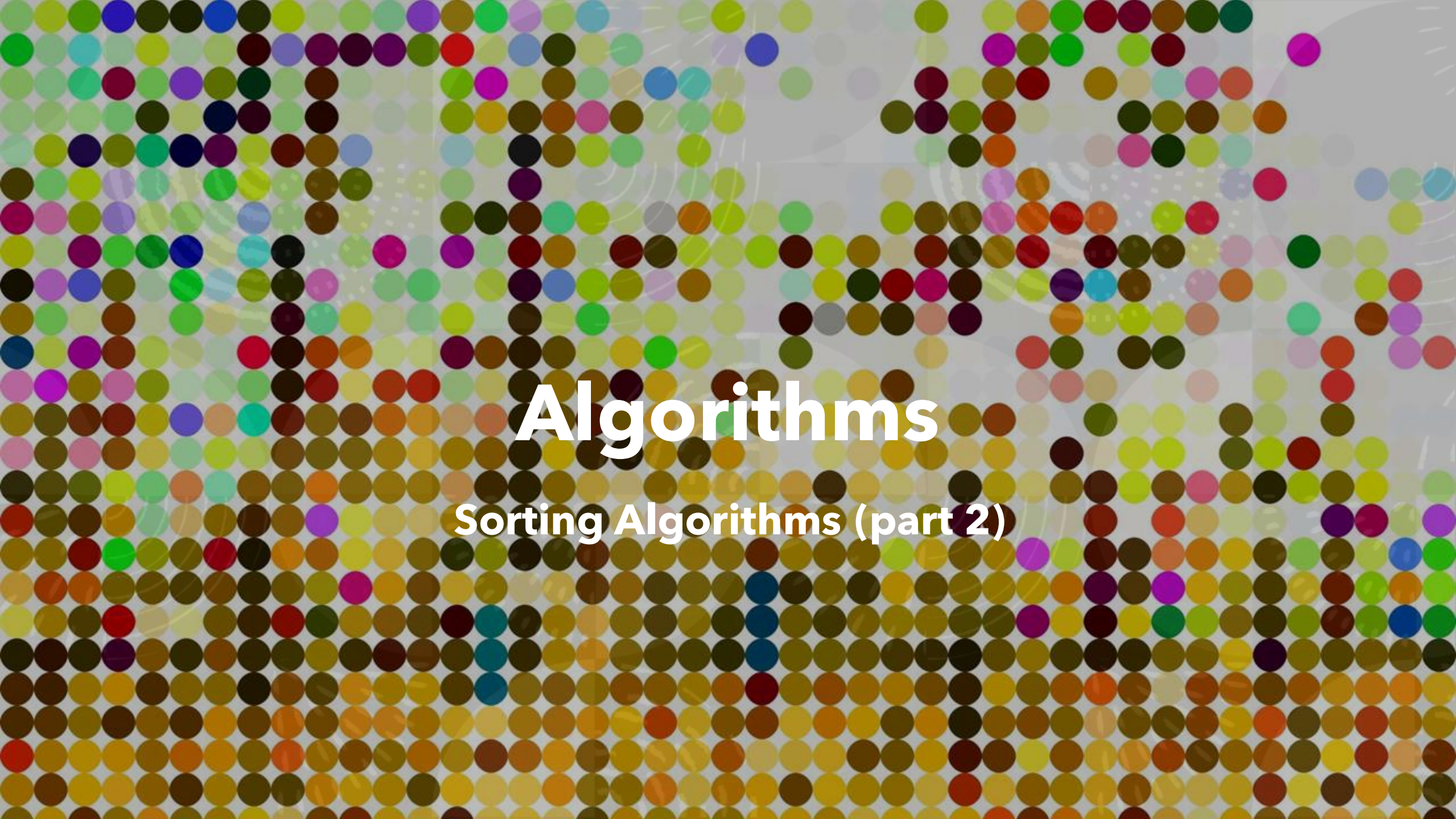

Algorithms
Sorting Algorithms (part 2)

How to analyze a Sorting algorithm?

- A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array.
- There are many factors to consider when choosing a sorting algorithm to use.

- All sorting algorithms share the goal of outputting a sorted list, but the way that each algorithm goes about this task can vary.

- When working with any kind of algorithm, it is important to know how fast it runs and in how much space it operates—in other words, its time complexity and space complexity.

Types of sorting algorithms

- **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 **Ω(nlogn)**.

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.

Types of sorting algorithms

- **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 **Ω(nlogn)**.

They determine for each element **x** how many elements are less than **x**. This information is used to place each element into the correct slot immediately—no need to rearrange lists.

Space complexity

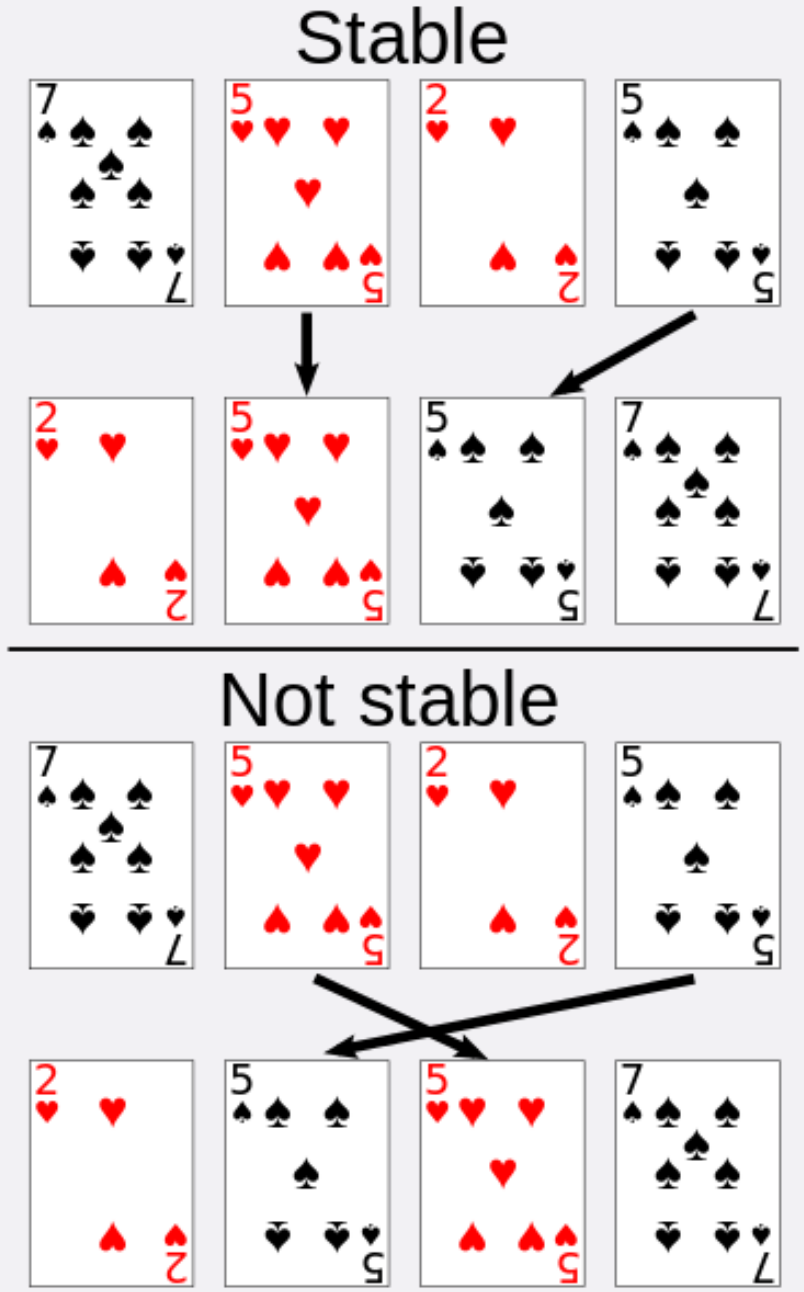
- The **running time** describes how many operations an algorithm must carry out before it completes.

- The **space complexity** describes how much space must be allocated to run a particular algorithm.
  - For example, if an algorithm takes in a list of size **n**, and for some reason makes a new list of size **n** for each element in **n**, the algorithm needs $\mathbf{n^2}$ space.

Stability

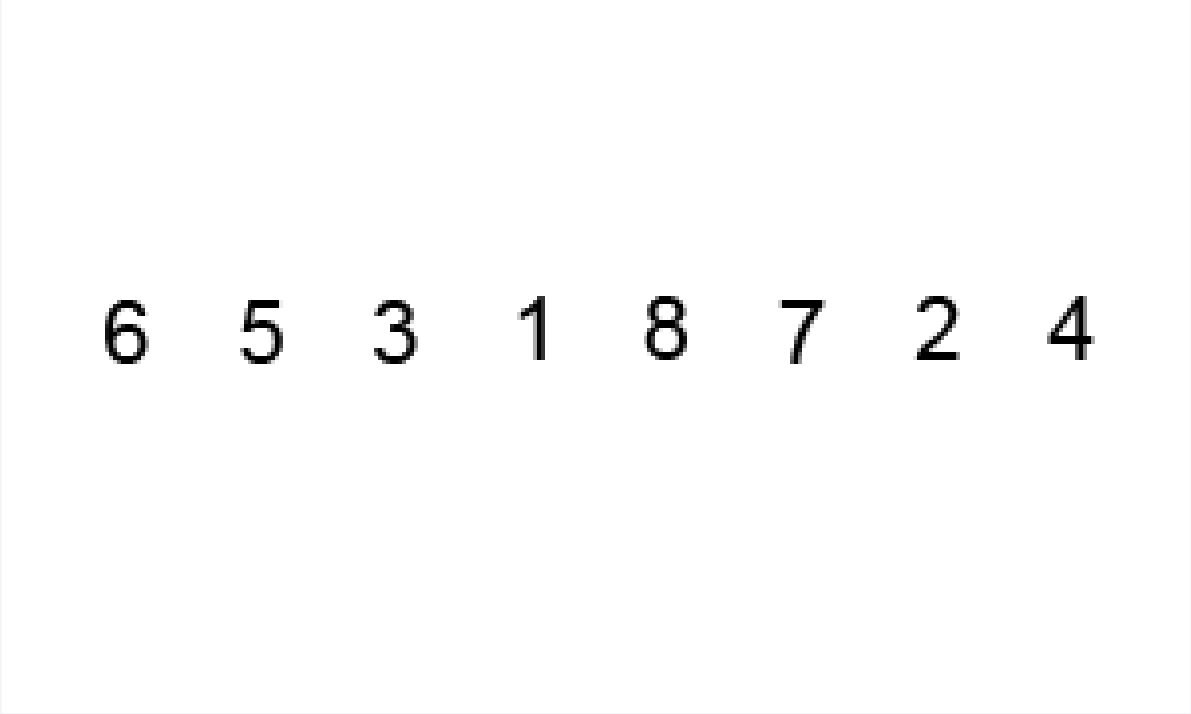
- Additionally, for sorting algorithms, it is sometimes useful to know if a sorting algorithm is **stable**.
- A sorting algorithm is stable **if it preserves the original order of elements with equal key values** (where the key is the value the algorithm sorts by). For example,

- When the cards are sorted by value with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.



