Lecture 3

Sorting Algorithms

Outline

- Insertion sort
- Quick sort

Sorting Algorithms

Insertion Sort

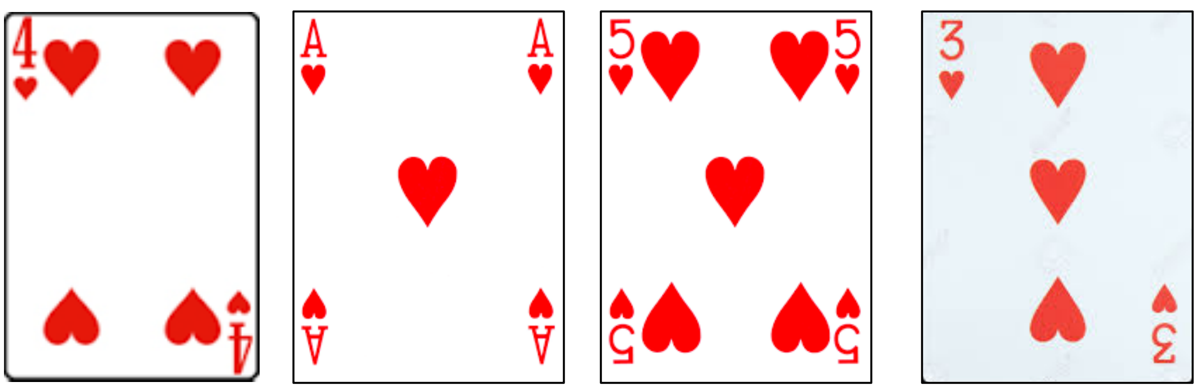
- Simple, but it does not have very good performance
- It still does not have good performance but it might be useful in practice, when the array is already partially sorted.
- It has quadratic performance in the worse case scenario.

Sorting Algorithms

Insertion Sort

- Simple, but it does not have very good performance
- It still does not have good performance but it might be useful in practice, when the array is already partially sorted.