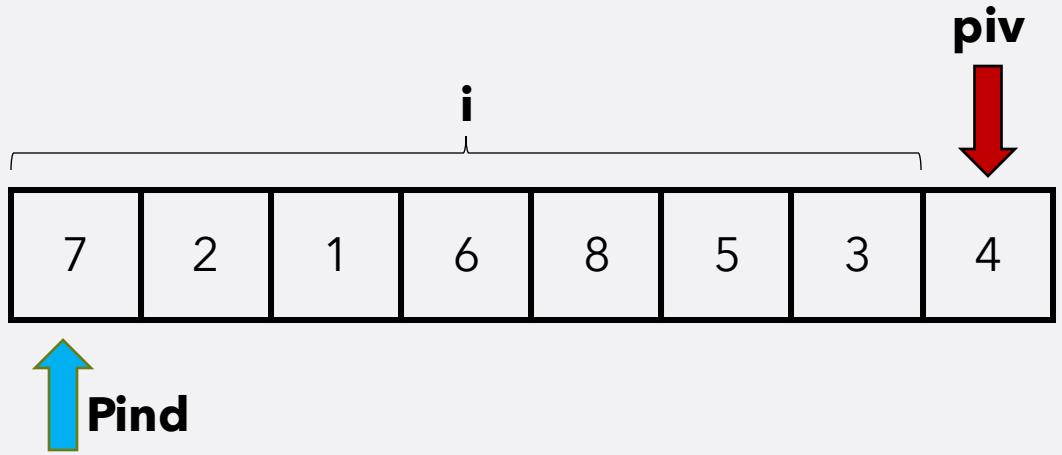
QuickSort

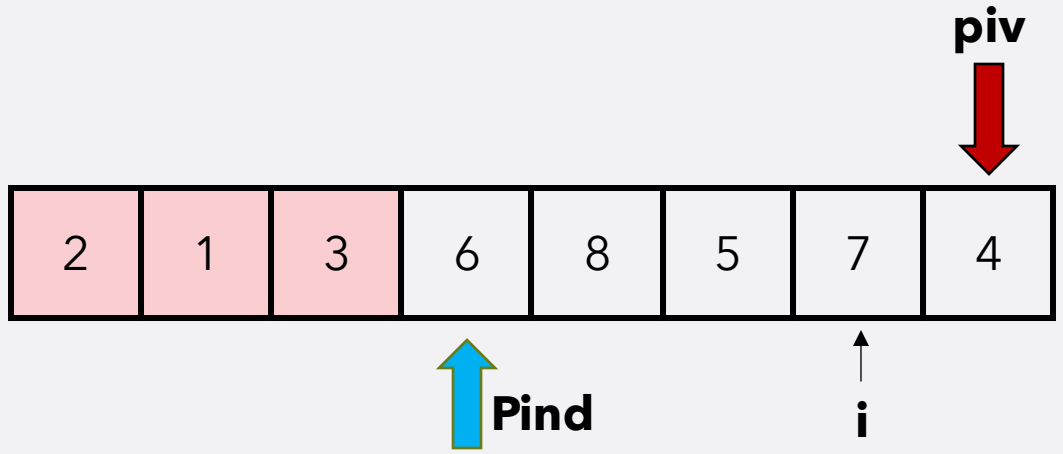
- Quicksort is a **comparison-based algorithm** that uses divide-and-conquer to sort an array.
- The algorithm picks a **pivot** element, **A[q]**, and then rearranges the array into two subarrays **A[p ... q-1]**, such that all elements are less than **A[q]**, and **A[q+1...r]**, such that all elements are greater than or equal to **A[q]**.

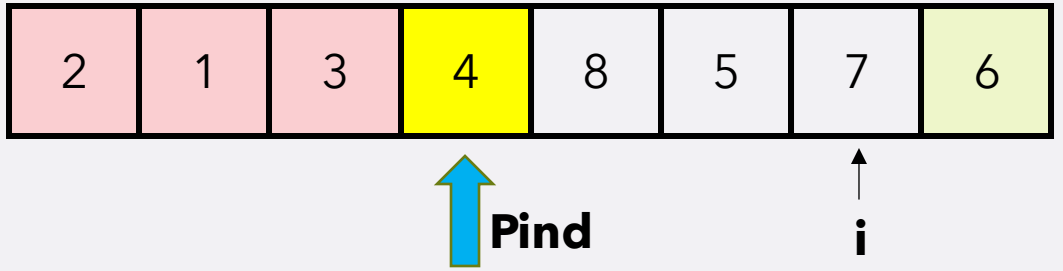



j
7
2
1
8
6
3
5
4

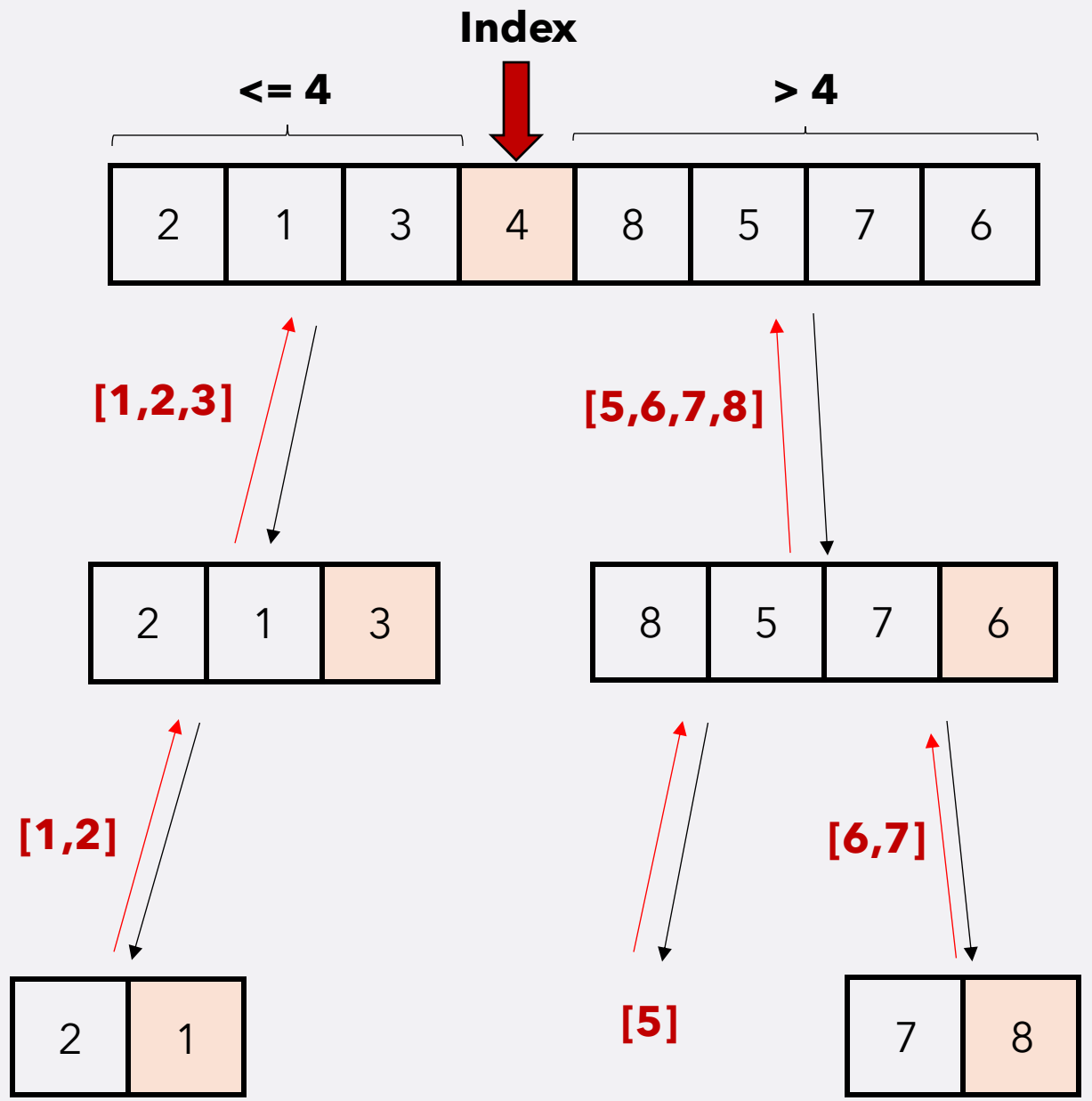
```c
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```





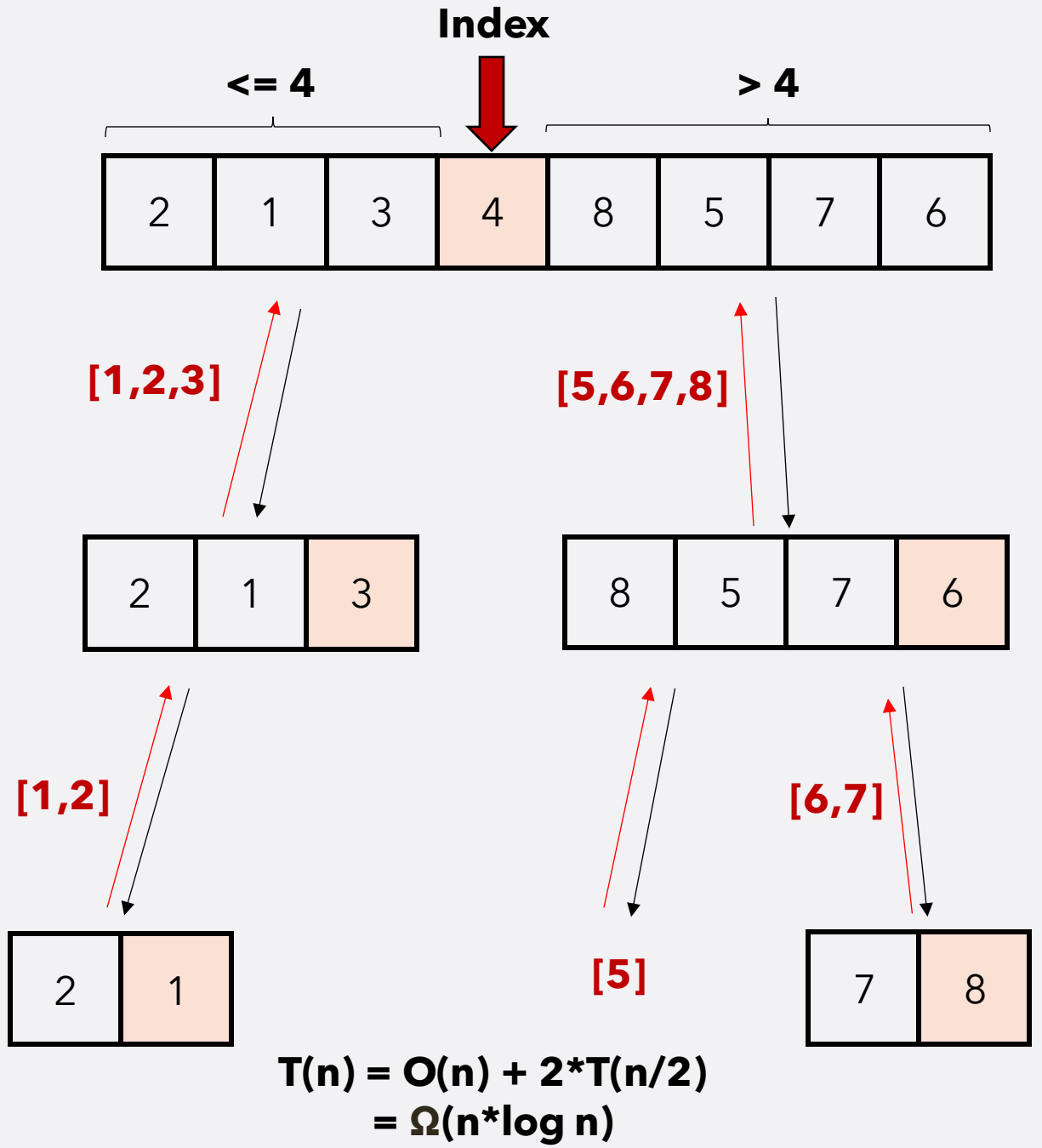


```c
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



Time analysis (best)

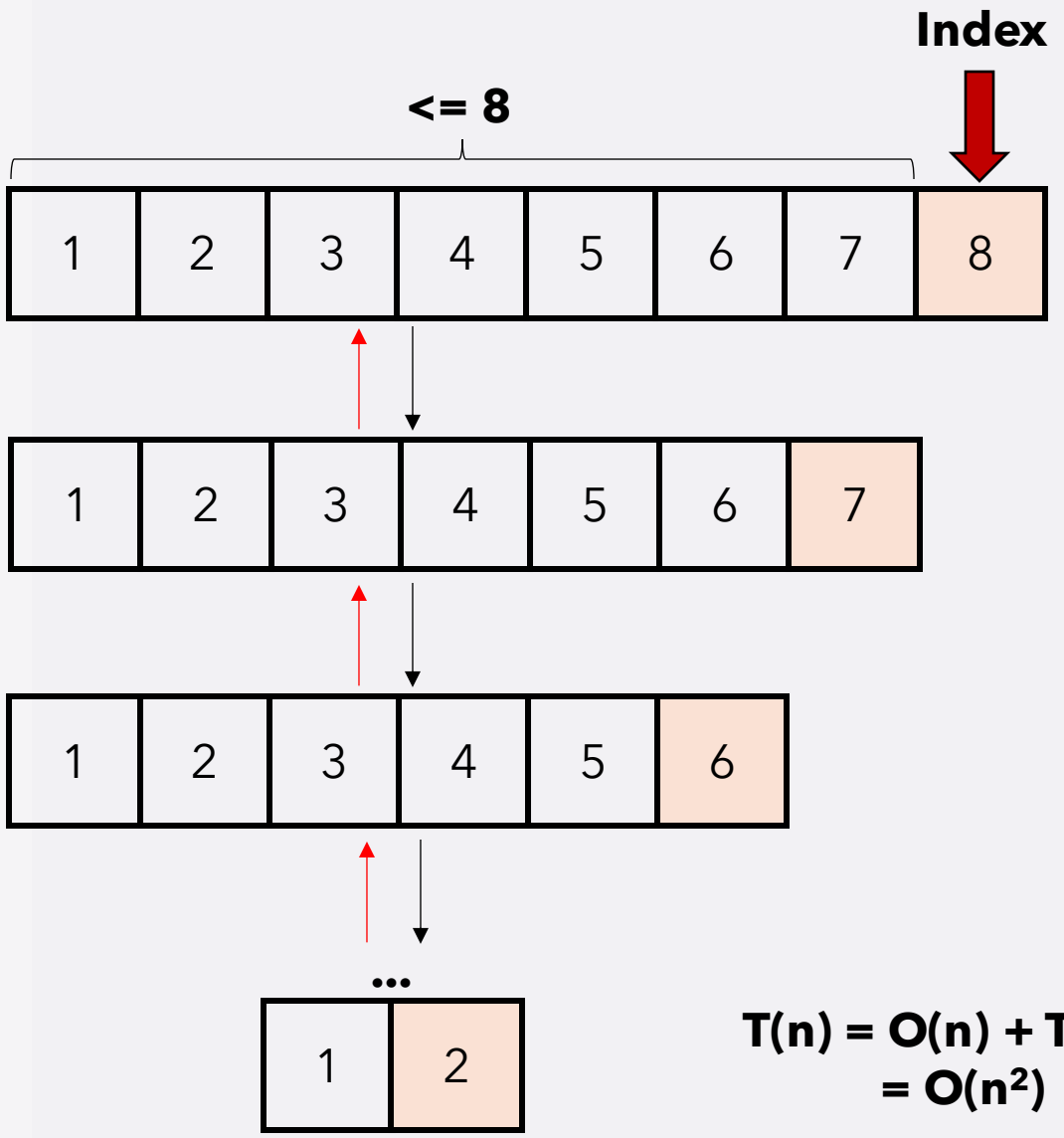
```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



$T(n) = O(n) + 2*T(n/2)$
$= \Omega(n*\log n)$

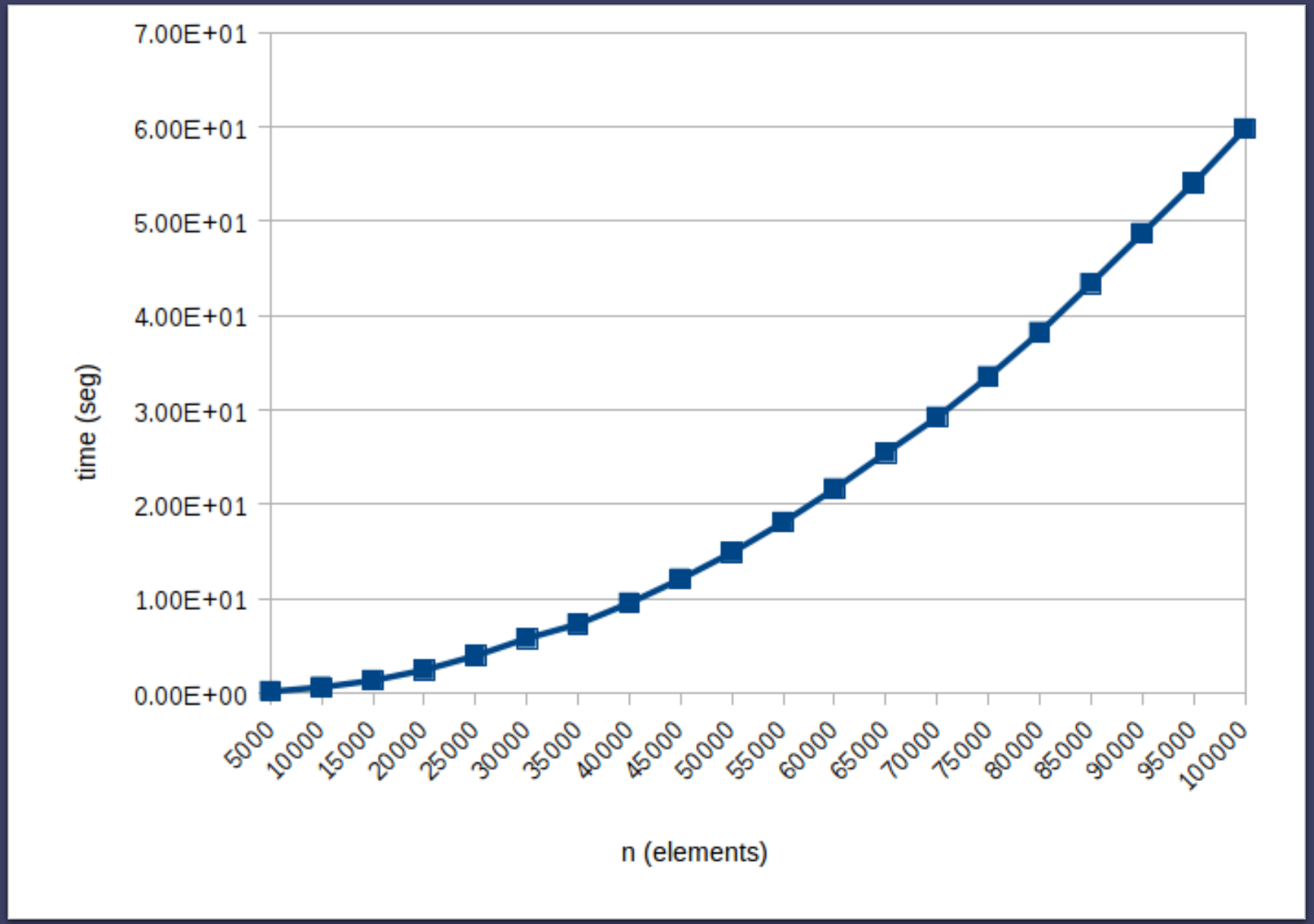
Time analysis (worst)