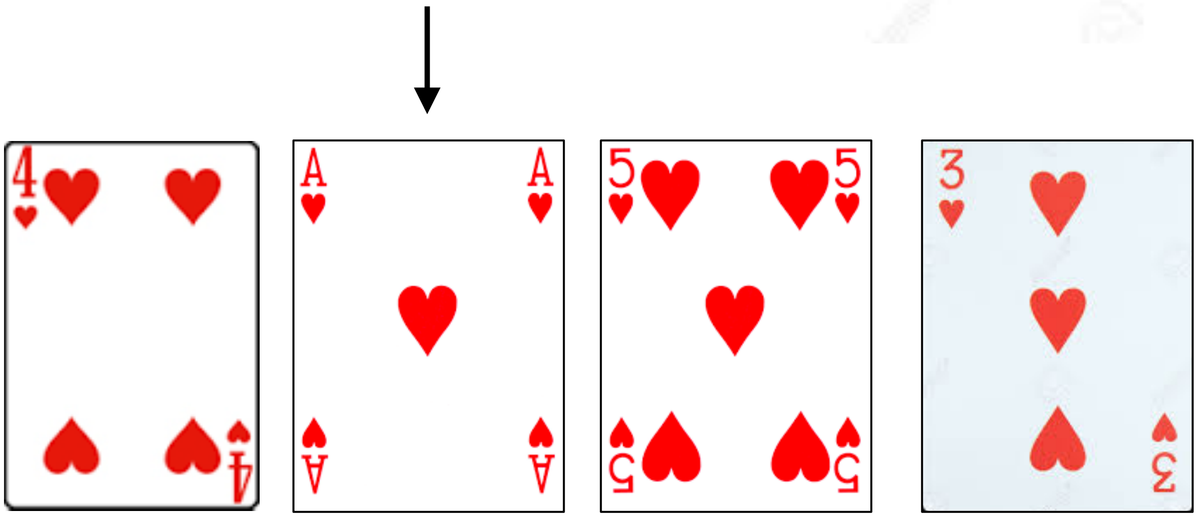
Insertion Sort – How it Works



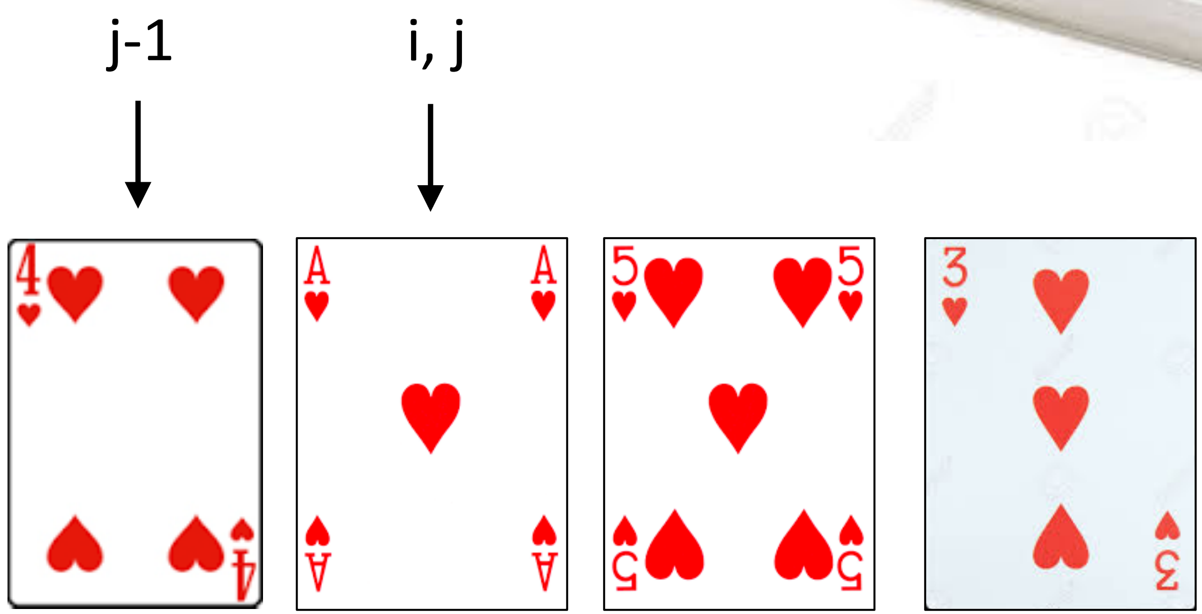
Insertion Sort – How it Works

- Index i points to the second element of the array
- j is set = i

i, j

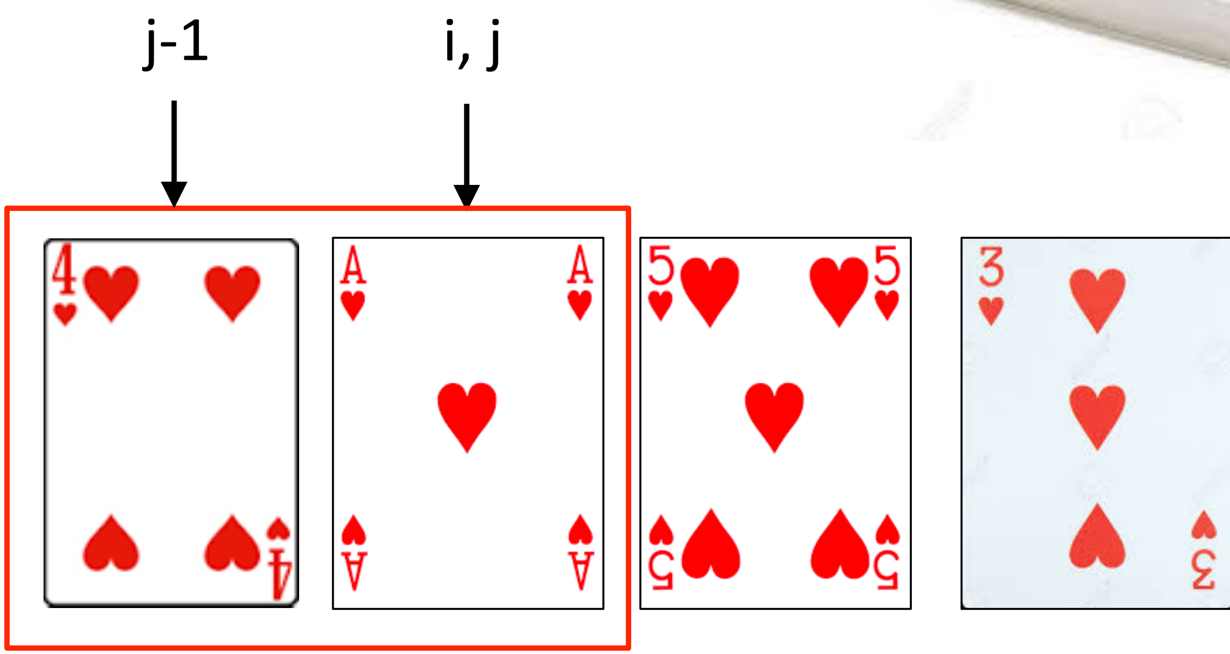


Insertion Sort – How it Works

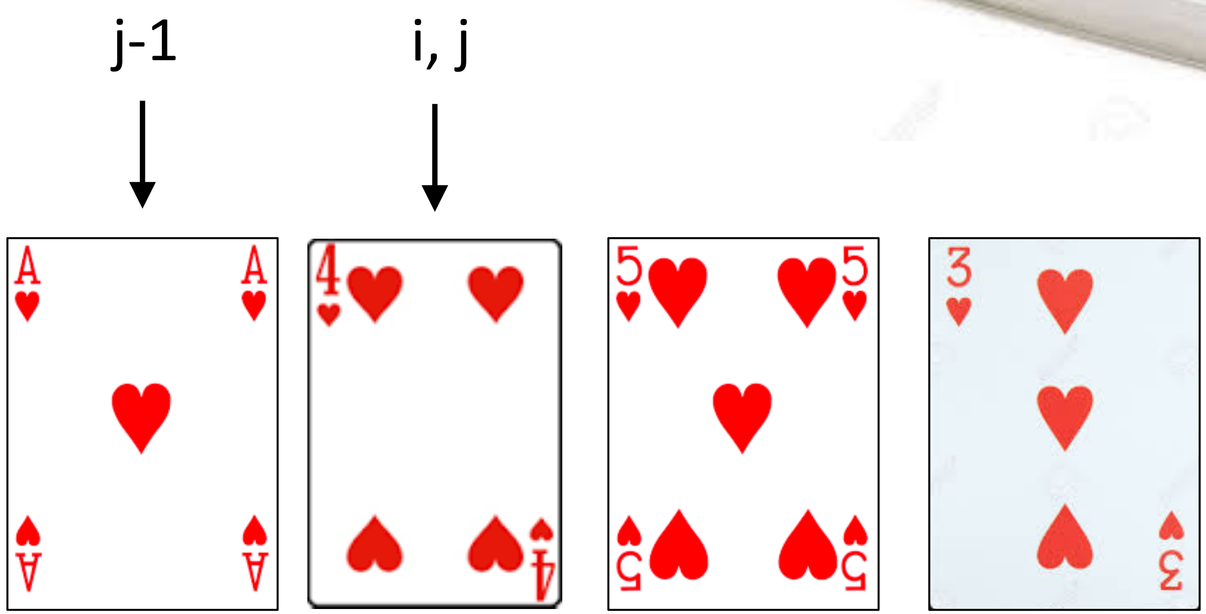


Insertion Sort – How it Works

If the element at a index j is smaller than the element at index j-1, the two elements are swapped

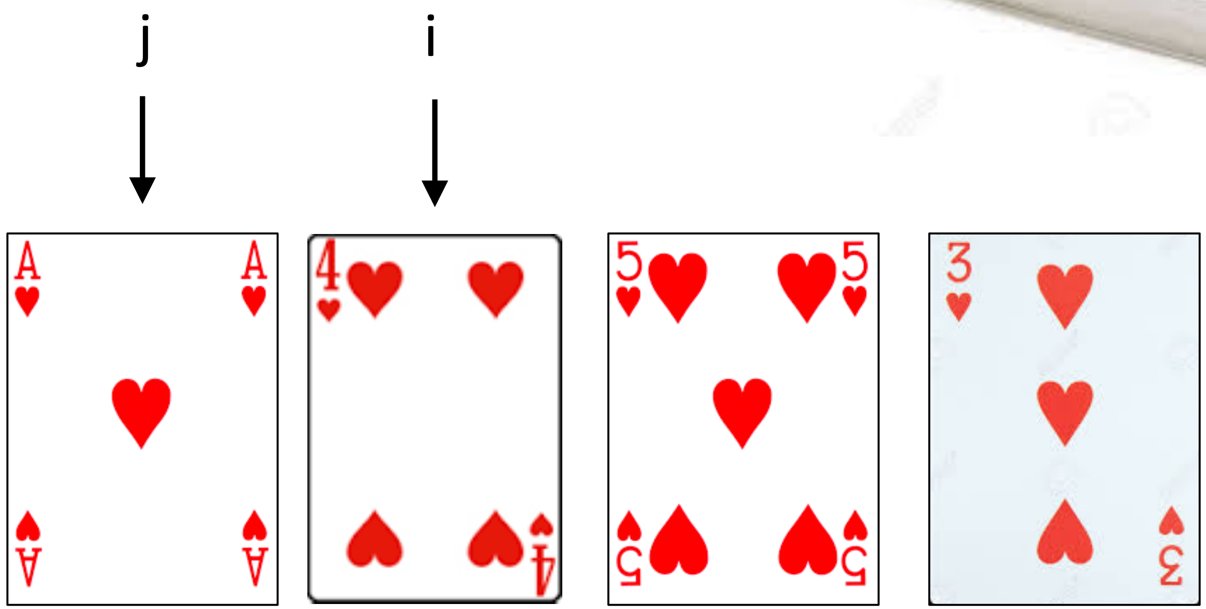


Insertion Sort – How it Works



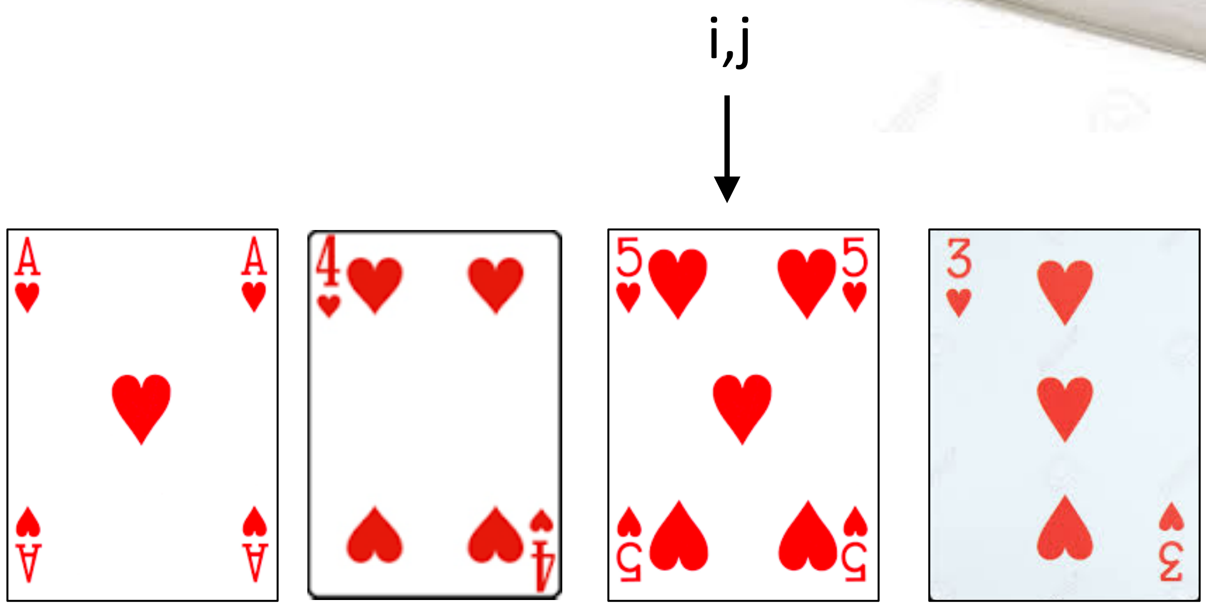
Insertion Sort – How it Works

Let’s decrement j until it is > 0



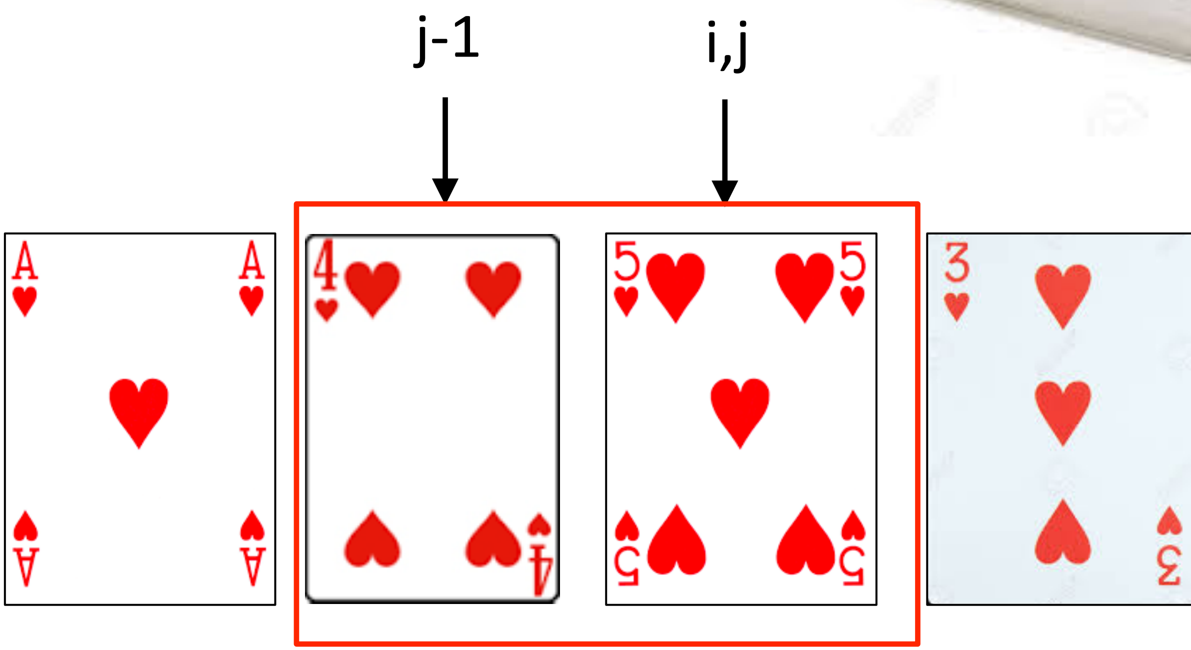
Insertion Sort – How it Works

- Let’s increment i
- Lets set j = i



Insertion Sort – How it Works

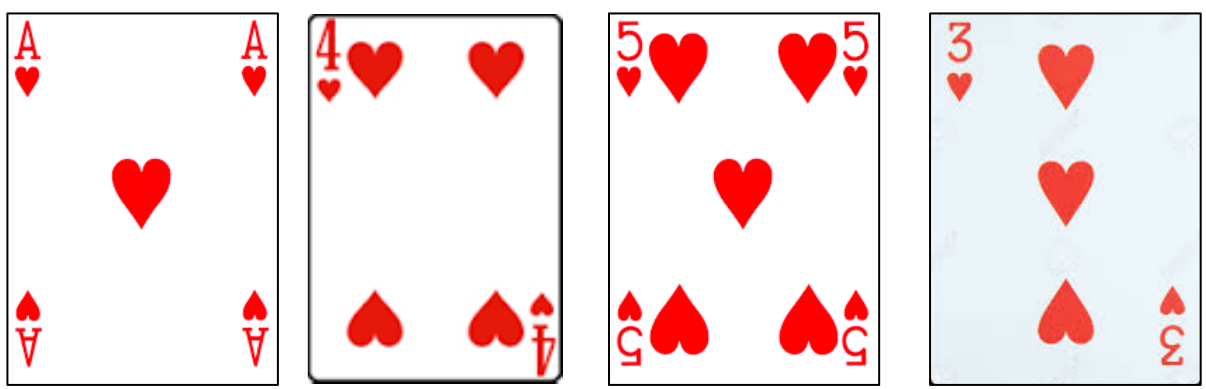
- If the element at a index j is smaller than the element at index j-1, the two elements are swapped
- If that is not the case, i is incremented…