```
int Quick(int *A, int start, int end)
{
    int piv = A[end];
    int Pind=start;

    for(int i=start;i<end;i++)
    {
        if(A[i] <= piv)
        {
            swap(A[i],A[Pind]);
            Pind++;
        }
    }

    swap(A[Pind],A[end]);

    return Pind;
}

void QuickSort(int *A,int start,int end)
{
    if(start >= end)
        return;
    int Index = Quick(A,start,end);

    QuickSort(A,start,Index-1);
    QuickSort(A,Index+1,end);
}
```



$T(n) = O(n) + T(n-1)$
$= O(n^2)$

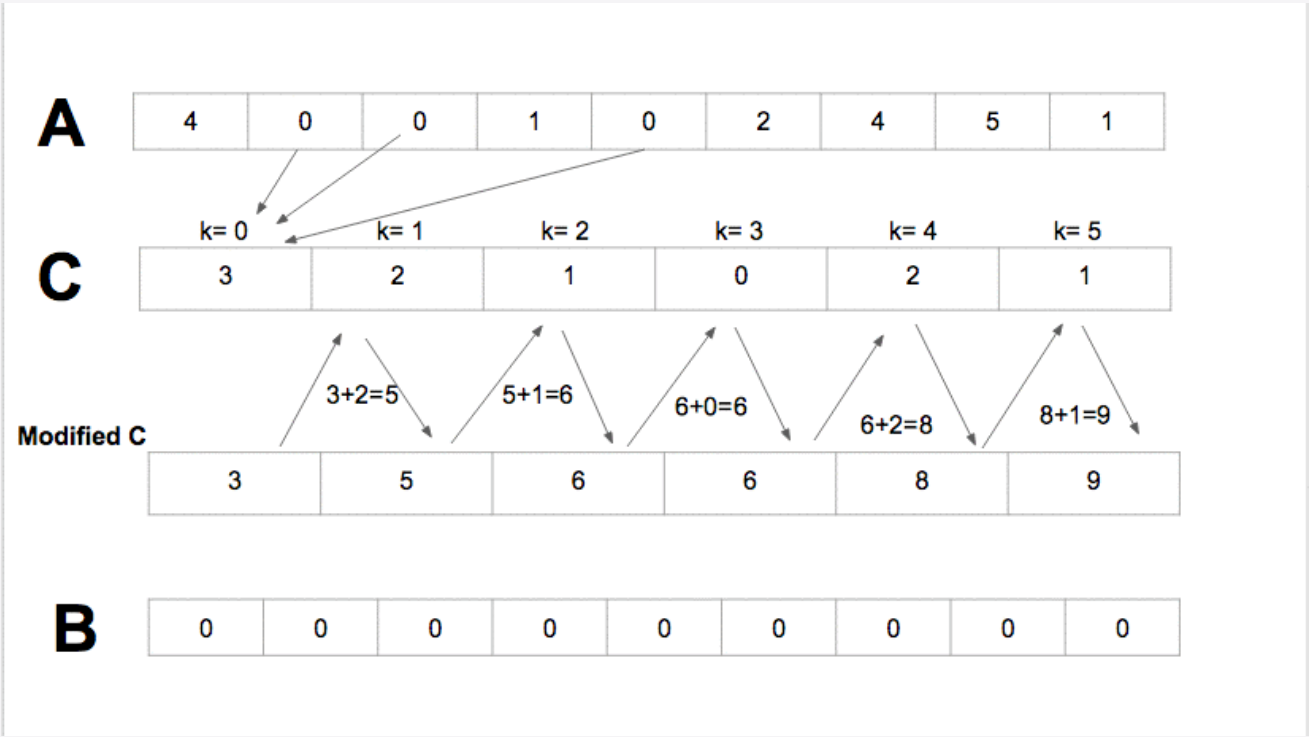
Is Quicksort a stable sort?



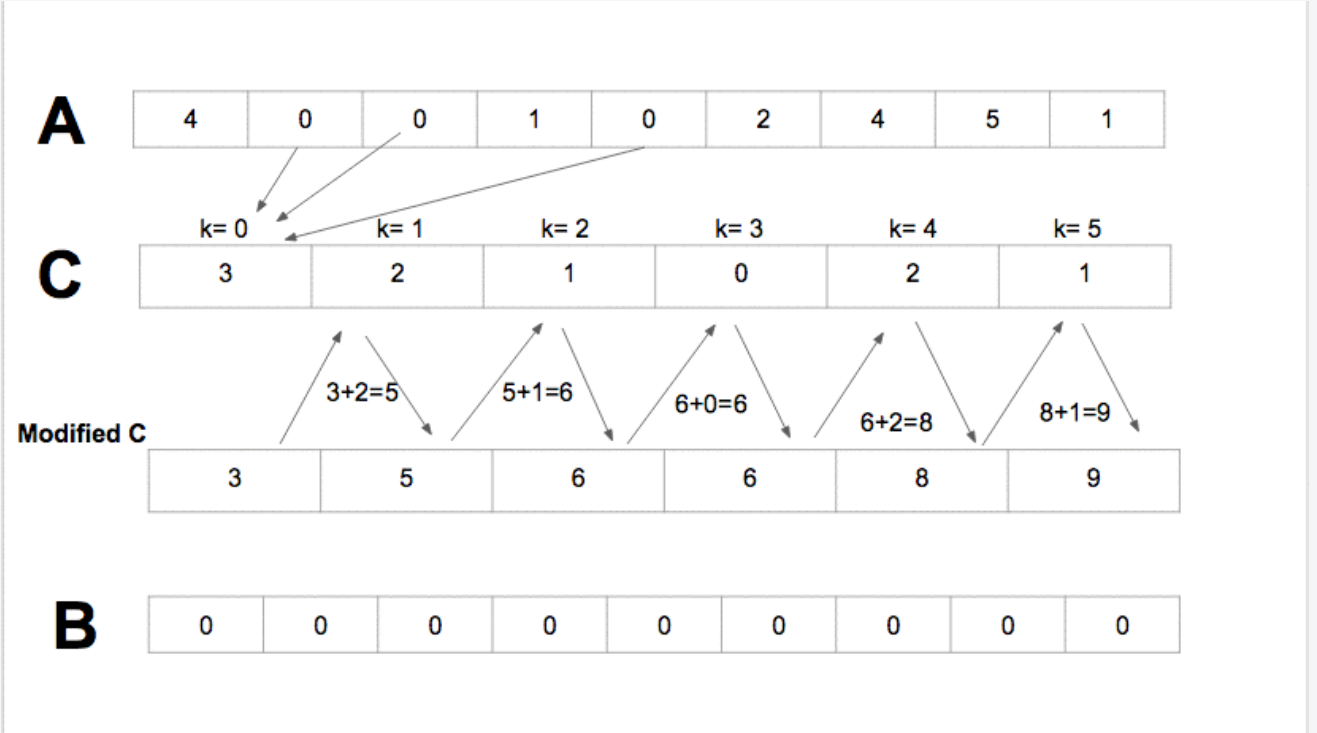
Time complexity (Worst)

Counting Sort

- Counting sort is an **integer sorting** algorithm that assumes that each of the **n** input elements in a list has a **key value** ranging from **0** to **k**, for some **integer k**.
- For each element in the list, counting sort determines the number of elements that are less than it. Counting sort can use this information to place the element directly into the correct slot of the output array.



- **Counting sort uses three lists:** the input list, **A[0,1,...,n]**, the output list, **B[0,1,...,n]**, and a list that serves as temporary memory, **C[0,1,...,k]**.
- Note that **A** and **B** have **n** slots (a slot for each element), while **C** contains **k** slots (a slot for each key value).

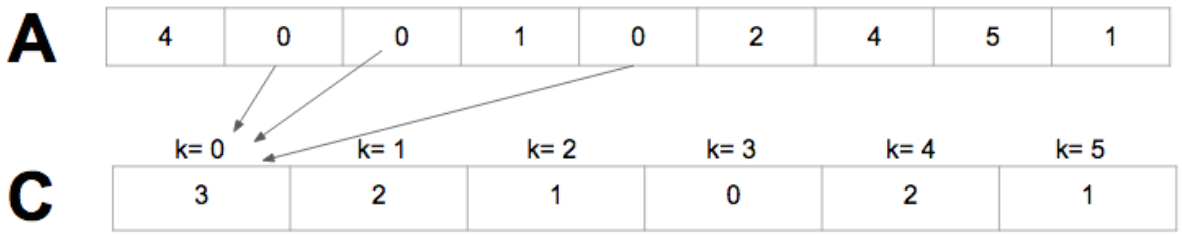


```cpp
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

Caso Base

Inicialización de listas auxiliares C y B

Contar número de elementos en A



- It starts by going through **A**, and for each element **A[i]**, it goes to the index of **C** that has the same value as **A[i]** (so it goes to **C[A[i]]**) and increments the value of **C[A[i]]** by one.
- This means that if **A** has **three 0's** in its list, after counting sort has gone through all **n** elements of **A**, the value at **C[0]** will be **3**.