Insertion Sort – How it Works

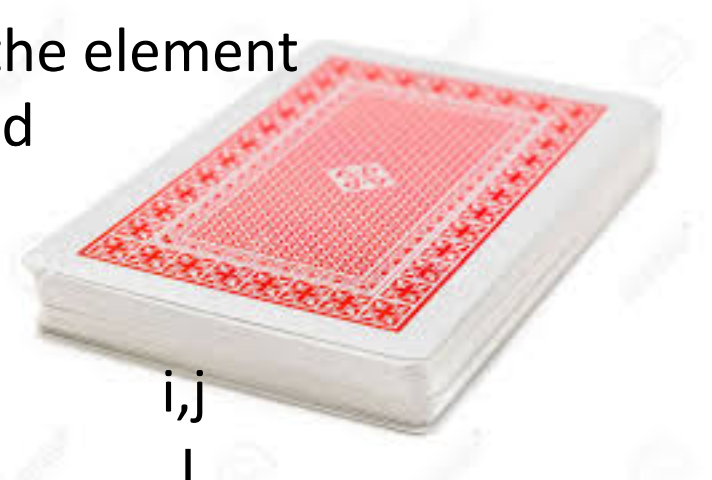
- Let’s increment i
- Lets set j = i

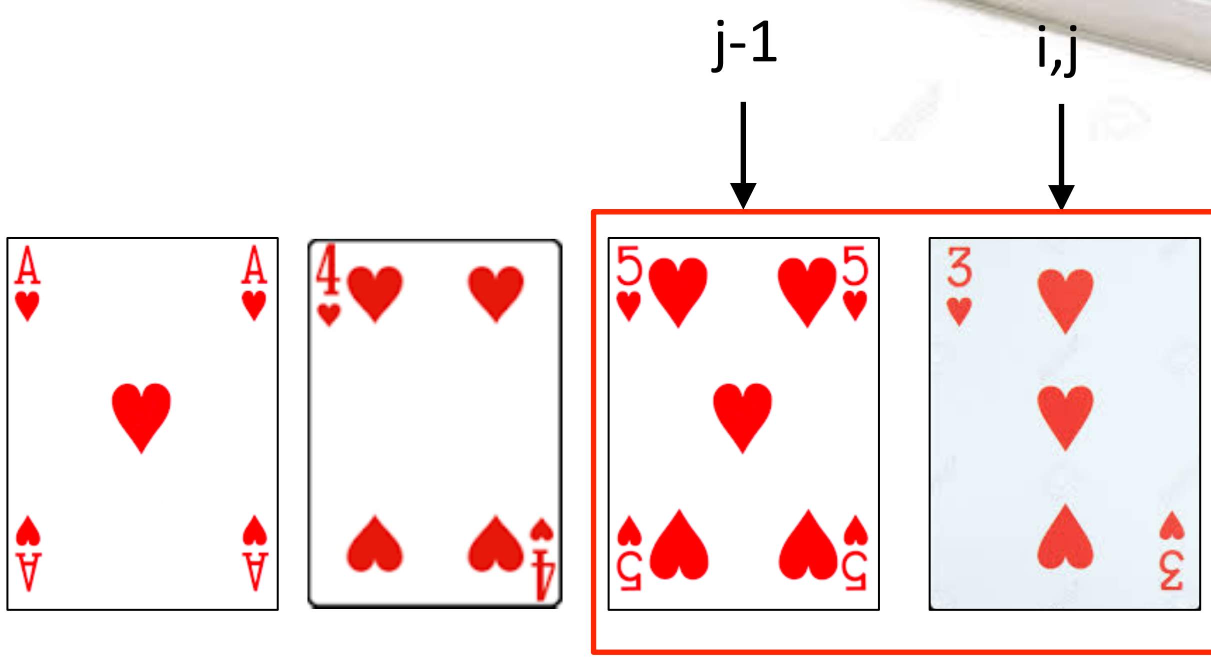
i,j



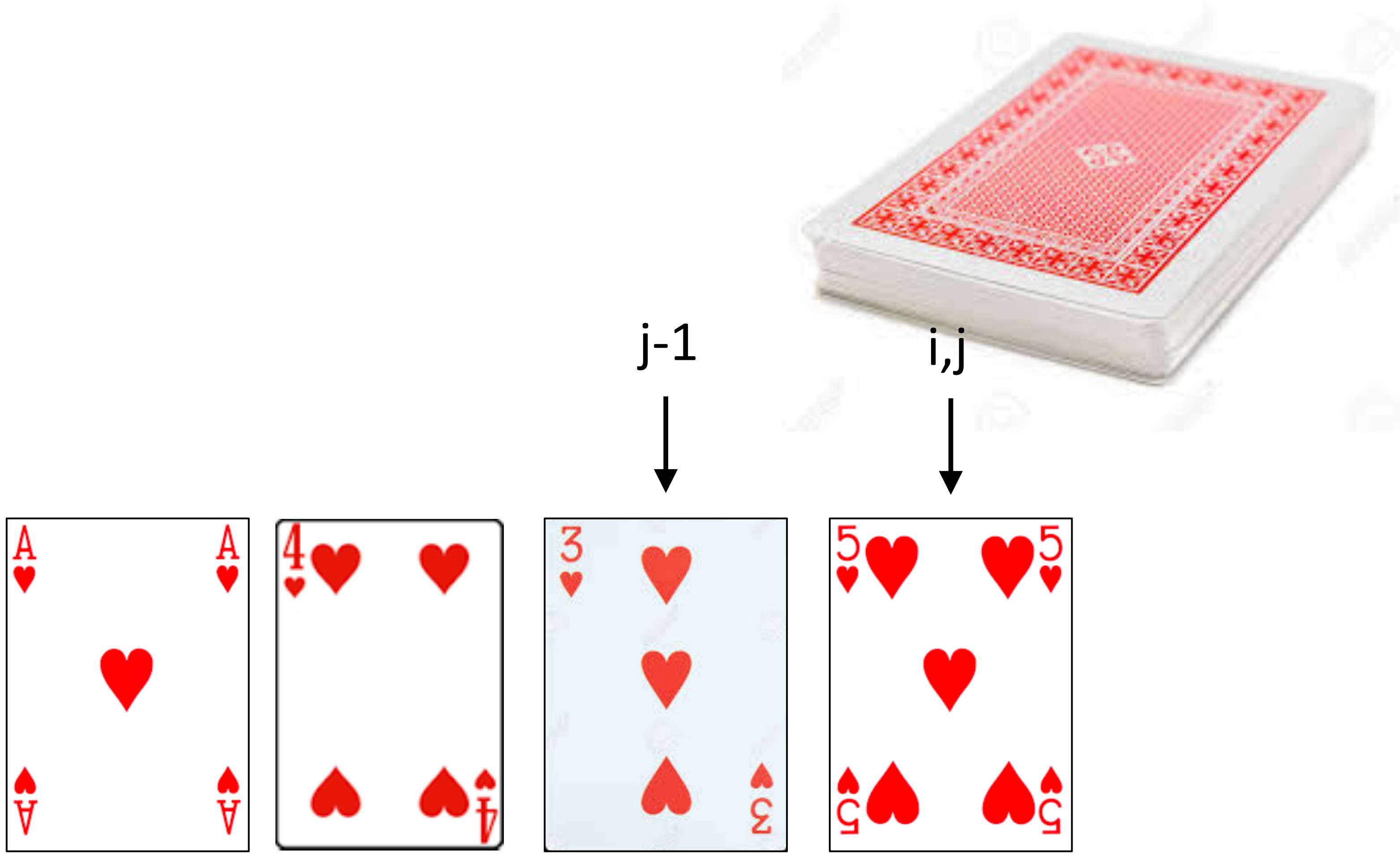
Insertion Sort – How it Works

If the element at a index j is smaller than the element at index j-1, the two elements are swapped



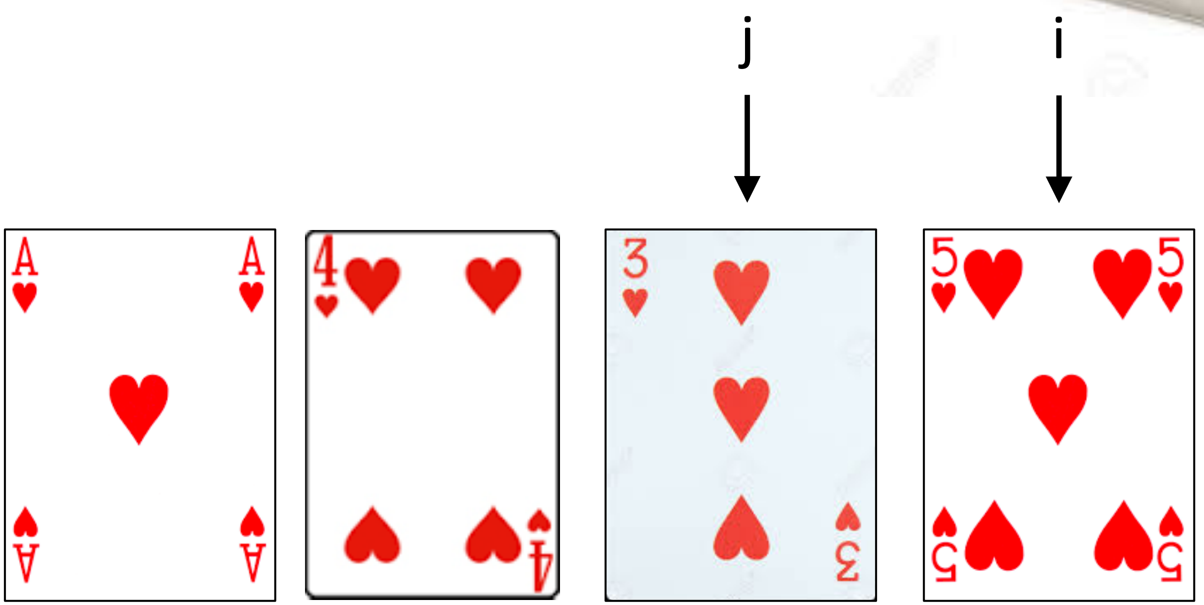


Insertion Sort – How it Works

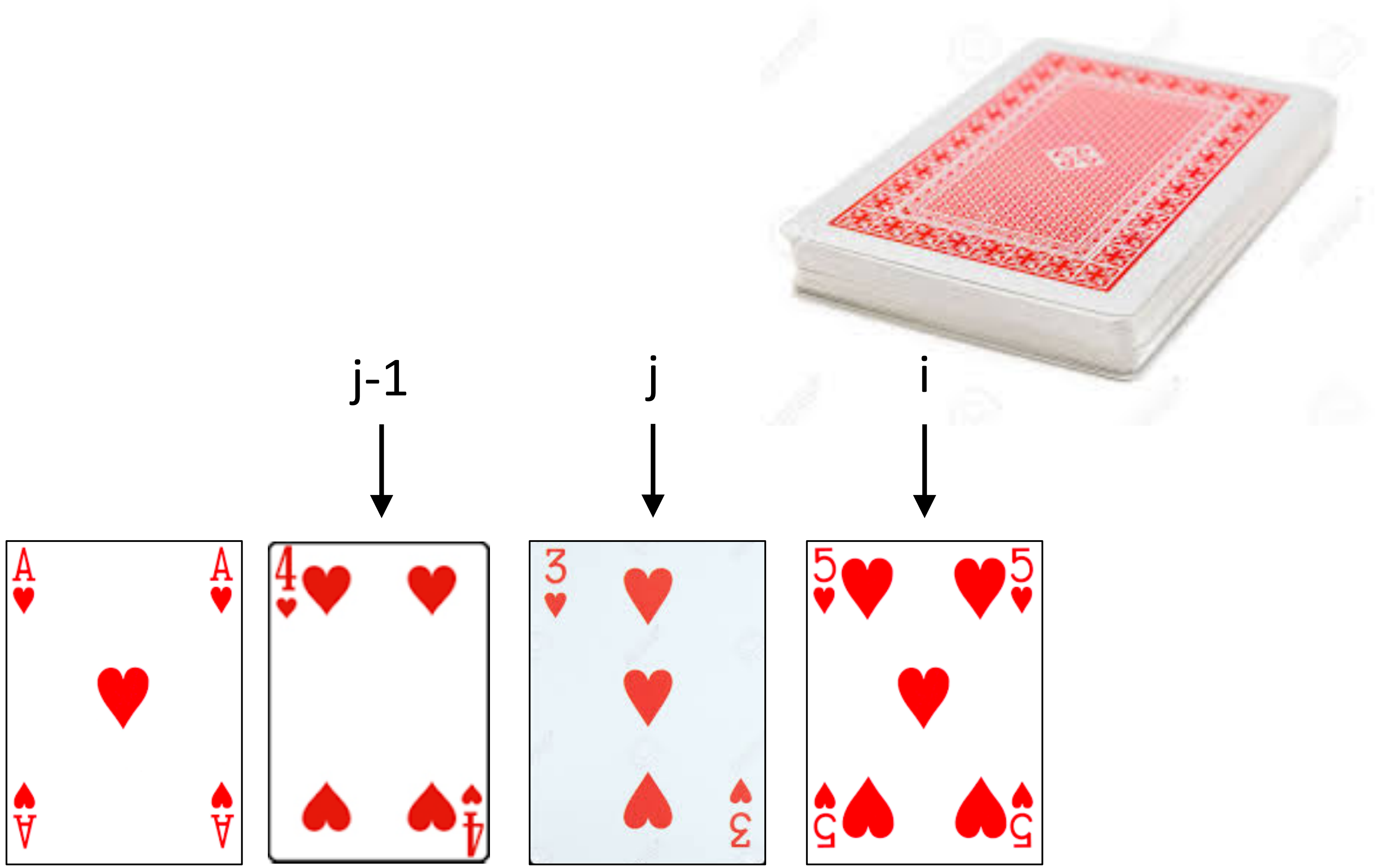


Insertion Sort – How it Works

- Lets' decrement j

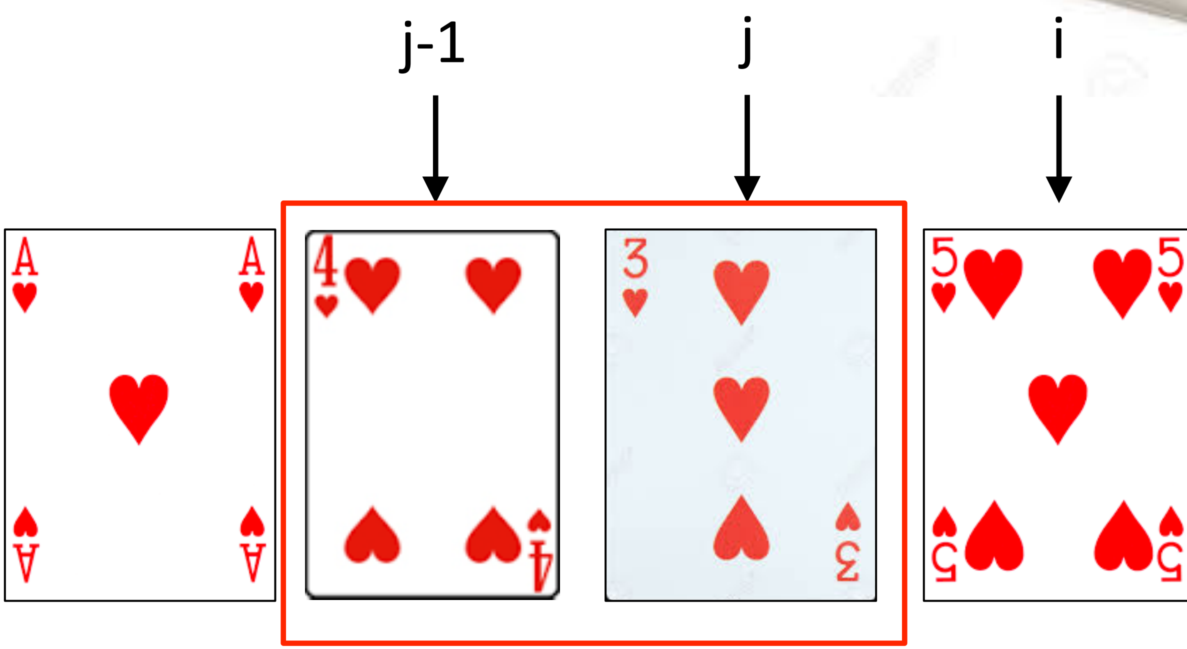


Insertion Sort – How it Works

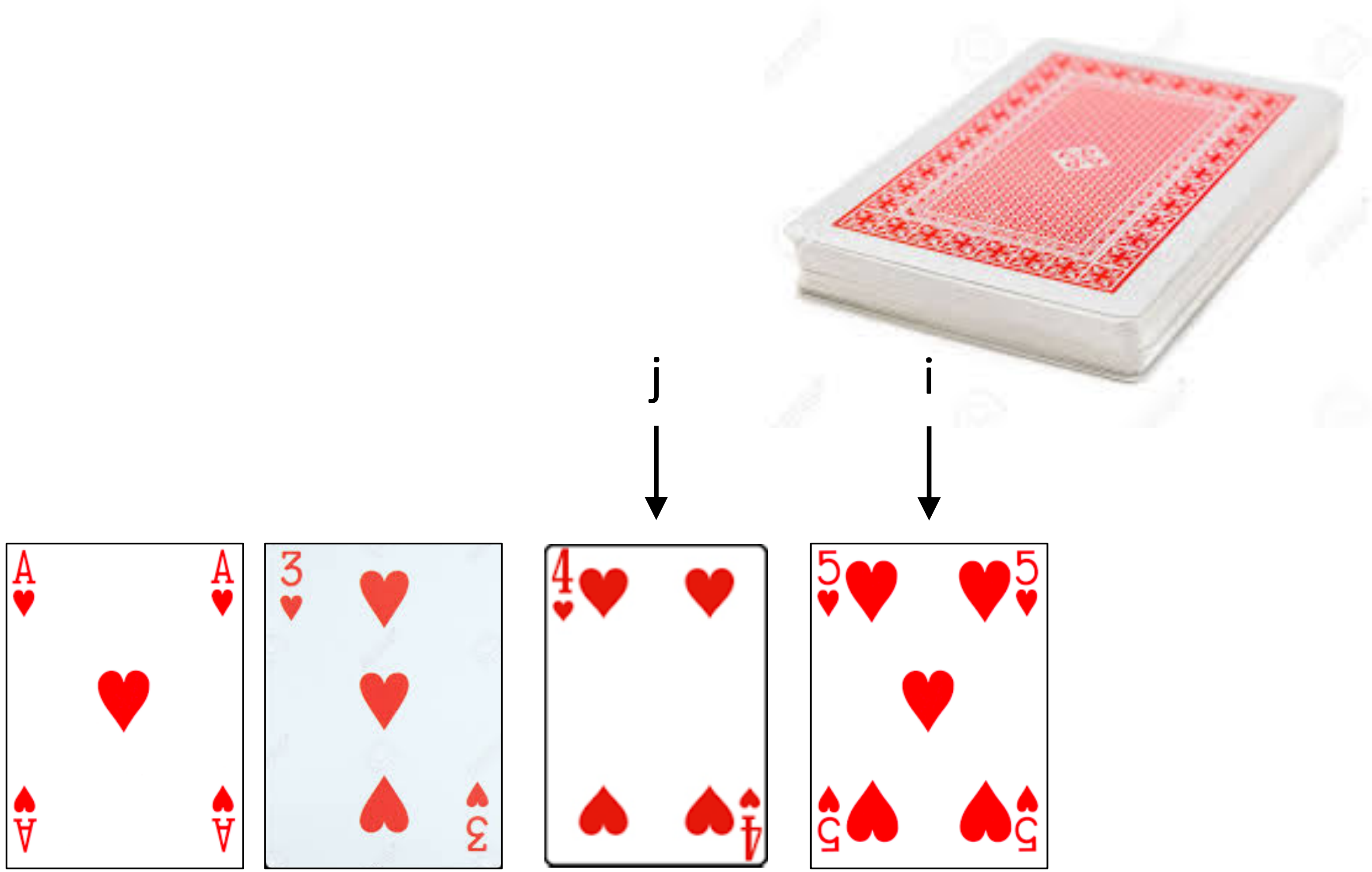


Insertion Sort – How it Works

- If the element at a index j is smaller than the element at index j-1, the two elements are swapped
- Otherwise i is incremented until the end of the array is reached