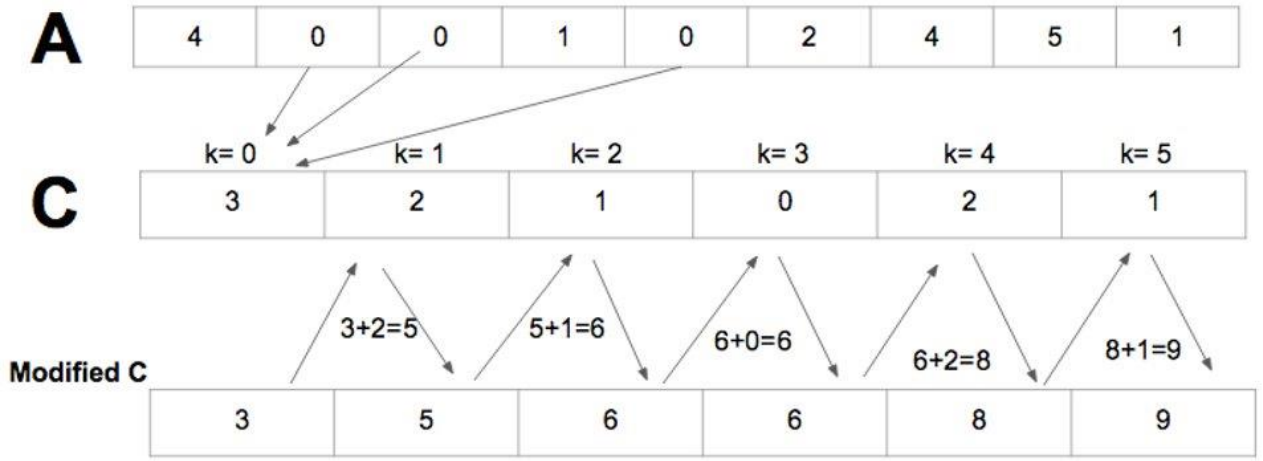
```
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

Caso Base

Inicialización de listas auxiliares C y B

Contar número de elementos en A

modificar C



- Next, modify **C** so that each **C[i]** includes the number of elements less than it. This can be accomplished by going through **C** and replacing each **C[i]** value with **C[i]+C[i−1]**.

- This step allows counting sort to determine at what index in **B** an element should be placed.

```cpp
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

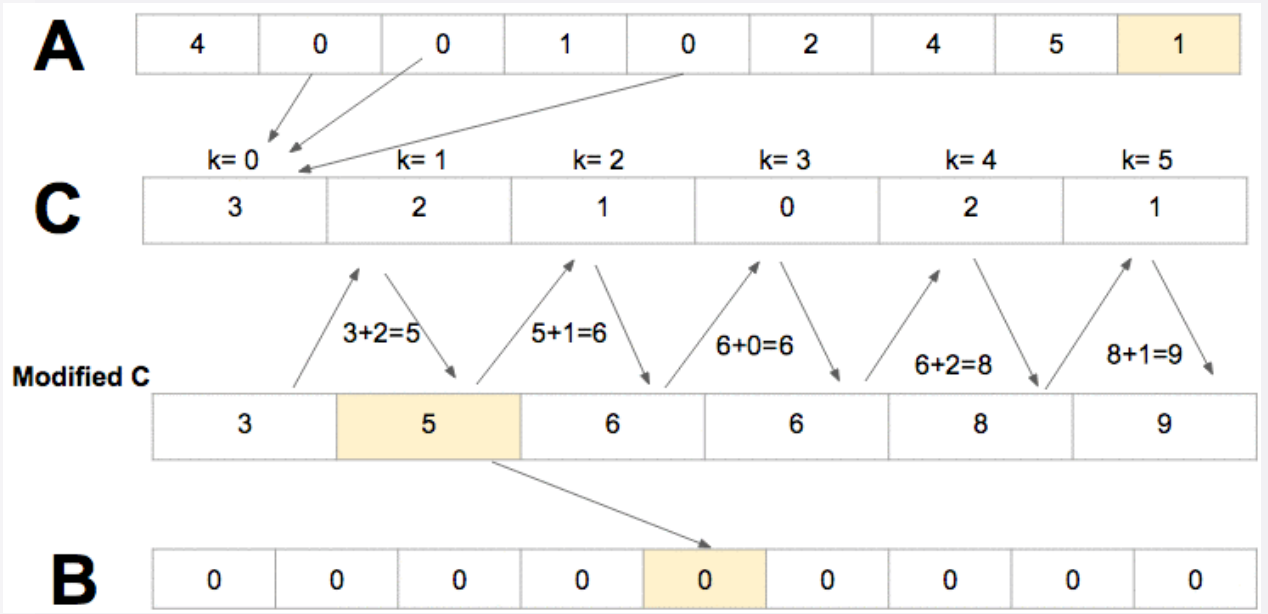
Caso Base

Inicialización de listas auxiliares C y B

Contar número de elementos en A

modificar C

Ordenar A



- Then, starting at the end of **A**, add elements to **B** by checking the value of **A[i]**, going to **C[A[i]]**, writing the value of the element at **A[i]** to **B[C[A[i]]]**.

- Finally, decrement the value of **C[A[i]]** by **1** since that slot in **B** is now occupied.

```cpp
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

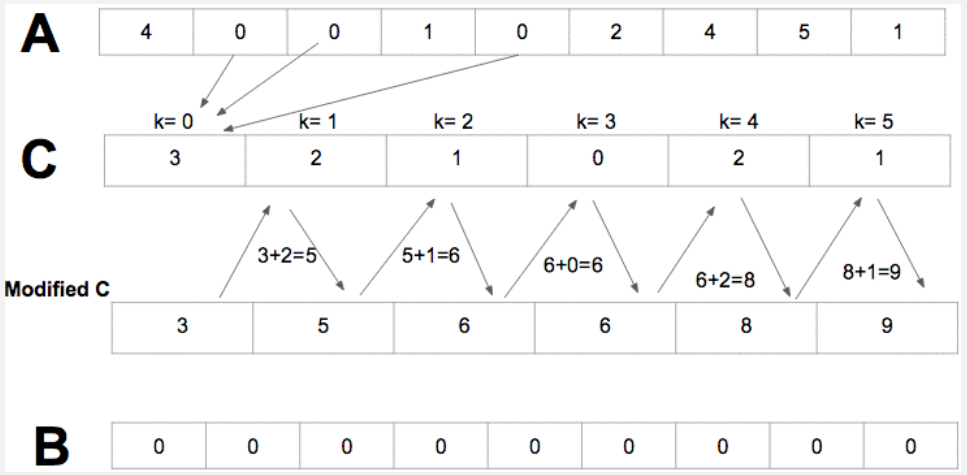
Caso Base

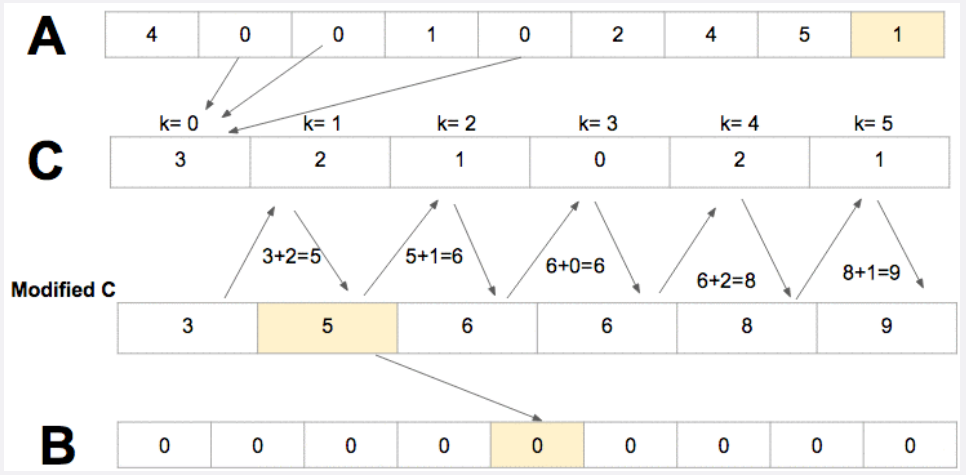
Inicialización de listas auxiliares C y B

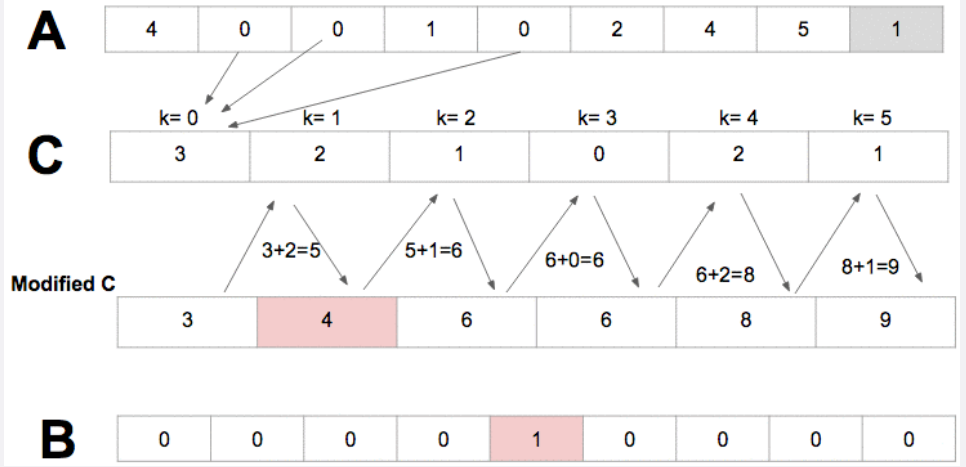
Contar número de elementos en A

modificar C

Ordenar A