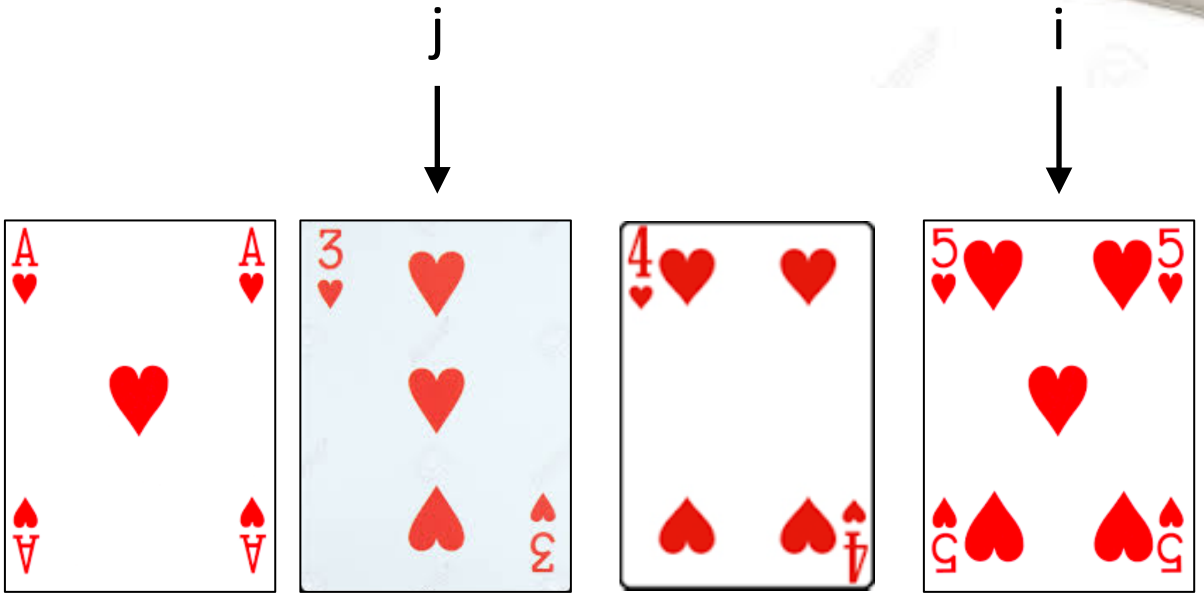


Insertion Sort – How it Works

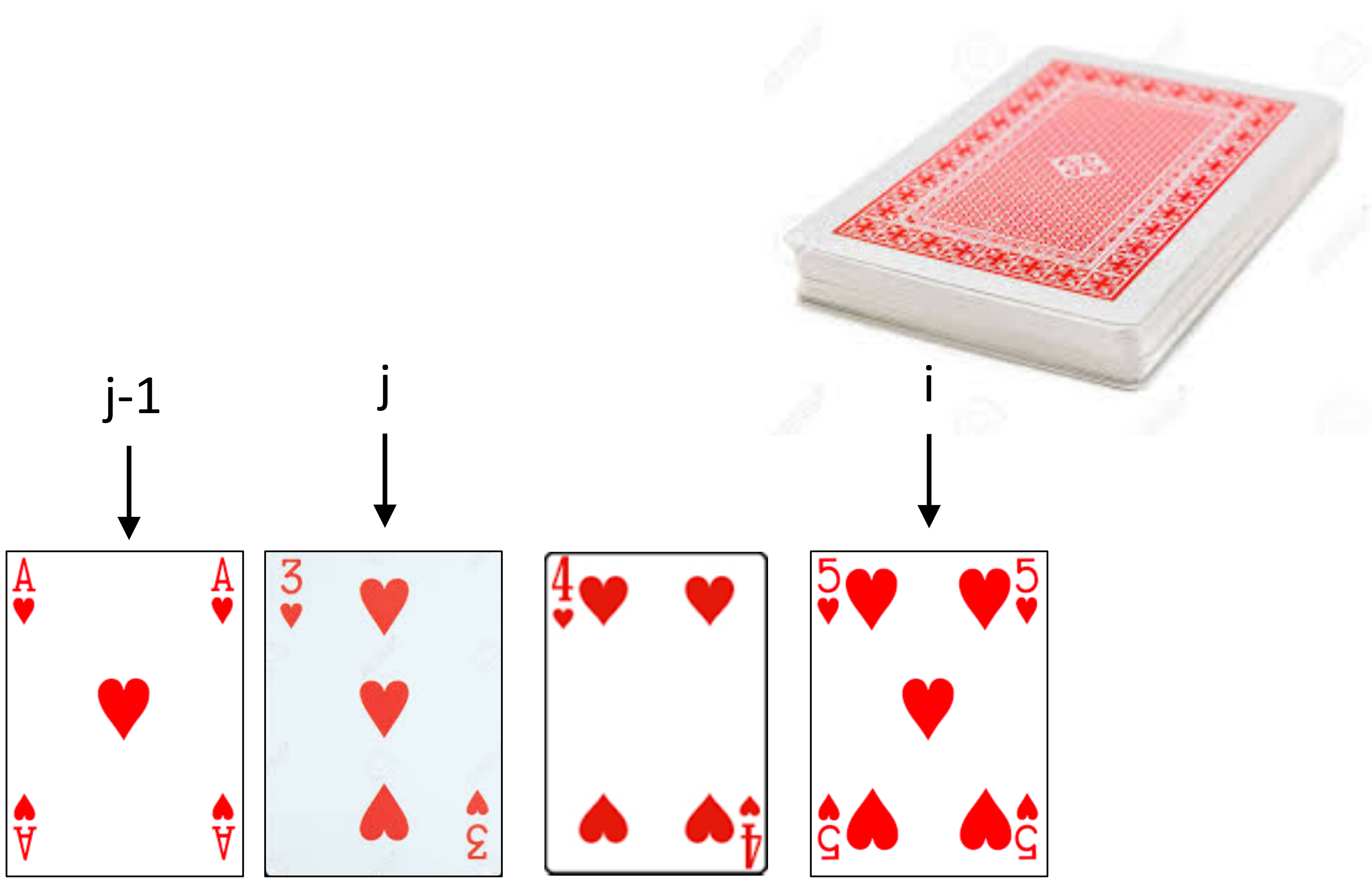


Insertion Sort – How it Works

- Lets' decrement j

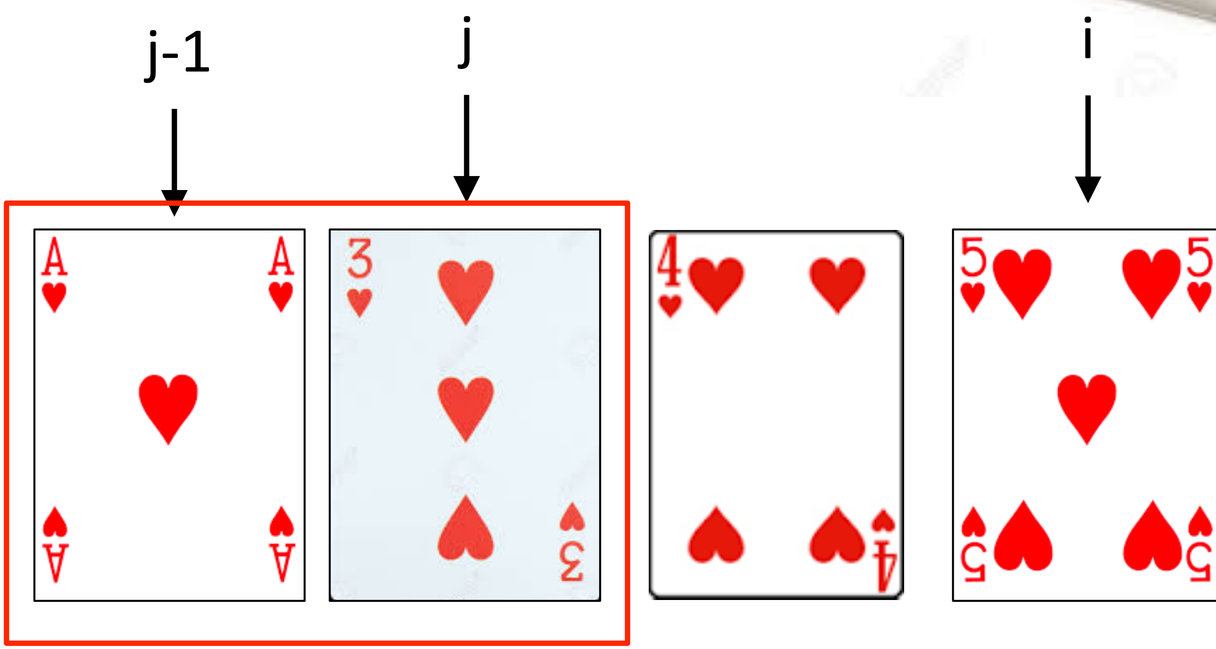


Insertion Sort – How it Works



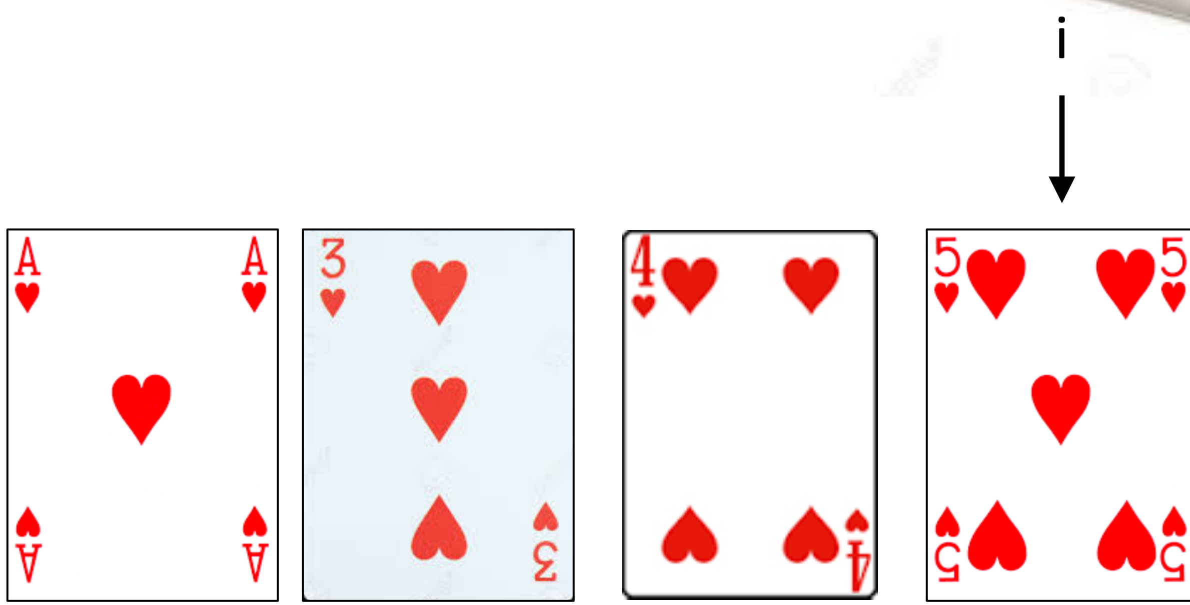
Insertion Sort – How it Works

- If the element at a index j is smaller than the element at index j-1, the two elements are swapped
- Otherwise i is incremented until the end of the array is reached



Insertion Sort – How it Works

- i reached the end of the array so the algorithm terminates



How to Code Insertion Sort?

```
int n, array[100], i, j, swap;
```

variables declaration

```
printf("Enter number of elements\n");
scanf("%d", &n);

printf("Enter %d integers\n", n);

for (i = 0; i < n; i++) {
  scanf("%d", &array[i]);
}

for (i = 1 ; i < n; i++) {
  j= i;

  while ( j > 0 && array[j] < array[j-1]) {
    swap         = array[j];
    array[j]   = array[j-1];
    array[j-1] = swap;

    j--;
  }
}
```