```
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

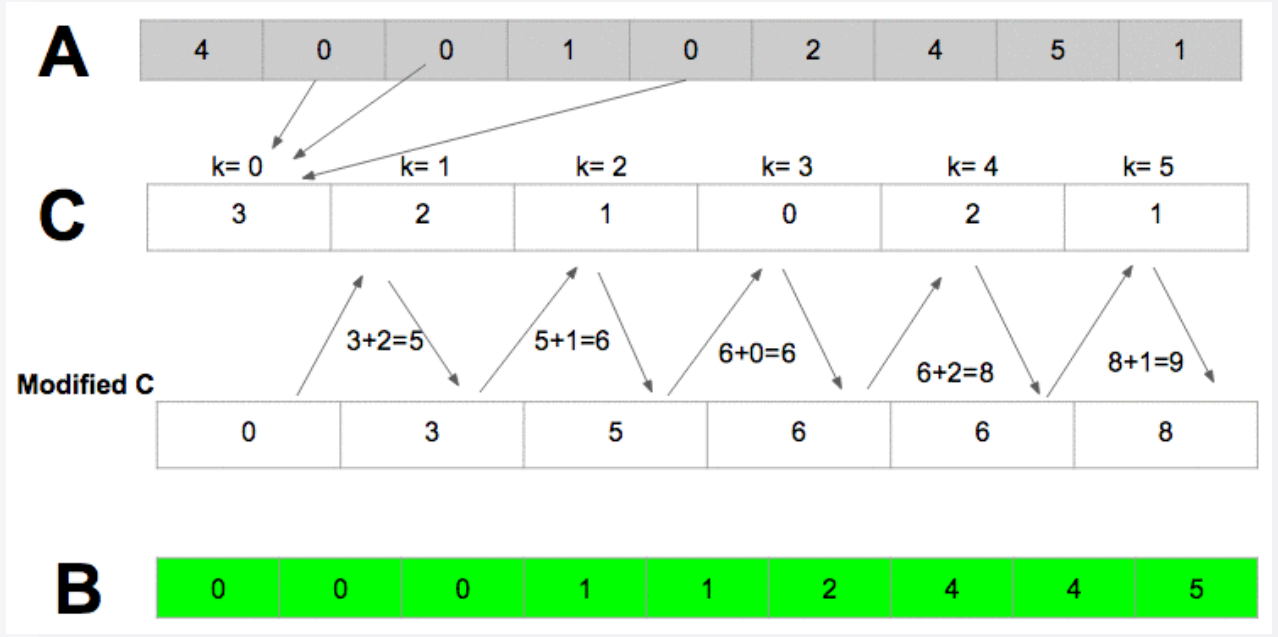
Caso Base

Inicialización de listas auxiliares C y B

Contar número de elementos en A

modificar C

Ordenar A



- Then, starting at the end of **A**, add elements to **B** by checking the value of **A[i]**, going to **C[A[i]]**, writing the value of the element at **A[i]** to **B[C[A[i]]]**.

- Finally, decrement the value of **C[A[i]]** by **1** since that slot in **B** is now occupied.

```
void CountingSort(int *A,int n)
{
    if(n < 2)
        return;

    int max,min;

    max = min = A[0];

    for(int i=1;i<n;i++)
    {
        if(A[i]<min)
            min = A[i];
        if(A[i]>max)
            max = A[i];
    }

    int m = max-min+1;
    vector<int> count(m);
    int B[n];

    for(int i=0;i<n;i++)
    {
        B[i] = A[i];
        count[A[i]-min]++;
    }

    for(int i=1;i<m;i++)
        count[i] += count[i-1];

    for(int i=0;i<n;i++)
    {
        m = B[i]-min;
        A[count[m]-1] = B[i];
        count[m]--;
    }
}
```

Inicialización C y B

Contar número de elementos en A

modificar C

Ordenar A

Time complexity

- $T(n) = O(n) + O(n) + O(m) + O(n)$
  $= \mathbf{O(n + m)}$
  $= \mathbf{\Omega(n + m)}$

Counting sort is efficient if the range of input data, **m**, is not significantly greater than the number of objects to be sorted, **n**.

Space complexity

Counting sort is a **stable** sort with a space complexity of **O(k+n)**.

Radix Sort

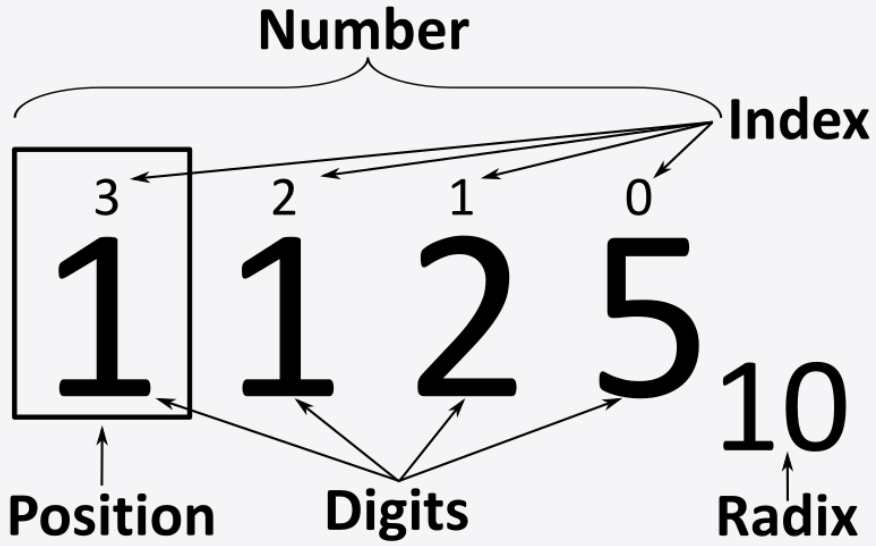
Radix sort is an **integer sorting algorithm** that sorts data with **integer keys** by grouping the keys by **individual digits that share the same significant position and value** (place value).

Radix sort uses **counting sort** as a **subroutine** to sort an array of numbers. Because integers can be used to represent **strings** (by **hashing** the strings to integers), radix sort works on data types other than just integers.

Because radix sort is not comparison based, it is not bounded by **Ω(nlogn)** for running time – in fact, radix sort can perform in linear time.

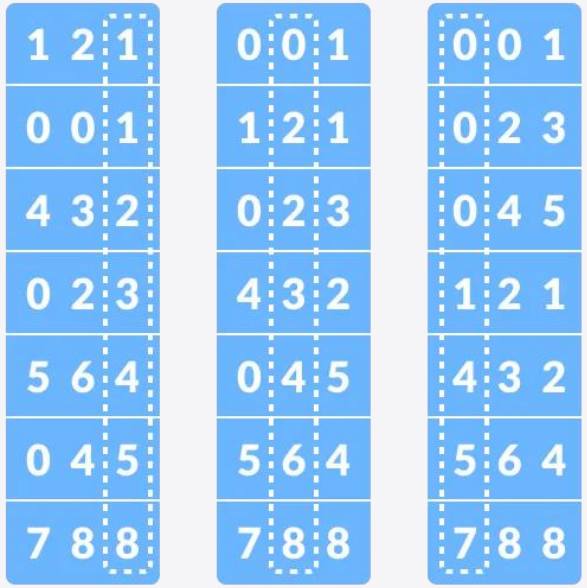
Radix sort takes in a list of **n** integers which are in **base b** (the **radix**) and so each number has at most **d** digits where **d=⌊(logb (k)+1)⌋** and **k** is the largest number in the list.

For example, three digits are needed to represent decimal **104** (in **base 10**).



Radix sort works by sorting each digit from least significant digit to most significant digit. So, in **base 10** (the decimal system), radix sort would sort by the digits in the **1's place**, then the **10's place**, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value.

This means that for a t**hree-digit number** in **base 10**, counting sort will be called to sort the **1's place**, then it will be called to sort the **10's place**, and finally, it will be called to sort the **100's place**, resulting in a completely sorted list.



sorting the integers according to units, tens and hundreds place digits

```
void RadixSort(int *A,int n)
{
    if(n < 2)
        return;

    vector<int> Buckets[10];
    int max = A[0];

    for(int i=0;i<n;i++)
        if(A[i]>max)
            max = A[i];

    int count=0;
    while(max>0)
    {
        count++;
        max/=10;
    }

    int m;
    m = max=0;
    while(count--)
    {
        m = pow(10,max++);

        for(int i=0;i<n;i++)
            Buckets[(A[i]/m)%10].push_back(A[i]);

        int k=0;
        for(int i=0;i<10;i++)
        {
            for(int j: Buckets[i])
                A[k++] = j;
            Buckets[i].clear();
        }
    }

}
```

Caso Base

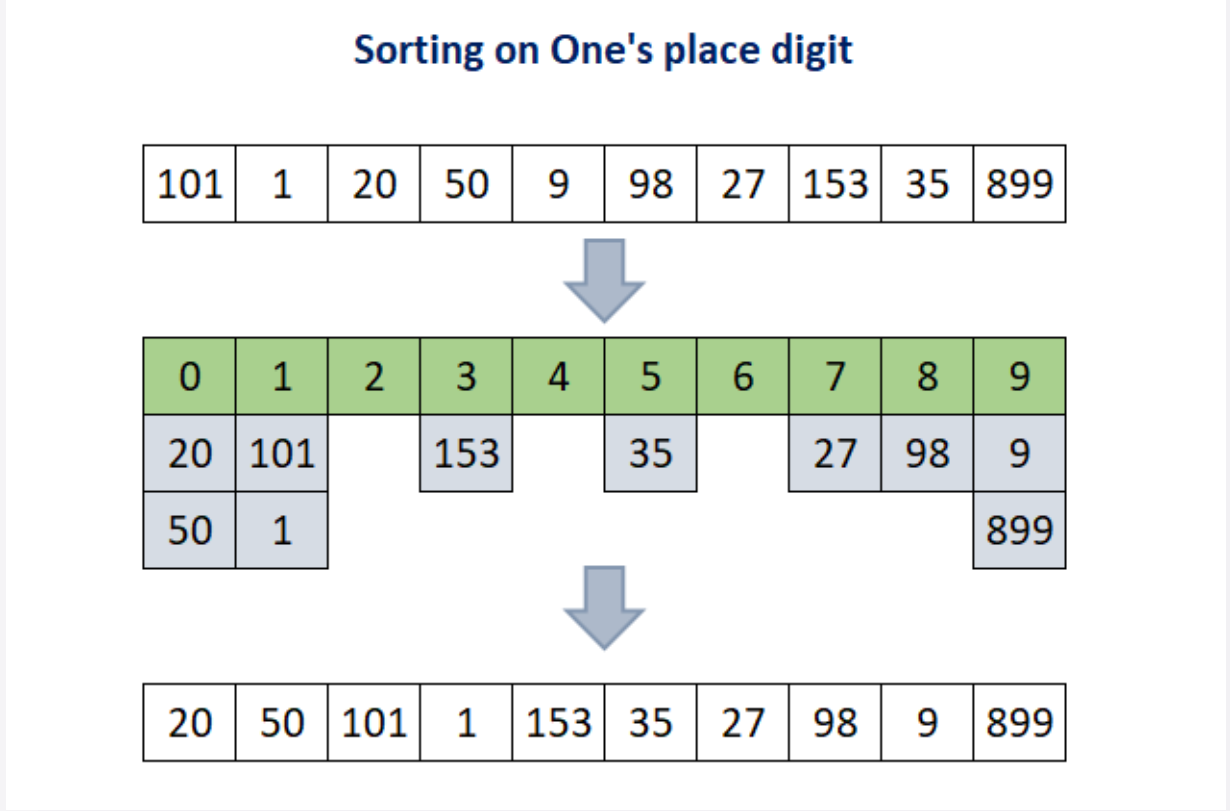
Encontrar máximo

Encontrar número de dígitos máximo

Por cada digito

ordenar por el m-digito

Reacomodo



Step 1: According to the algorithms, the input data is first sorted based on least significant digit. Therefore, array **A[ ]** is sorted based on **one's digit**.

After sorting it on one's digit it will become **[20, 50, 101, 1, 153, 35, 27, 98, 9, 899]**.

```
void RadixSort(int *A,int n)
{
    if(n < 2)
        return;

    vector<int> Buckets[10];
    int max = A[0];

    for(int i=0;i<n;i++)
        if(A[i]>max)
            max = A[i];

    int count=0;
    while(max>0)
    {
        count++;
        max/=10;
    }

    int m;
    m = max=0;
    while(count--)
    {
        m = pow(10,max++);

        for(int i=0;i<n;i++)
            Buckets[(A[i]/m)%10].push_back(A[i]);

        int k=0;
        for(int i=0;i<10;i++)
        {
            for(int j: Buckets[i])
                A[k++] = j;
            Buckets[i].clear();
        }
    }

}
```

Caso Base

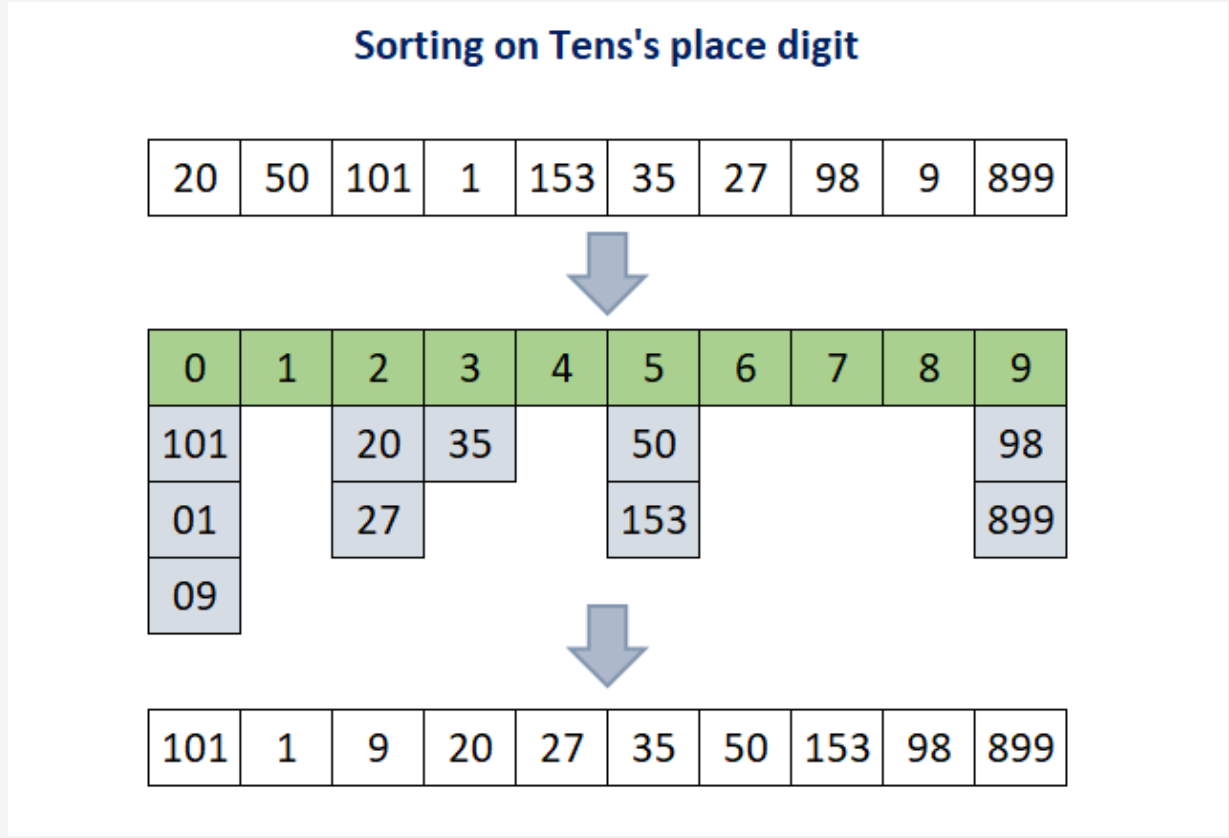
Encontrar máximo

Encontrar número de dígitos máximo

Por cada digito

ordenar por el m-digito

Reacomodo



<u>Step 2:</u> In this step, the array A[ ] array is sorted based on **ten's digit**.

After this step it will become
[101, 1, 9, 20, 27, 35, 50, 153, 98, 899].

```
void RadixSort(int *A,int n)
{
    if(n < 2)
        return;

    vector<int> Buckets[10];
    int max = A[0];

    for(int i=0;i<n;i++)
        if(A[i]>max)
            max = A[i];

    int count=0;
    while(max>0)
    {
        count++;
        max/=10;
    }

    int m;
    m = max=0;
    while(count--)
    {
        m = pow(10,max++);

        for(int i=0;i<n;i++)
            Buckets[(A[i]/m)%10].push_back(A[i]);

        int k=0;
        for(int i=0;i<10;i++)
        {
            for(int j: Buckets[i])
                A[k++] = j;
            Buckets[i].clear();
        }
    }

}
```

Caso Base

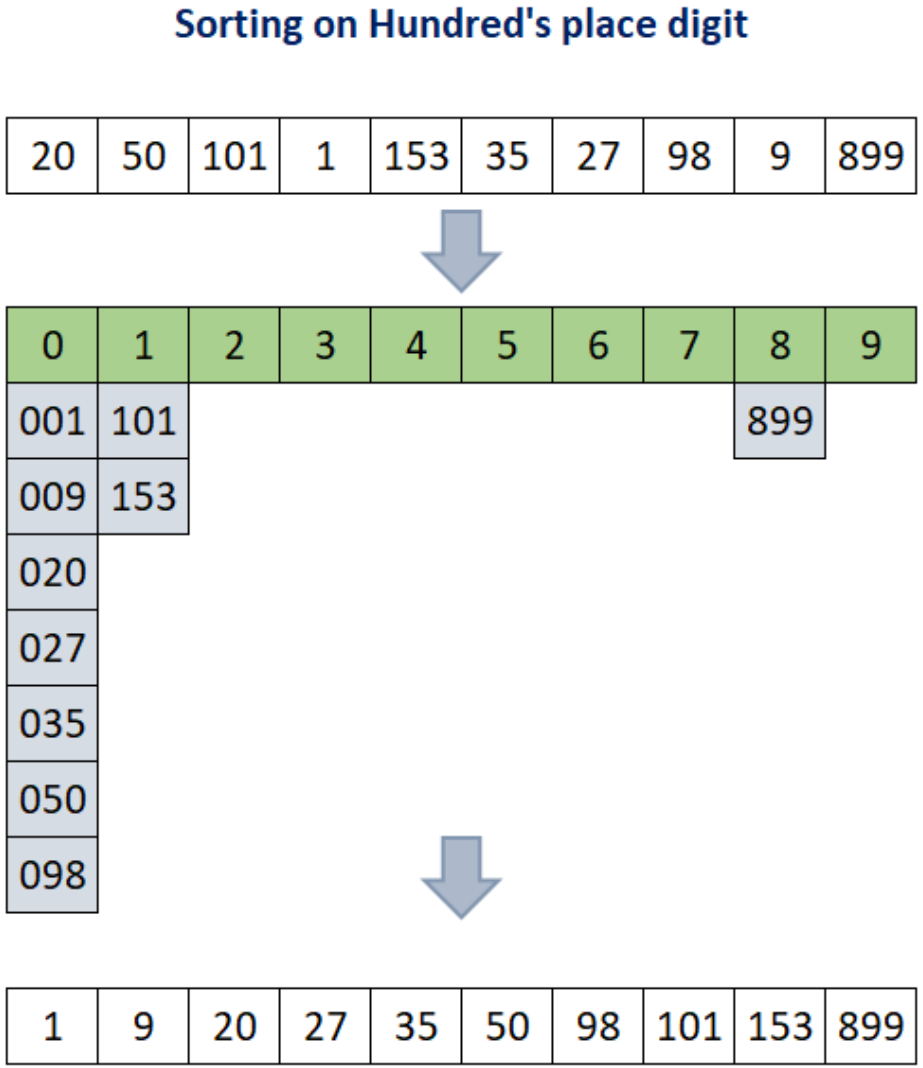
Encontrar máximo

Encontrar número de dígitos máximo

Por cada digito

ordenar por el m-digito

Reacomodo



Step 3: Finally, the array **A[ ]** is sorted based on **hundred's digit** (most significant digit).