How to Code Insertion Sort?

```
int n, array[100], i, j, swap;

 printf("Enter number of elements\n");
 scanf("%d", &n);

 printf("Enter %d integers\n", n);

 for (i = 0; i < n; i++) {
   scanf("%d", &array[i]);
 }

 for (i = 1 ; i < n; i++) {
   j= i;

   while ( j > 0 && array[j] < array[j-1]) {
     swap           = array[j];
     array[j]    = array[j-1];
     array[j-1] = swap;

     j--;
   }
 }
```

Array insertion

How to Code Insertion Sort?

```
int n, array[100], i, j, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (i = 0; i < n; i++) {
    scanf("%d", &array[i]);
  }

  for (i = 1 ; i < n; i++) {
    j= i;

    while ( j > 0 && array[j] < array[j-1]) {
      swap          = array[j];
      array[j]   = array[j-1];
      array[j-1] = swap;
      j--;
    }
  }
```

- i points to the 2nd element of the array
- j is set = i

How to Code Insertion Sort?

```
int n, array[100], i, j, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (i = 0; i < n; i++) {
    scanf("%d", &array[i]);
  }

  for (i = 1 ; i < n; i++) {
    j= i;

    while ( j > 0 && array[j] < array[j-1]) {
      swap          = array[j];
      array[j]    = array[j-1];
      array[j-1] = swap;
      j--;
    }
  }
```

If the element at a index j is smaller than the element at index j-1, and j > 0 the the elements at index j and (j-1) continue to be swapped

Sorting Algorithms

Insertion Sort

- Simple, but it does not have very good performance
- It still does not have good performance but it might be useful in practice, when the array is already partially sorted.
- It has quadratic performance in the worse case scenario.

Quick Sort

- It has logarithmic performance in the worse case scenario.

Divide and Conquer

- Very important strategy in computer science:
  - Divide a problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- Idea: Partition array into items that are "small" and items that are "large", then sort the two sets recursively → **Quicksort**

Quicksort

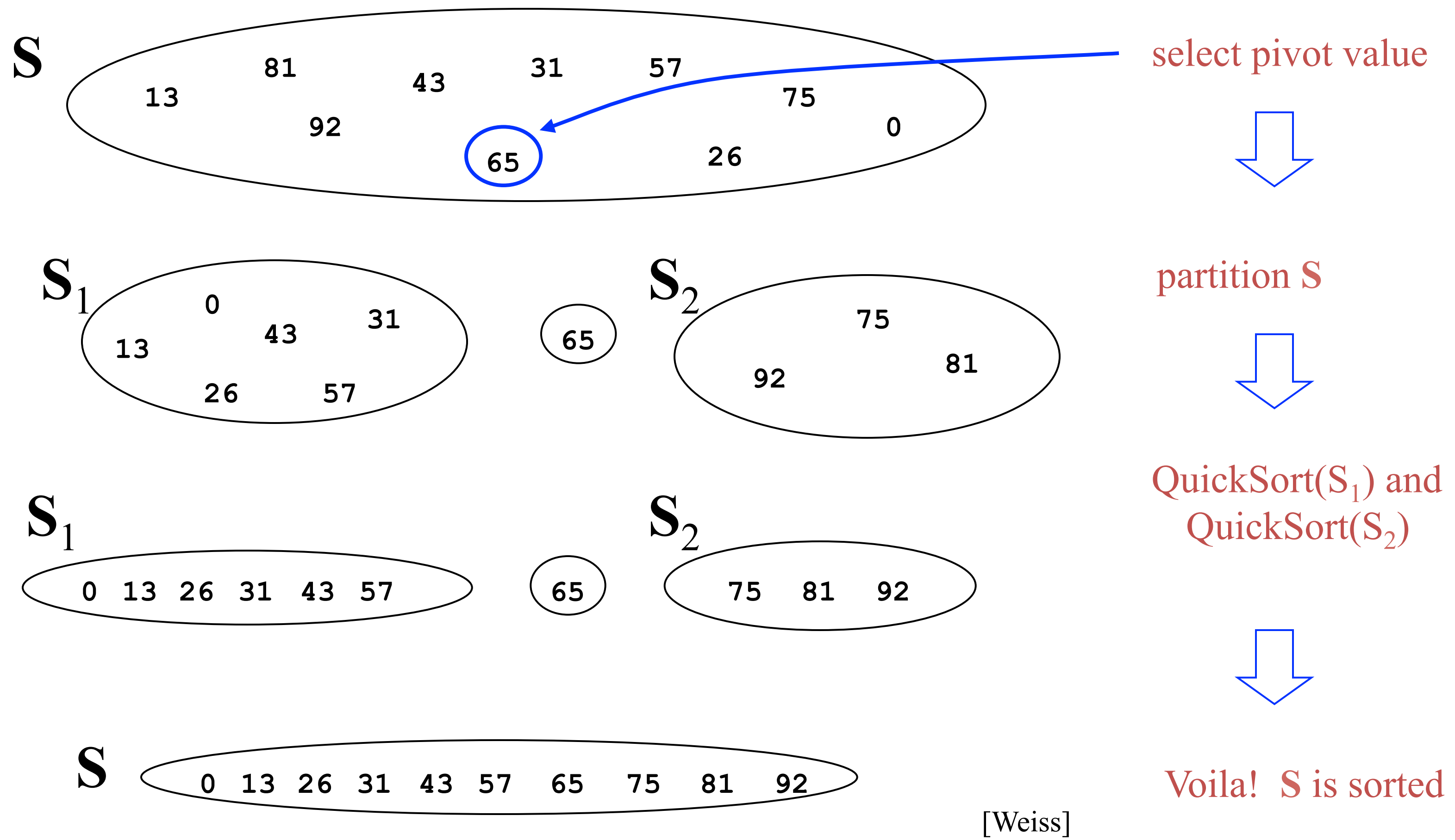
- Quicksort is more space efficient than other algorithms (e.g., MergeSort)
  - Partition array into left and right sub-arrays:
    - Choose an element of the array, called pivot
    - The elements in left sub-array are less than pivot
    - Elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays

“Four easy steps”

- To sort an array **S**
  1. If the number of elements in S is 0 or 1, then return. The array is sorted
  2. Pick an element v in **S**. This is the pivot value
  3. Partition **S**-{*v*} into two disjoint subsets,

     $\mathbf{S}_1$ = {all values $x \le v$}, and $\mathbf{S}_2$ = {all values $x \ge v$}.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The Steps of QuickSort

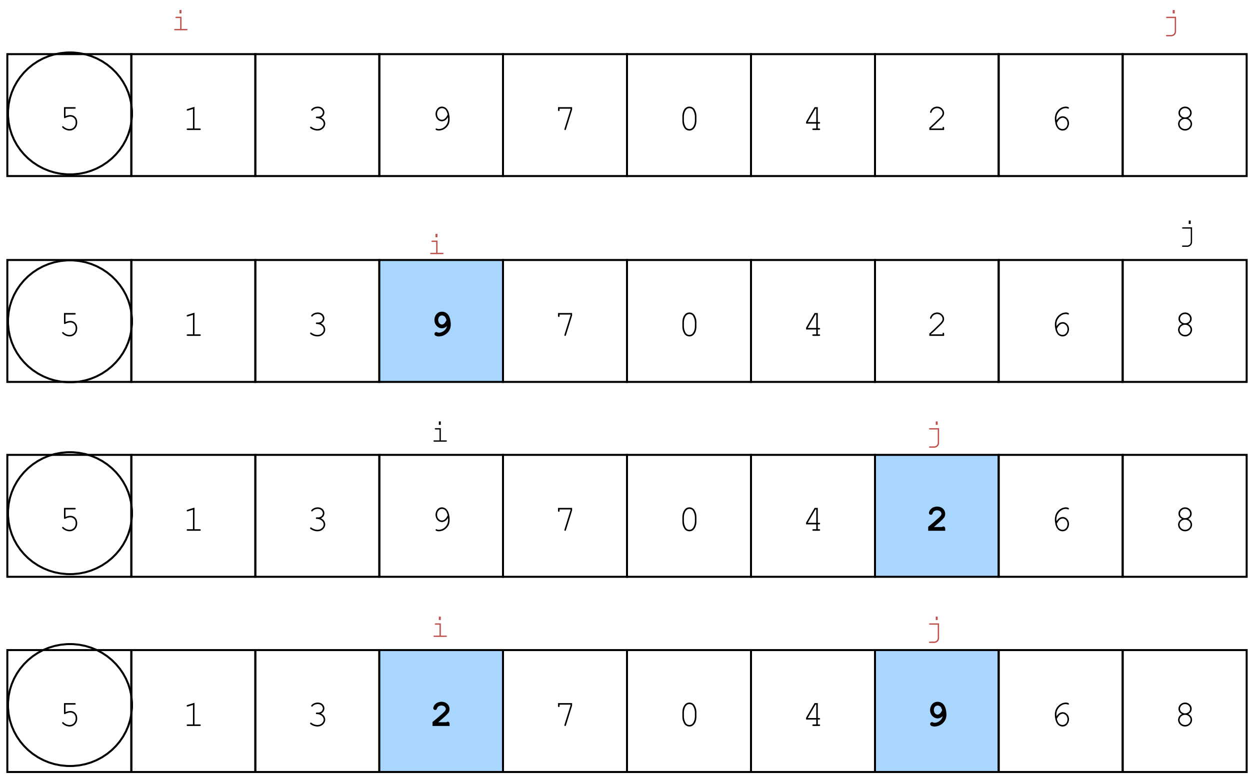


How do we implement it?

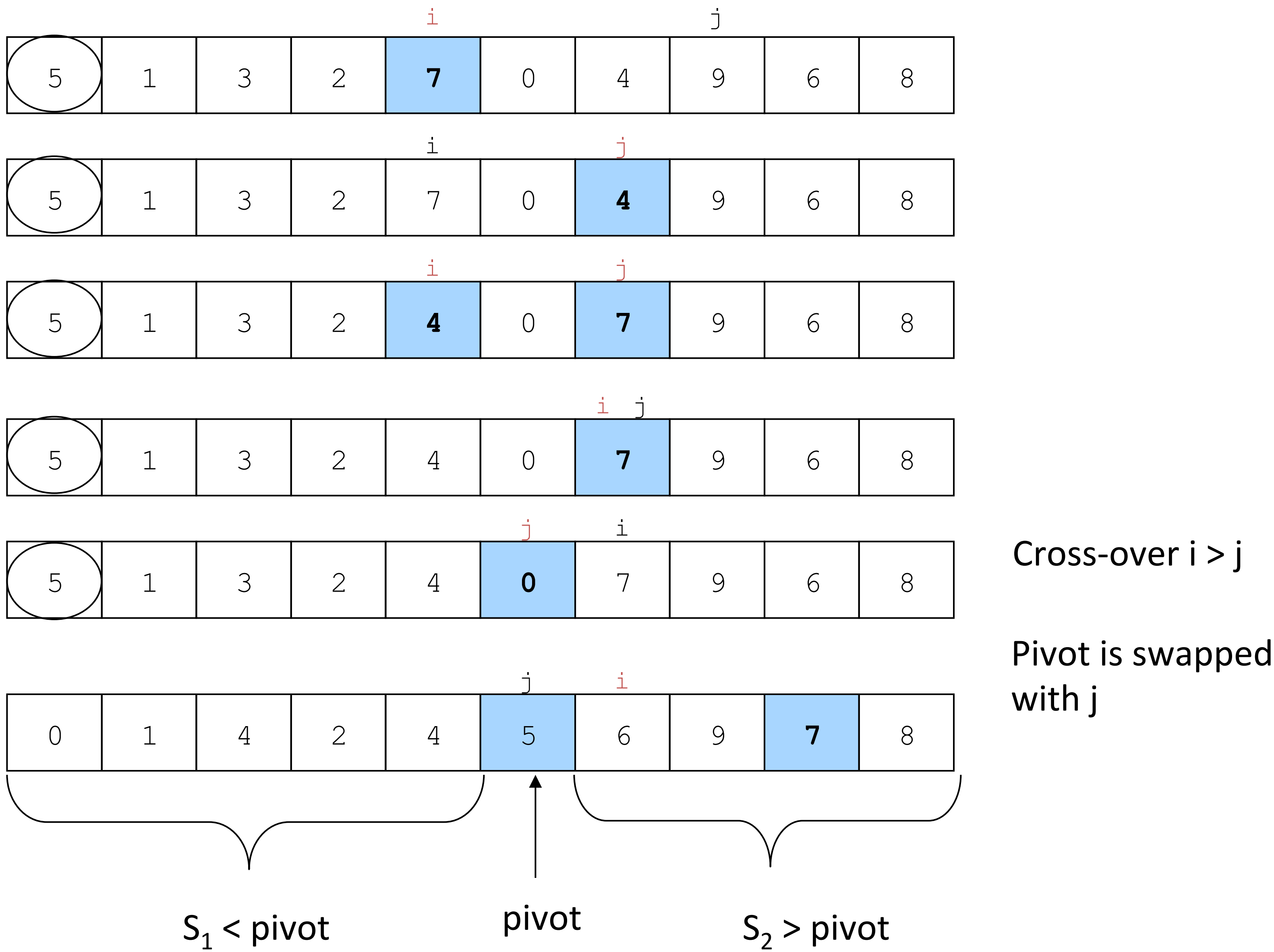
- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

A solution could be to pick the pivot as the first or the median element in an array

In this Implementation, the Pivot is the First Element of the Array (circled)



- Move i to the right to be larger than pivot.
- Move j to the left to be smaller than pivot.
- Swap



i j
5 1 3 2 7 0 4 9 6 8
i j
5 1 3 2 7 0 4 9 6 8
i j
5 1 3 2 4 0 7 9 6 8
i j
5 1 3 2 4 0 7 9 6 8
j i
5 1 3 2 4 0 7 9 6 8
Cross-over i > j
Pivot is swapped with j
j i
0 1 4 2 4 5 6 9 7 8
S_1 < pivot
pivot
S_2 > pivot

Folklore

- "Quicksort is the best in-memory sorting algorithm."
- Truth
  - Quicksort uses very few comparisons on average.
  - Quicksort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality

Quick Sort - Implementation

```c
int main() {
    int i, count, number[25];

    printf("How many elements would you like to enter?: ");
    scanf("%d", &count);

    printf("Enter %d elements: ", count);

    for (i =0; i < count; i++)
        scanf("%d", &number[i]);

    quicksort(number, first: 0, last: count-1);

    printf("Sorted Array:\n");
    for (i =0; i < count; i++)
        printf("%d ", number[i]);
}
```

Invoke quicksort providing as input the array and the index of the first and the last elements

Quick Sort – Implementation

```
void quicksort(int array[], int first, int last){

    //if the size of the array is equal to 0 or 1, the array is sorted by definition
    if(first < last){
        int pivotindex = partition(array, first, last);
        quicksort(array, first, last: pivotindex-1);
        quicksort(array, first: pivotindex+1, last);
    }

}
```

It sorts the array only when the size of the array is greater than 1.

Otherwise, the array is sorted by definition.

Quick Sort – Implementation

```
void quicksort(int array[], int first, int last){

    //if the size of the array is equal to 0 or 1, the array is sorted by definition
    if(first < last){
        int pivotindex = partition(array, first, last);
        quicksort(array, first, last: pivotindex-1);
        quicksort(array, first: pivotindex+1, last);
    }

}
```

Identifies the index of the pivot using function partition

Sorts the element at the left and at the right of the pivot using the quicksort function.

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;    // index of last unknown value
    while (index1 <= index2) { // while some values still unknown
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2);
    int pivot = array[first];    // remember p

    int index1 = first + 1; // index of first
    int index2 = last;    // index of last unk
    while (index1 <= index2) { // while some v
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

Swaps the middle value of the array and considers it as the pivot

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; /
    int index2 = last;     //
    while (index1 <= index2)
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

Considers the portion of the array at the right of the pivot. Remember the pivot is in the first position

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;    // index of last unknown value
    while (index1 <= index2) { // whi
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

This cycle continues until the leftmost index becomes bigger than the rightmost index …

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;    // index of last unknown value
    while (index1 <= index2) { // while some values still unknown
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivo
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

Moves the leftmost index to the right if the pointed value is smaller than the pivot

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];     // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;     // index of last unknown value
    while (index1 <= index2) { // while some values still unknown
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the pivot value between the two
    // sublists and return its index
    return index2;
}
```

Moves the rightmost index to the left if the point
value is bigger than the pivot

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;    // index of last unknown value
    while (index1 <= index2) { // while some values still unknown
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); // put the p
    // sublists and return its index
    return index2;
}
```

Otherwise, the elements pointed by the leftmost and the rightmost index are swapped.
The leftmost index is incremented and the rightmost index is decremented

Quick Sort – Auxiliary Functions

```
int partition(int array[], int first, int last) {
    swap(array, first, j: (first + last) / 2); // swap middle value into first pos
    int pivot = array[first];    // remember pivot

    int index1 = first + 1; // index of first unknown value
    int index2 = last;    // index of last unknown value
    while (index1 <= index2) { // while some values still unknown
        if (array[index1] <= pivot)
            index1++;
        else if (array[index2] > pivot)
            index2--;
        else {
            swap(array, index1, index2);
            index1++;
            index2--;
        }
    }
    swap(array, first, index2); /
    // sublists and return its in
    return index2;
}
```

The pivot is swapped with the rightmost index

The index of the pivot is returned