The array will be sorted after this step.

```cpp
void RadixSort(int *A,int n)
{
    if(n < 2)
        return;

    vector<int> Buckets[10];
    int max = A[0];

    for(int i=0;i<n;i++)
        if(A[i]>max)
            max = A[i];

    int count=0;
    while(max>0)
    {
        count++;
        max/=10;
    }

    int m;
    m = max=0;
    while(count--)
    {
        m = pow(10,max++);

        for(int i=0;i<n;i++)
            Buckets[(A[i]/m)%10].push_back(A[i]);

        int k=0;
        for(int i=0;i<10;i++)
        {
            for(int j: Buckets[i])
                A[k++] = j;
            Buckets[i].clear();
        }
    }

}
```

Caso Base

Encontrar máximo

Encontrar número de dígitos máximo

Por cada digito

ordenar por el m-digito

Reacomodo

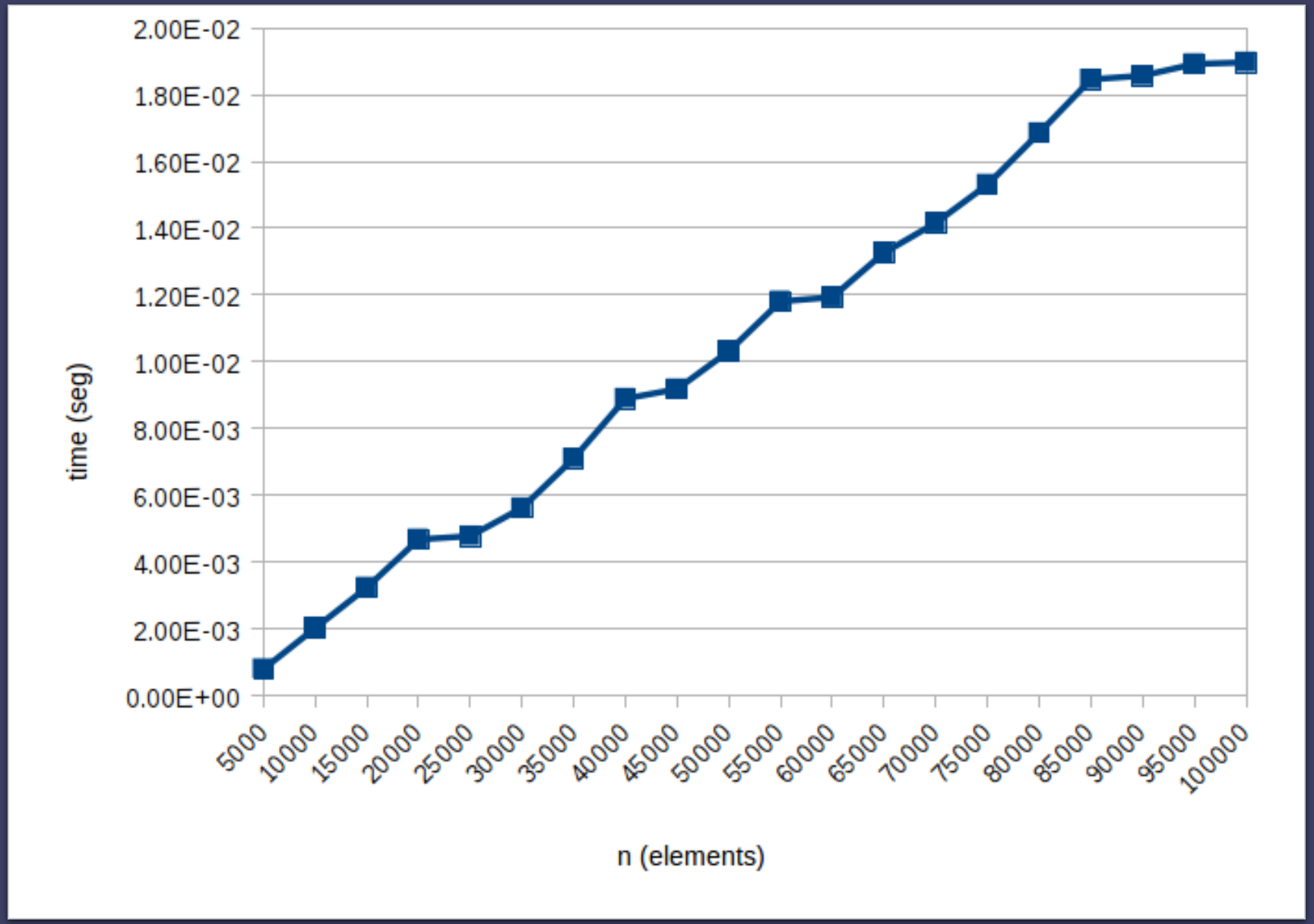
Time complexity

- $T(n) = O(n) + O(k) + O(k*n) + O(k*10)$
  $= \mathbf{O(k*n)}$
  $= \mathbf{\Omega(k*n)}$

It is important that radix sort can work with any base since the running time of the algorithm, **O(k(n+base))**, depends on the base it uses. The algorithm runs in linear time when **base** and **n** are of the same size magnitude, so knowing **n**, **base** can be manipulated to optimize the running time of the algorithm.

Space complexity

Radix sort is a **stable** sort with a space complexity of **O(k+n)**.



Time complexity

Summary

To choose a sorting algorithm for a particular problem, consider the running time, space complexity, and the expected format of the input list.

Algorithm	Best-case	Worst-case	Average-case	Space Complexity	Stable?
Quicksort	O(n*logn)	$O(n^2)$	O(nlogn)	**logn** best, **n** avg	Usually not*
Radixsort	O(k*n)	O(k*n)	O(k*n)	O(k+n)	Yes
Counting Sort	O(k+n)	O(k+n)	O(k+n)	O(k+n)	Yes

*Most **quicksort** implementations are not stable, though stable implementations do exist.

When choosing a sorting algorithm to use, weigh these factors. For example, **quicksort** is a very fast algorithm but can be pretty tricky to implement; **bubble sort** is a slow algorithm but is very easy to implement. To sort small sets of data, **bubble sort** may be a better option since it can be implemented quickly, but for larger datasets, the speedup from **quicksort** might be worth the trouble implementing the algorithm.

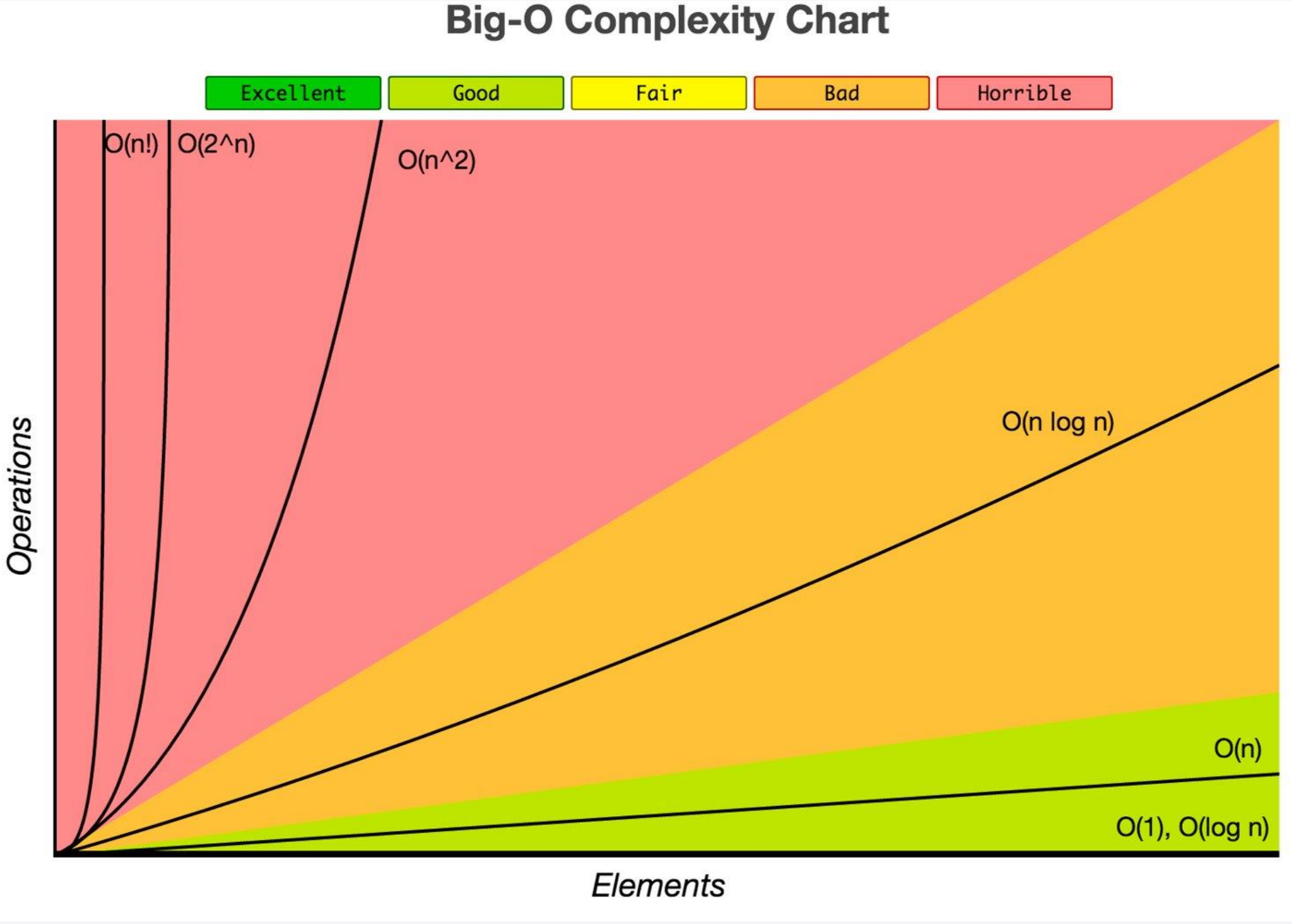
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Time complexity classifications

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

The slower growing functions are listed first.

Clase	Nombre
1	constante
$\log n$	logarítmico
n	lineal
$n \log n$	n-log-n o linearítmico
n^2	cuadrático
n^3	cúbico
2^n	exponencial
$n!$	factorial


Big-O Complexity Chart
Excellent
Good
Fair
Bad
Horrible
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements