Lecture 8

DVA104 Data Structures, Algorithms, and Program Design

Abu Naser Masud

Masud.abunaser@mdh.se

Reading Materials

- Introduction to Algorithms, Corman et al, chapter 7 (read the relevant sections)
- Data structure using C, Reema Thareja, chapter 14
  PDF:
http://indexof.es/Miscellanous/Data%20Structures%20Using%20C,%202nd%20edition.pdf
- Read relevant sections from the book that are taught in the class.

Topics

Sorting:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quicksort

Sorting

Sorting means *arranging* the elements of an array in some relevant order which may be either *ascending* or *descending*. We formulate the sorting problem as follows*:*

Input: A sequence of n numbers $<a_1,a_2,\ldots,a_n>$

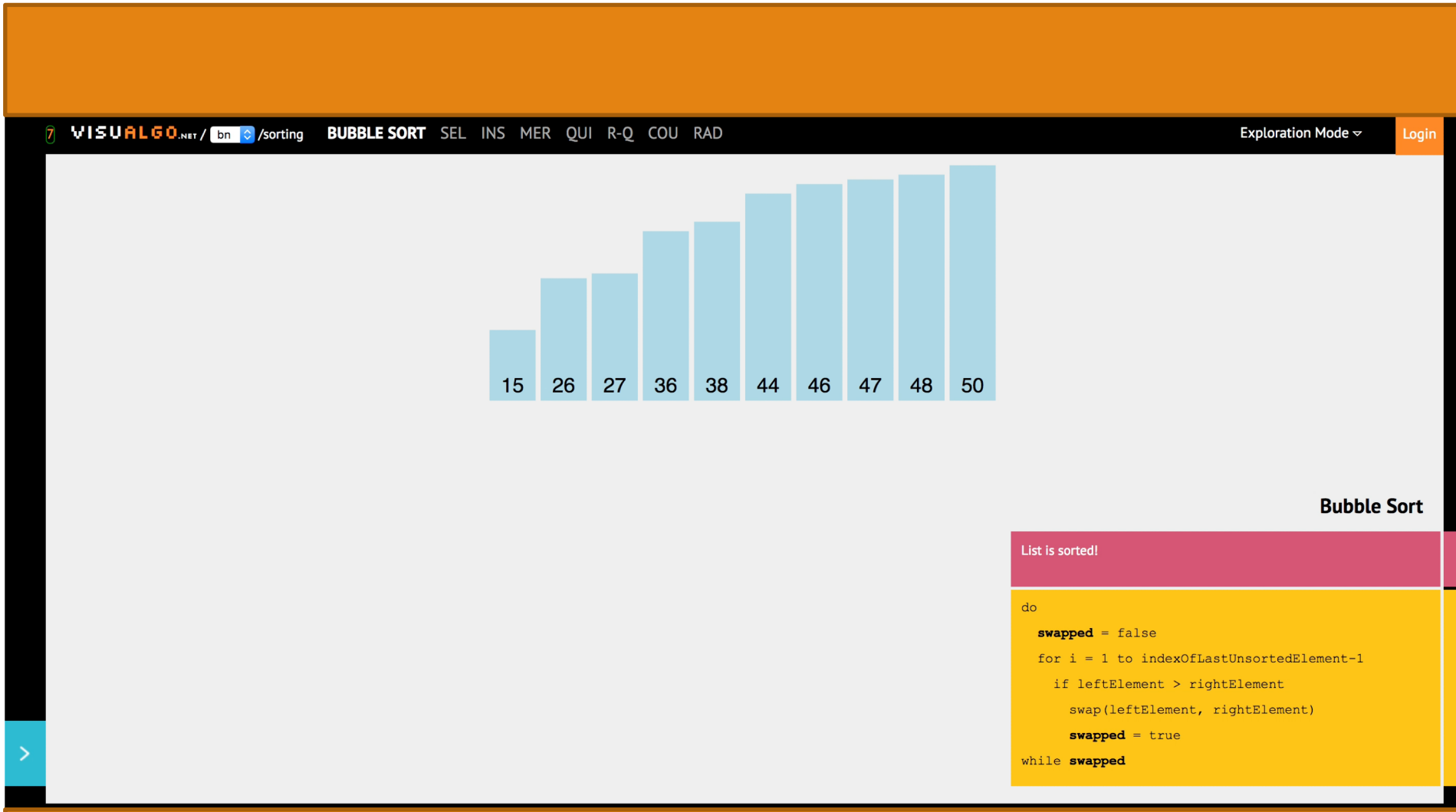
Output: A permutation (reordering) $<b_1,b_2,\ldots,b_n>$ of the input sequence such that $b_1 \leq b_2 \leq \ldots \leq b_n$

Bubble Sort

It is a very simple sorting method that works as follows (sorting in ascending order):

- Compare adjacent pair of array elements
- If the element at the lower index is greater than the element at the higher index, then swap their position
- Repeat the above two steps until all the elements of the array are sorted

Bubble Sort


VISUALGO.NET / bn /sorting
BUBBLE SORT SEL INS MER QUI R-Q COU RAD
Exploration Mode
Login
15 26 27 36 38 44 46 47 48 50
Bubble Sort
List is sorted!
do
swapped = false
for i = 1 to indexOfLastUnsortedElement-1
if leftElement > rightElement
swap(leftElement, rightElement)
swapped = true
while swapped

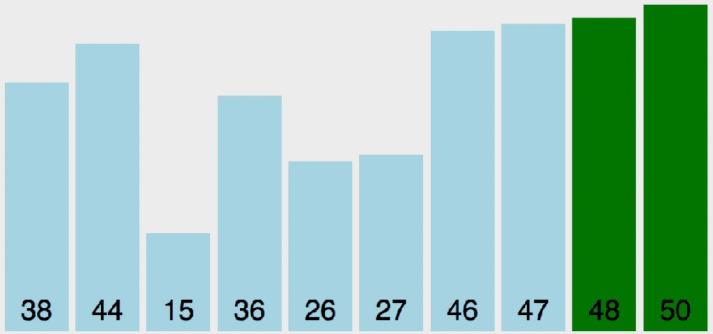
Bubble Sort

```
do
    swaped = false
    for i=1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
            swap (leftElement, rightElement)
            swaped = true
While swaped
```

What happens after one iteration of the inner loop?

and, what happens after the second iteration?

Bubble Sort



Bubble Sort

Swapping the positions of 50 and 48.
Set **swapped** = true.

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

Bubble Sort- Algorithm

```
do
    swaped = false
    for i=1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
            swap (leftElement, rightElement)
            swaped = true
While swaped
```

What happens after one iteration of the inner loop?

When should the outer loop terminate?

Bubble Sort - Complexity

- Best Case Scenario:

  The array is already sorted. The best case asymptotic complexity of this algorithm is O(n) where n is the size of the array. Why?

- Worst case scenario:

  The array is sorted in descending order. The worst case asymptotic complexity is $O(n^2)$. Why?

- But in the worst-case it makes fewer comparison than the presented algorithm. Can you make the presented algorithm even smarter?

Bubble Sort

Can you make the presented algorithm even smarter?

- By the end of the *first* iteration of the inner loop, the *biggest* element is set at the *last index* of the array.
- By the end of the *second* iteration of the inner loop, the *second biggest* element is set at the *second last index* of the array.
- By the end of the *i-th* iteration of the inner loop, the *i-th biggest* element is set at the *ith-last index* of the array.

Do you see a pattern? We can avoid doing comparison for the elements that are already in place.

Bubble Sort

Summary:

Iterate through the array and swap adjacent elements that are not in the right order. Repeat until the array is sorted.

The name comes from the highest element "bubbling up" to the end of the array (or vice versa if you sort the other way)

Characteristics:
Complexity in the worst/average case: $O(n^2)$.
Complexity in the best case: $O(n)$

Easy to understand and implement. Determines quickly if data is sorted.

Reading Material:

1. Data structures using C, Reema Thareja, (Chapter 14.7)

Insertion Sort

Insertion sort is an efficient algorithm for sorting a small number of elements. The process is very much similar to how people arrange a hand of poker cards:

We pick a card from the table, and put it in the right place at hand. The cards at hand are sorted, the cards at the table is unsorted.

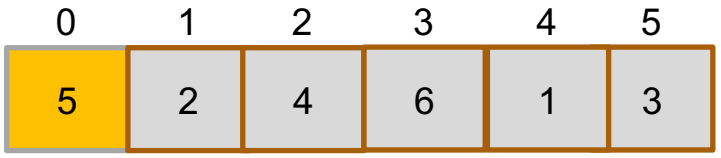
Insertion Sort - Algorithm

Algorithm:

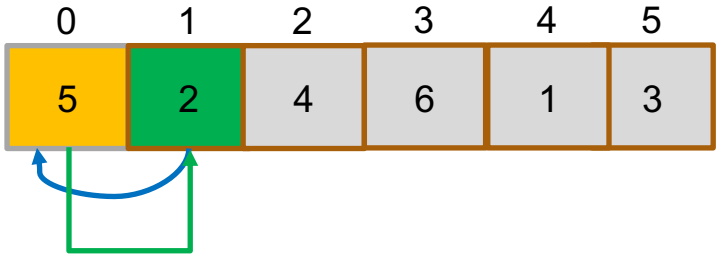
- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set. (Outer loop)
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set. (Inner loop)

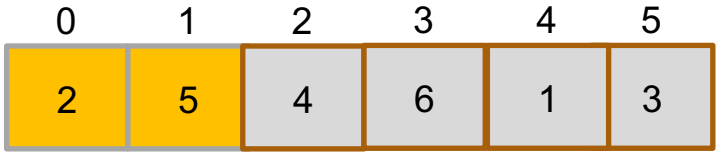
Insertion Sort - Example

Initialization:

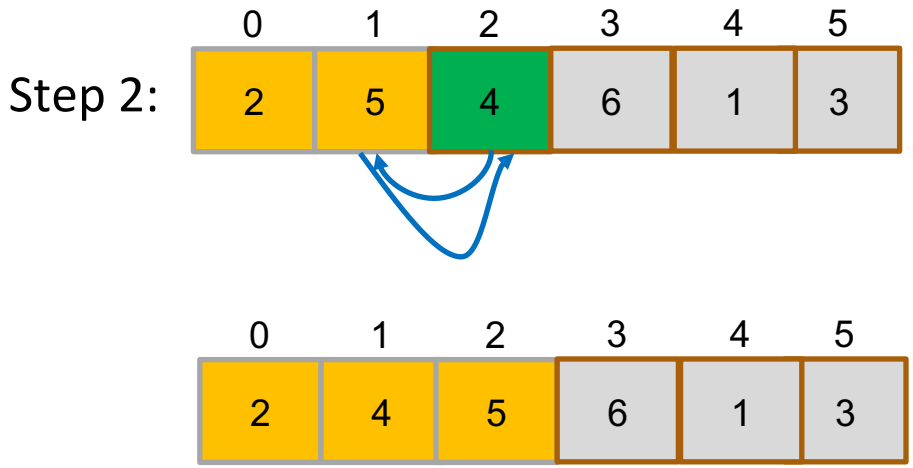


Step 1:





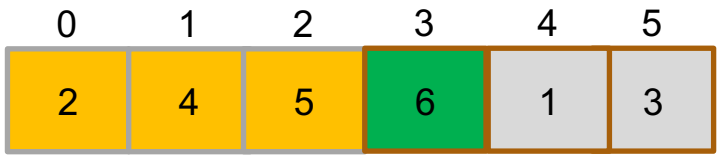
Insertion Sort - Example

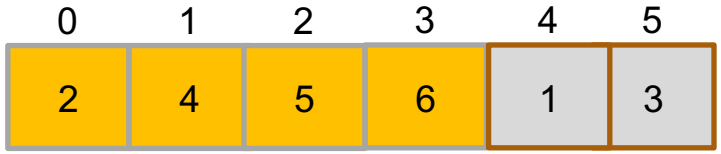


- The size of the sorted array is increasing, and the size of unsorted array is decreasing

Insertion Sort - Example

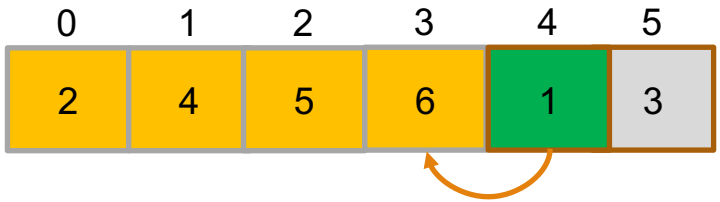
Step 3:

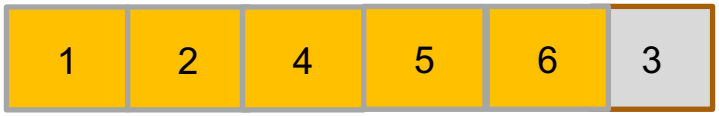


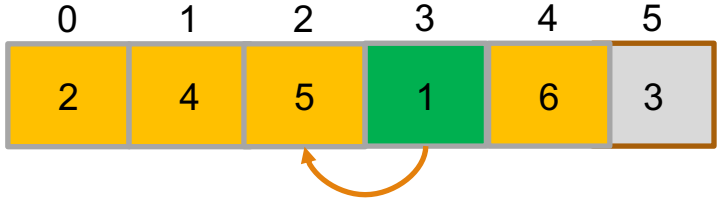


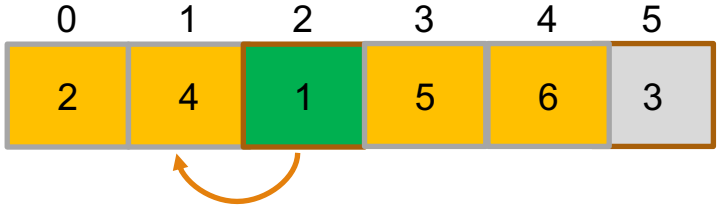
Insertion Sort - Example

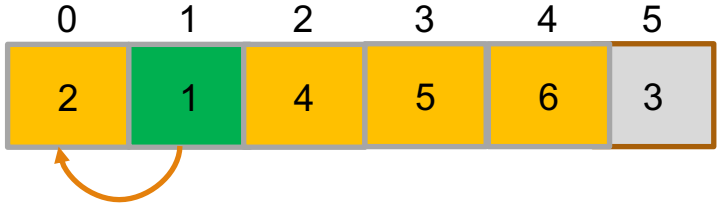
Step 4:





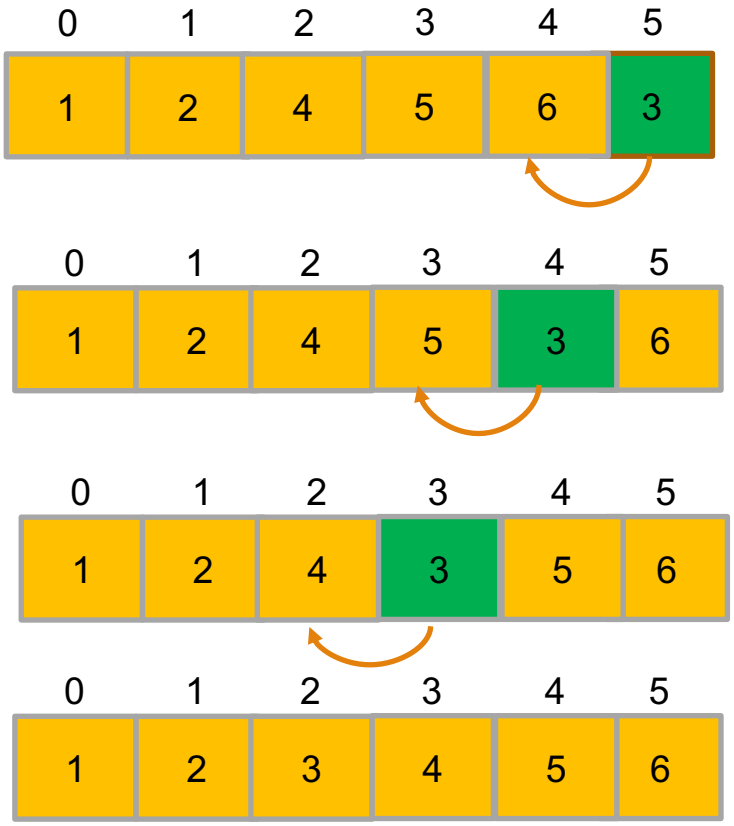






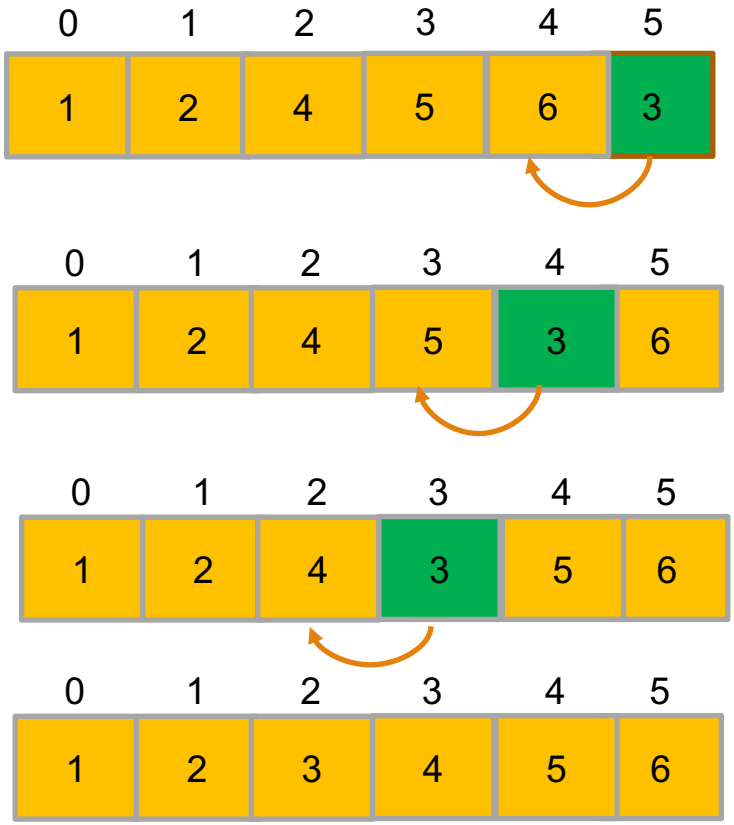
Insertion Sort - Example

Step 5:



Insertion Sort -Complexity

Step 5:



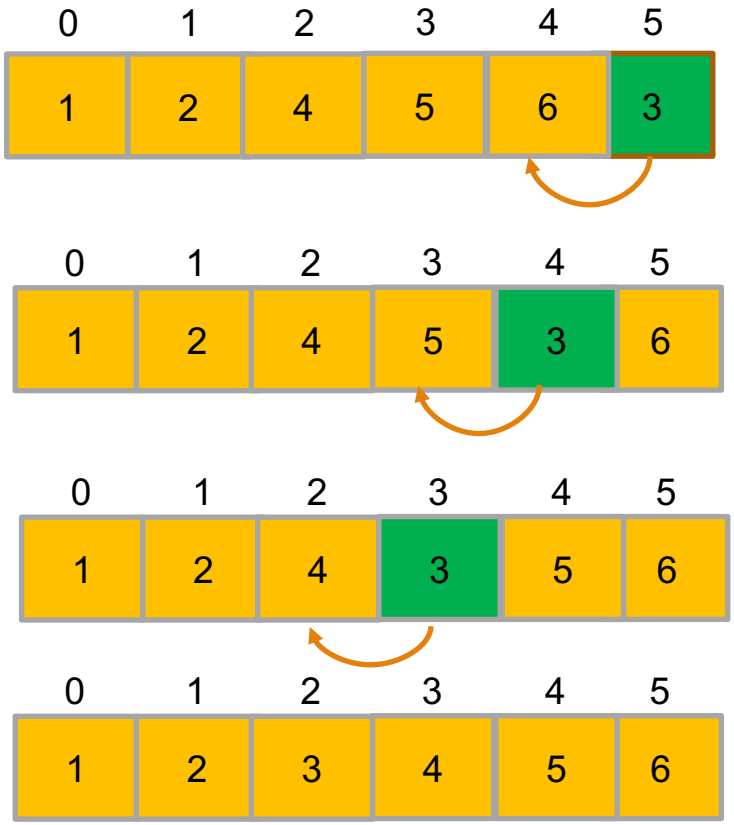
- In each step, first element of the unsorted array is picked to place in the sorted array.

How many times we can pick the first element of an unsorted array?

n-2 times where n is the array length

Insertion Sort -Complexity

Step 5:



- What is the maximum number of steps required to place the first element from the unsorted array into the sorted array?

n-1 steps (worst case)
1 steps (best case)

Insertion Sort -Complexity

- In each step, first element of the unsorted array is picked to place in the sorted array.
  How many times we can pick the first element of an unsorted array?

  n-2 times where n is the array length

- What is the maximum number of steps required to place the first element from the unsorted array into the sorted array?
  n-1 steps (worst case)
  1 steps (best case)

- Worst Case time complexity:

  $O((n-2)*(n-1)) = O(n^2-3n+2) = O(n^2)$

- Best case time complexity:
  $O((n-2)*1) = O(n)$

Merge Sort

- Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.

- The solution follows the *divide-and-conquer* approach:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.

Merge Sort

- The merge sort algorithm closely follows the *divide-and-conquer* paradigm. Intuitively, it operates as follows.
  - **Divide:** Divide the *n-element sequence* to be sorted into *two subsequences* of n/2 elements each.
  - **Conquer:** Sort the two subsequences recursively using merge sort.
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer

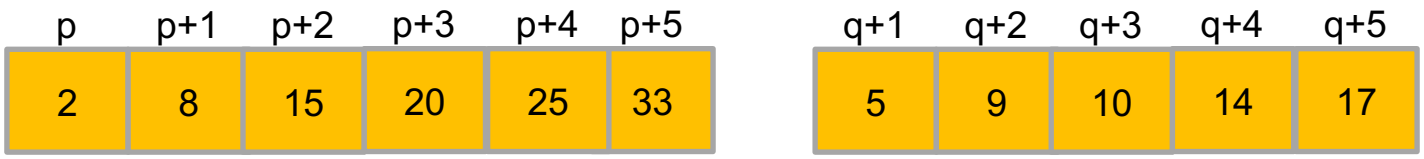
Merge Sort

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step

We use the auxiliary procedure `merge(A,p,q,s)` where

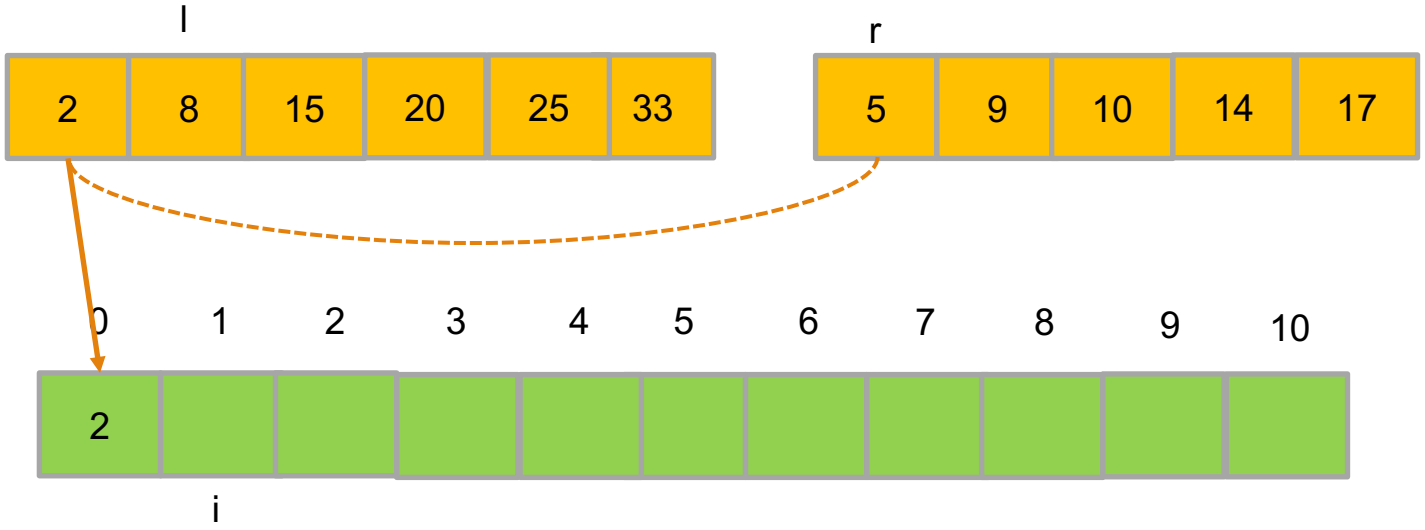
- `A` is the array to be sorted
- `p,q,s` are the array indices such that $p \leq q < s$
- The merge procedure assumes that the subarray `A[p..q]` and `A[q+1,s]` are sorted

merge(A,p,q,s)



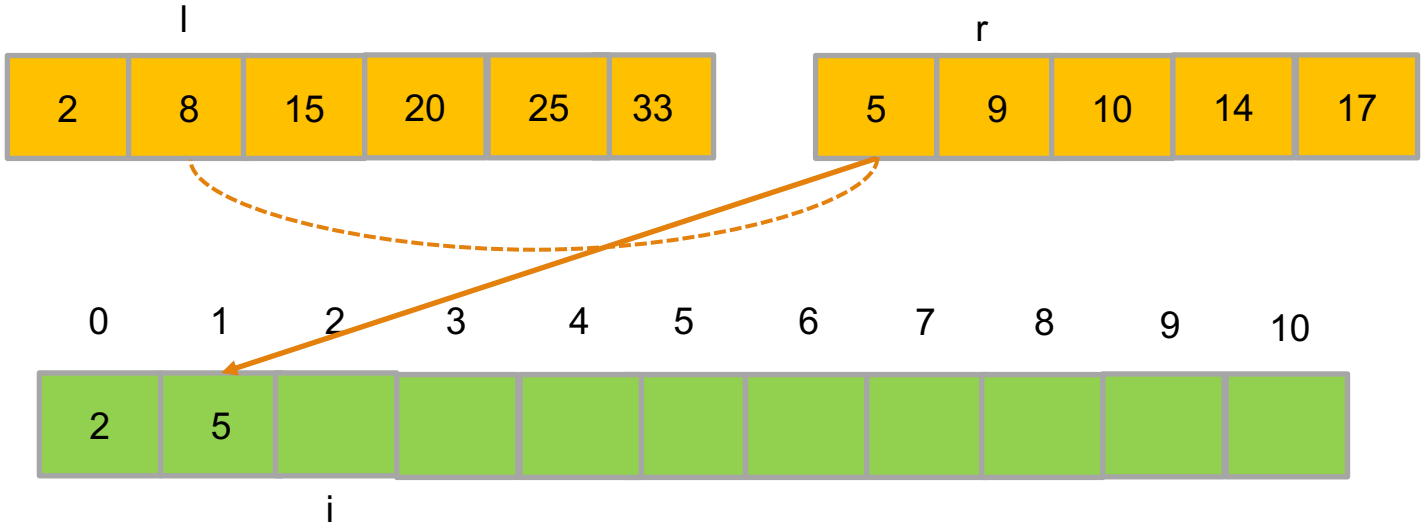
- We take a temporary array `Temp` where we shall merge the two sorted subarray.
- Suppose `i` is the index variable of `Temp` which starts from `0`. Let's assume two index variable `l=p` and `r=q+1` for the left subarray and right subarray.
- We shall copy data from `A` to `Temp` in ascending order

merge(A,p,q,s)



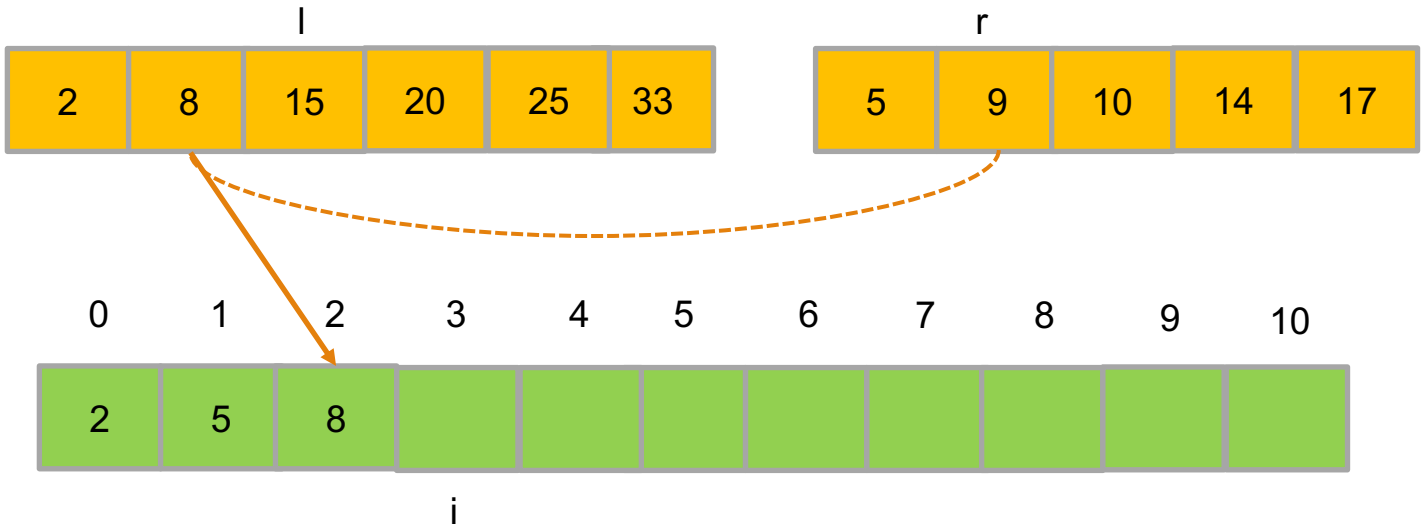
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



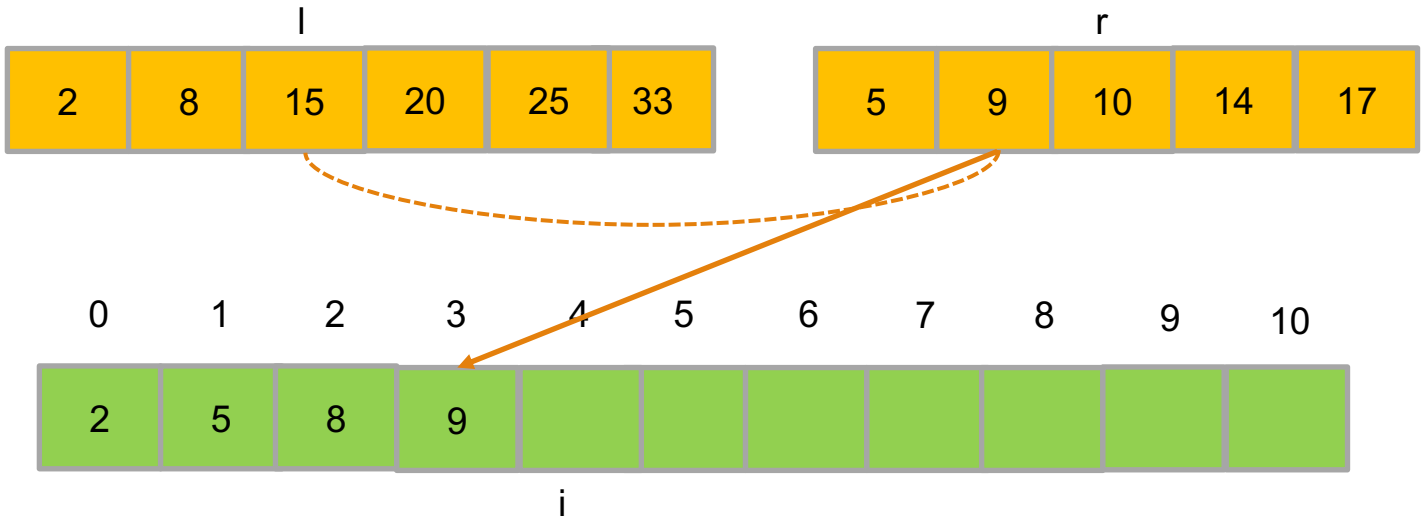
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



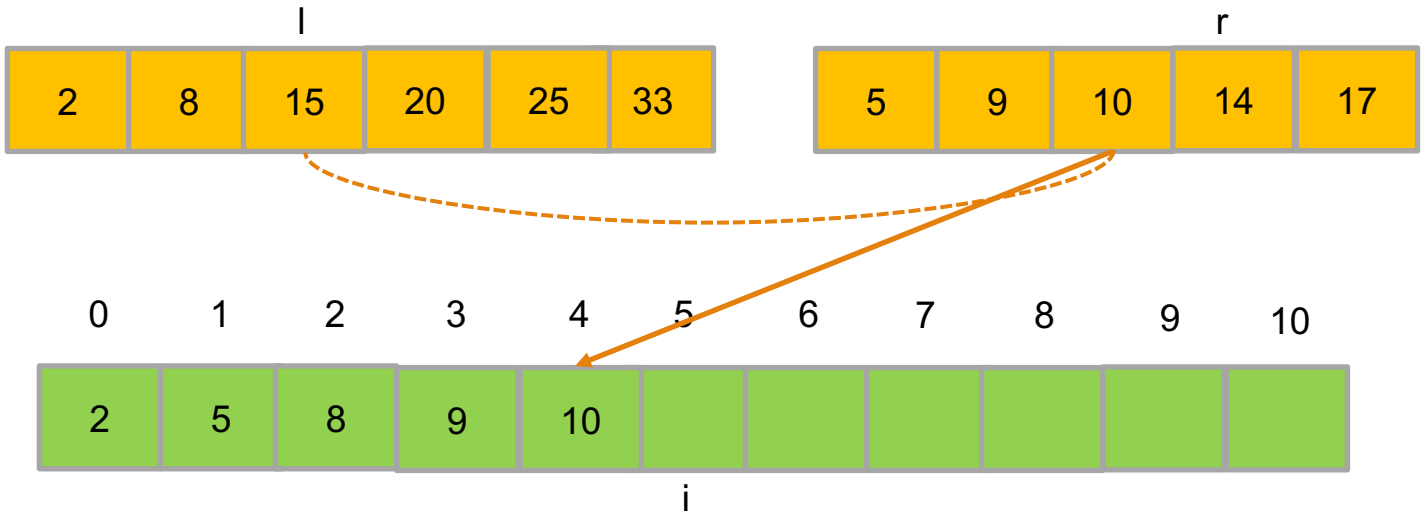
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



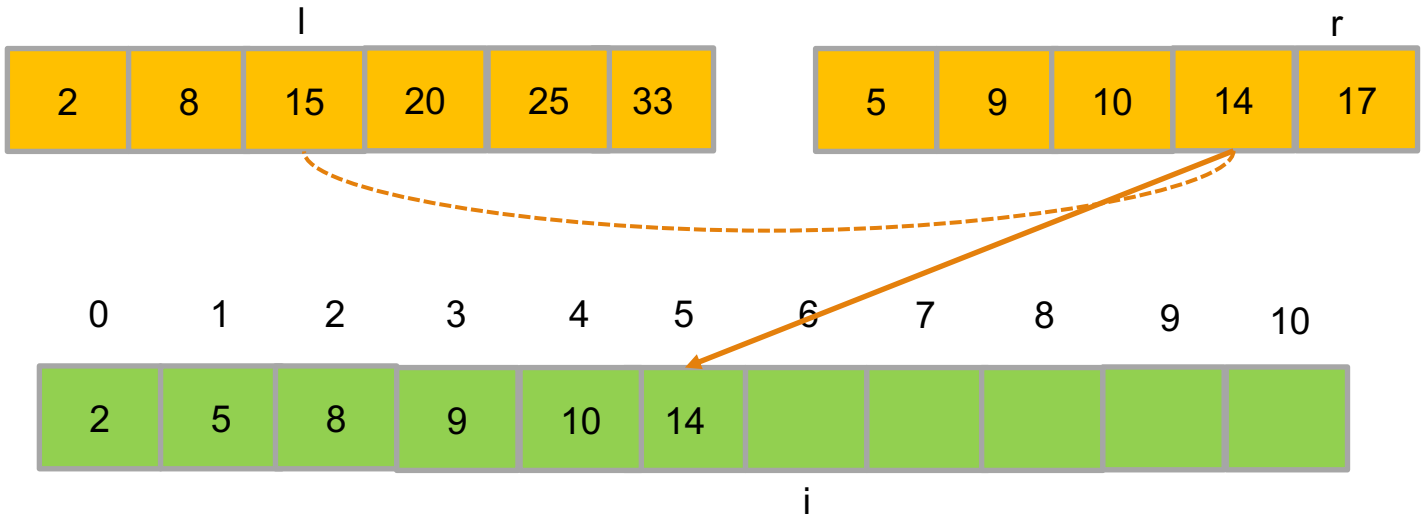
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



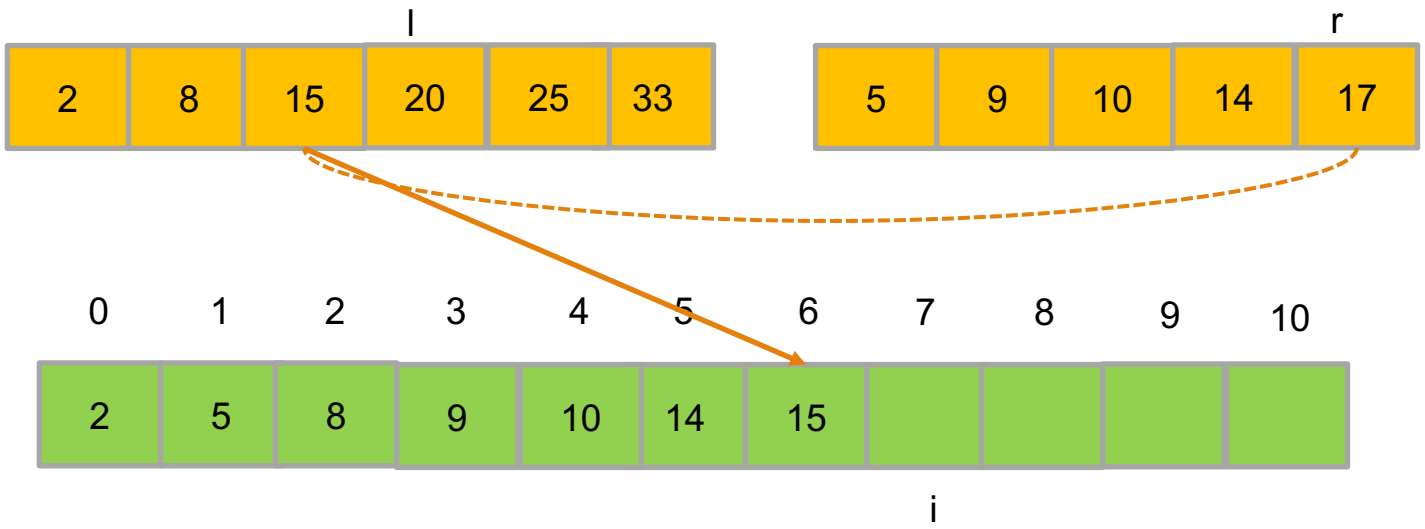
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



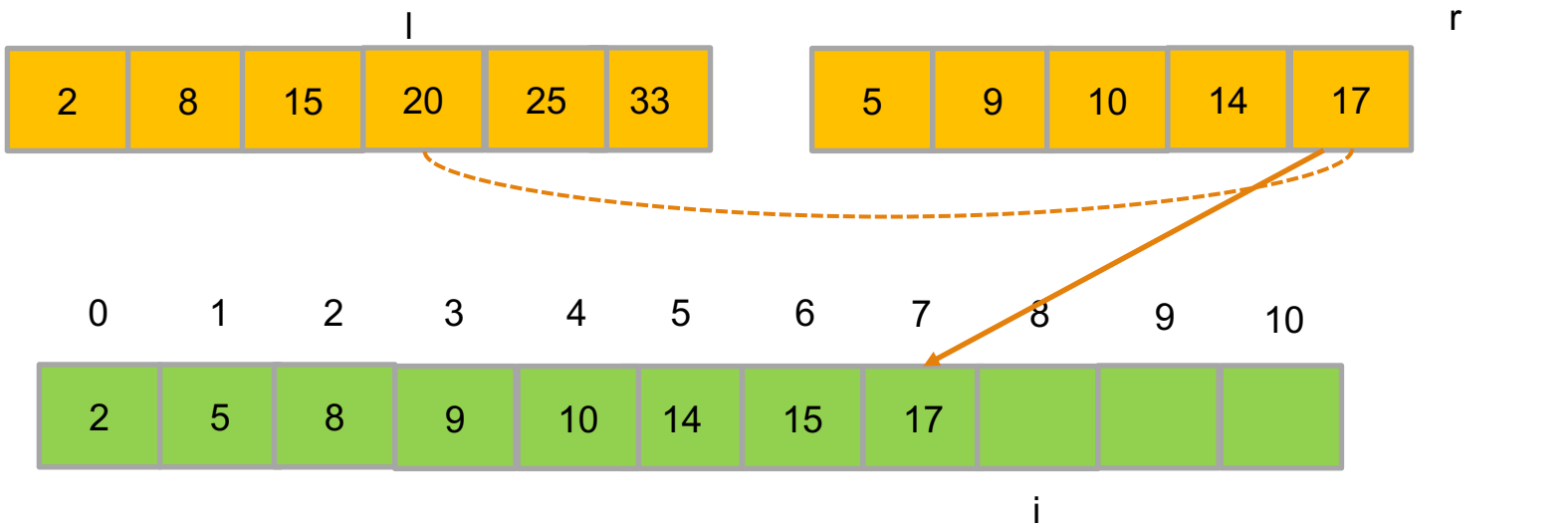
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



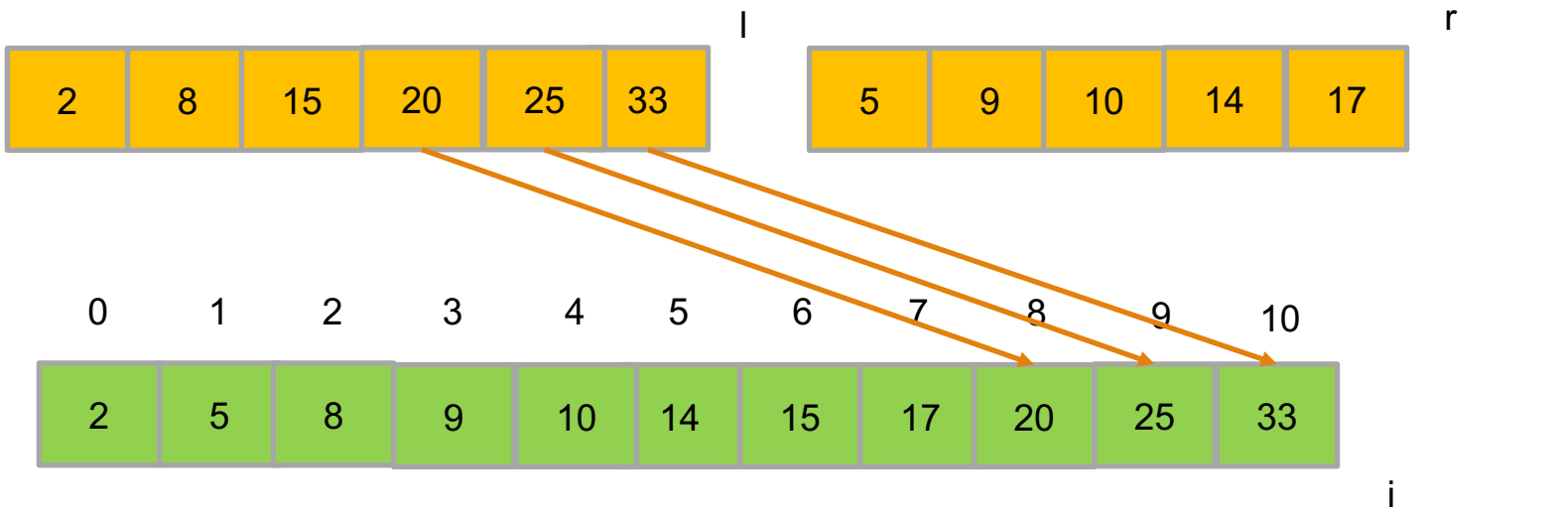
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



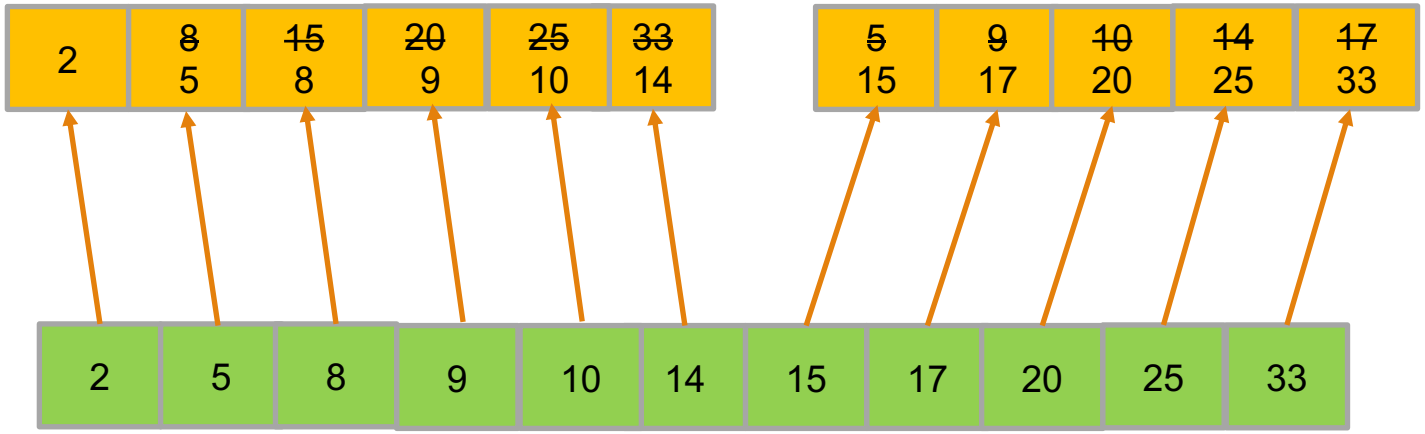
- **Compare** A[l] with A[r]
- **Set** Temp[i] = A[l] **if** A[l] < A[r]
- **Else** Temp[i] = A[r]
- **Increment** index l or r depending on from where Temp[i] gets value
- **Increment** index i

merge(A,p,q,s)



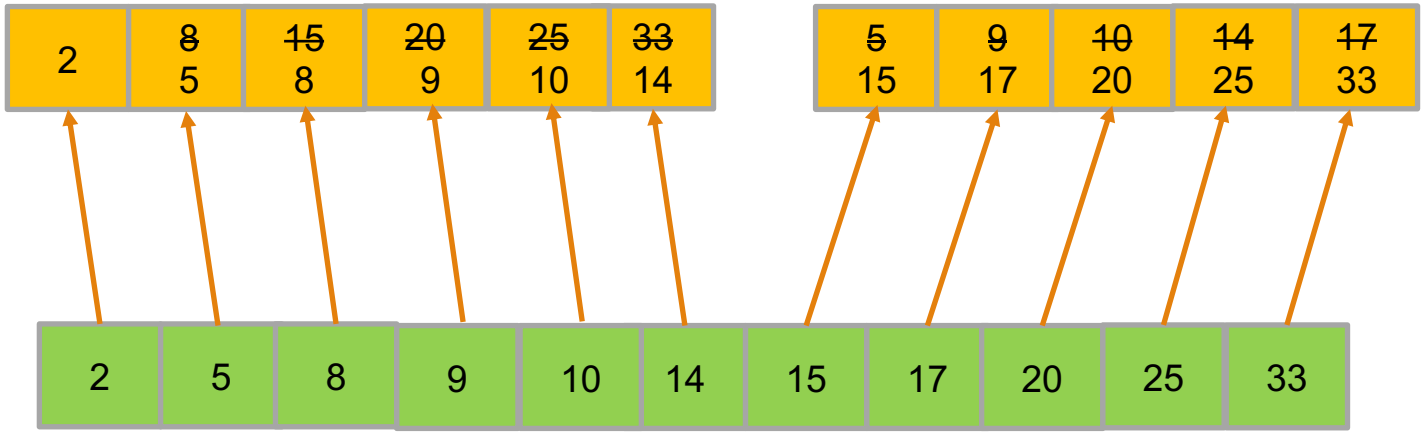
- Now `r>s` meaning that the right subarry has no more elements
- We could have l>q meaning that the left subarray has no more elements
- Copy the rest of the elements of the subarray that has not finished yet to `Temp`

merge(A,p,q,s)



- Now copy Temp to A back
- What should be the run time complexity of `merge(A,p,q,s)`?

merge(A,p,q,s)



- What should be the run time complexity of `merge(A,p,q,s)`?
- We run through each element of the left and right subarray only once. So, the complexity is dominated by the number `s-p+1`

Merge Sort

- The merge sort algorithm closely follows the *divide-and-conquer* paradigm. Intuitively, it operates as follows.
  - **Divide:** Divide the *n-element sequence* to be sorted into *two subsequences* of n/2 elements each.
  - **Conquer:** Sort the two subsequences recursively using merge sort.
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer

Merge Sort

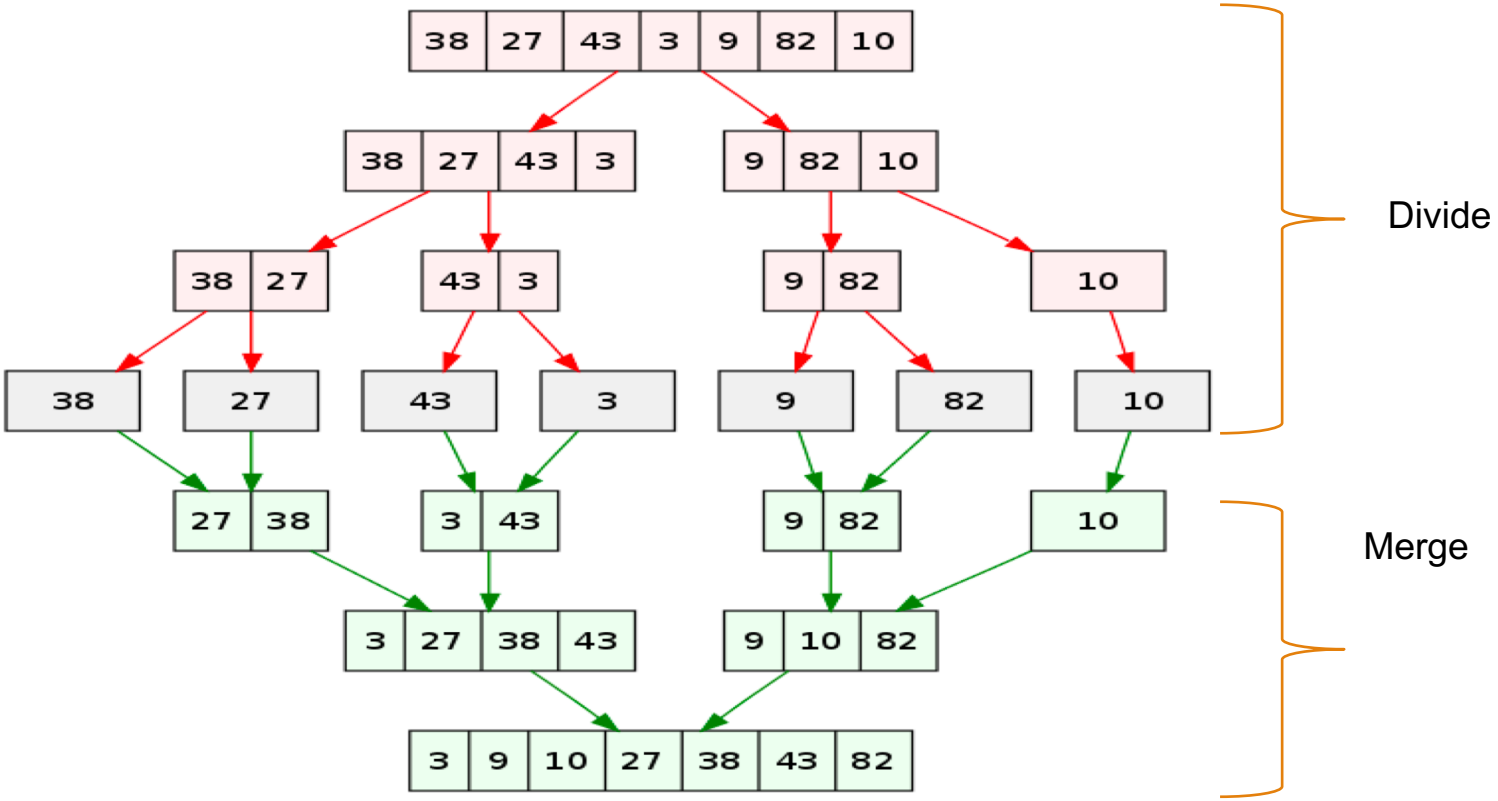
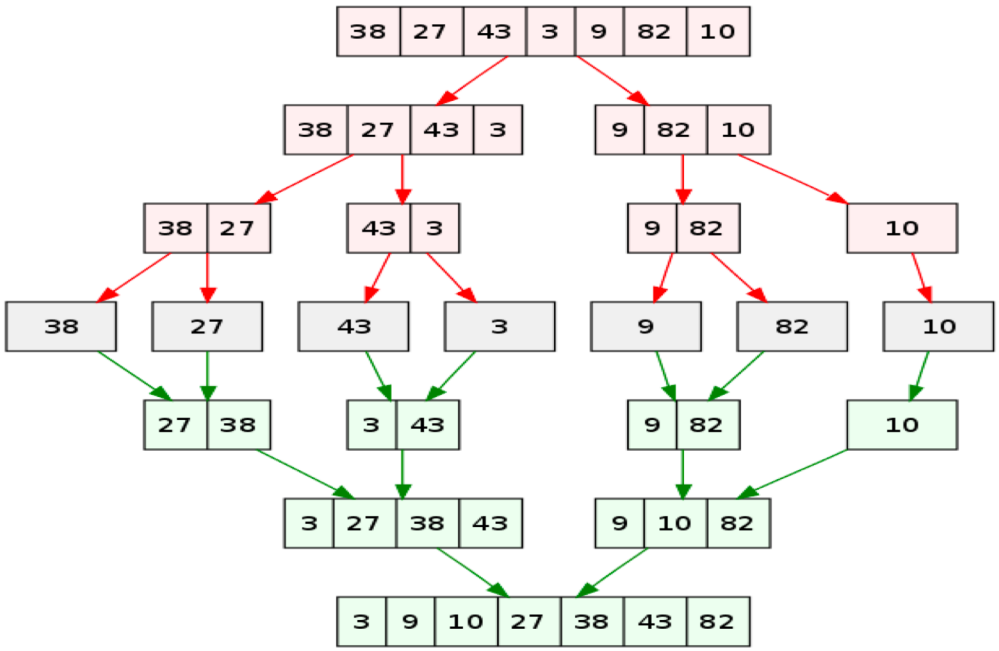


Image source - author: **VineetKumar, Source: wikimedia common**

Merge Sort




```
merge-sort (A, p, s)
     if p < s then
          q= p + (s-p)/2
          merge-sort(A,p,q)
          merge-sort(A,q+1,s)
          merge(A,p,q,s)
```

Image source - author: **VineetKumar, Source: wikimedia common**

Merge Sort: Complexity

```
merge-sort (A, p, s)
     if p < s then
           q= p + (s-p)/2
           merge-sort(A,p,q)
           merge-sort(A,q+1,s)
           merge(A,p,q,s)
```

The complexity of mergesort can be expressed by the following *recurrence relation*:

$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$$

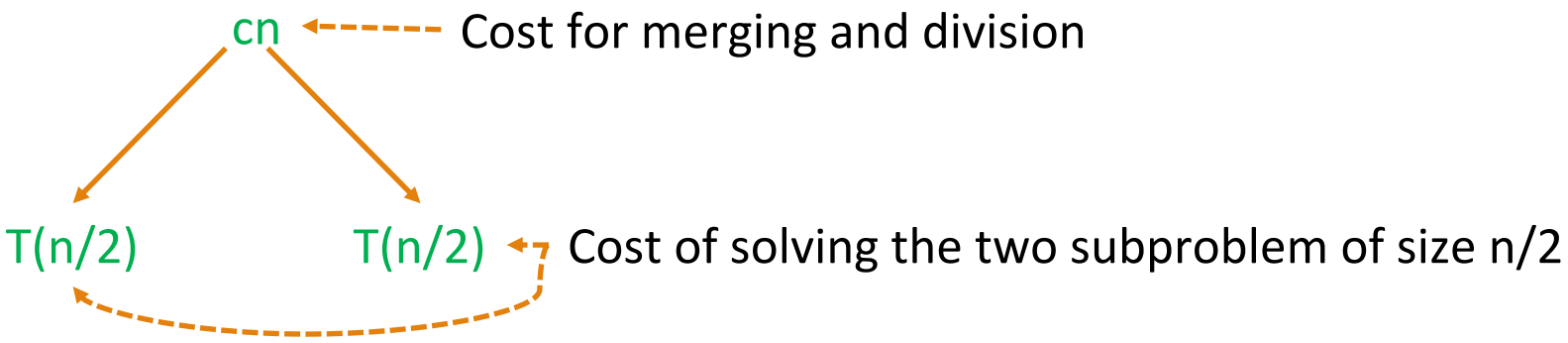
Where, c is a constant

Merge Sort: Complexity

Solving the *recurrence relation.* :

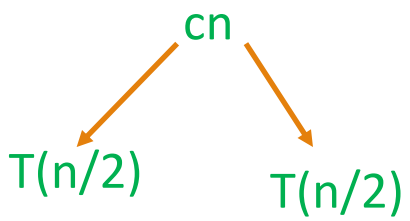
$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$$

Recursion tree for the merge sort



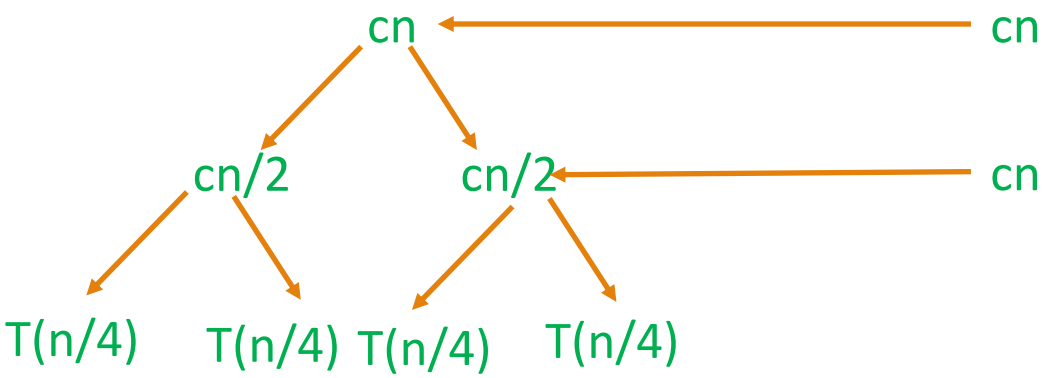
Merge Sort: Complexity

Solving the *recurrence relation.* :



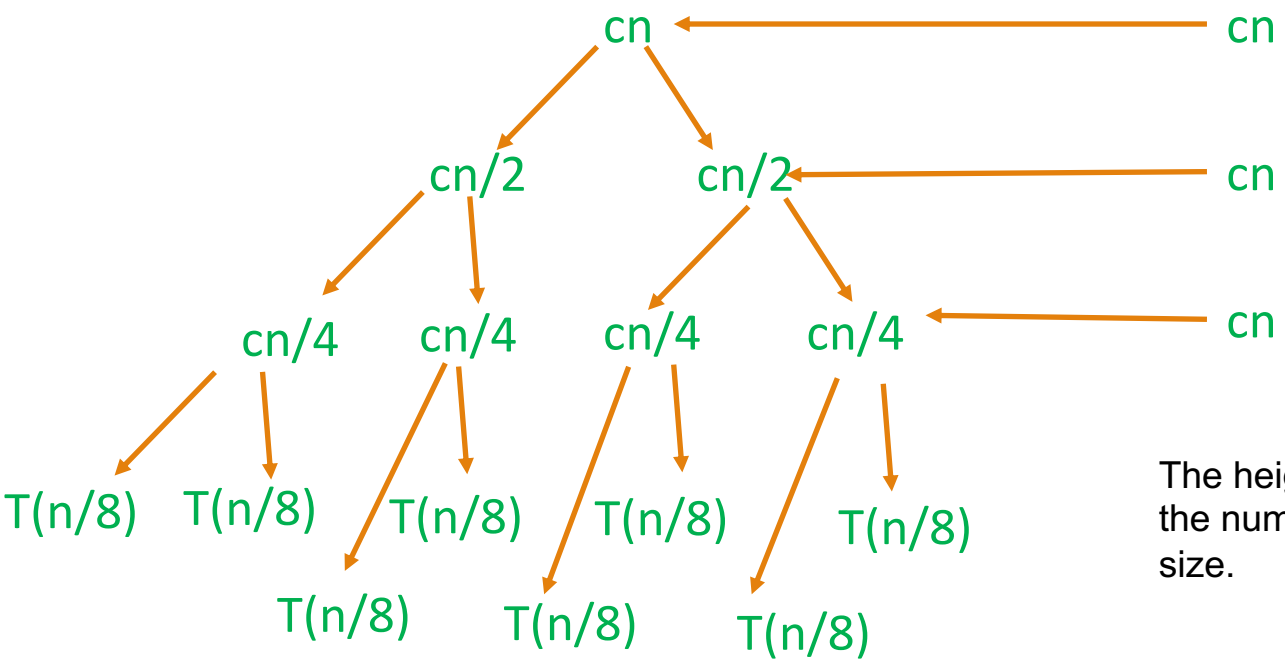
Merge Sort: Complexity

Solving the *recurrence relation.* :



Merge Sort: Complexity

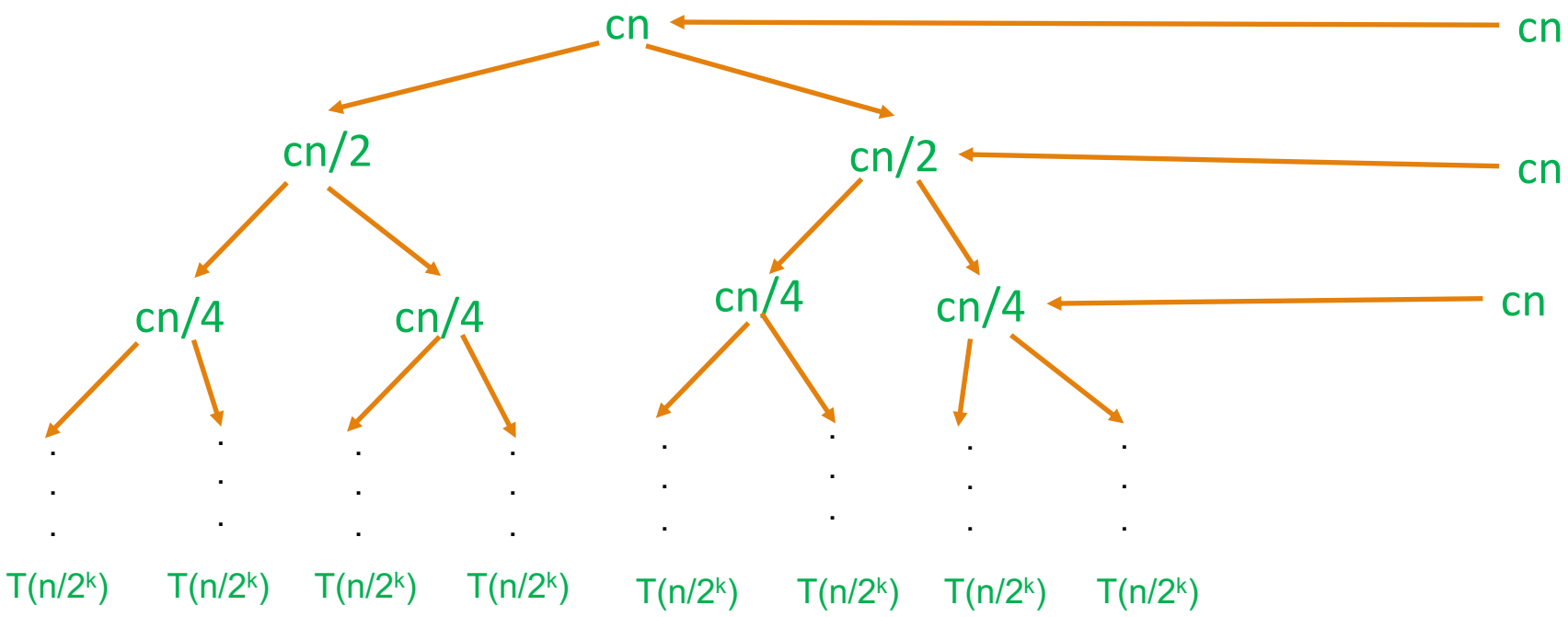
Solving the *recurrence relation.* :



The height of this tree depends on the number of subproblems of distinct size.

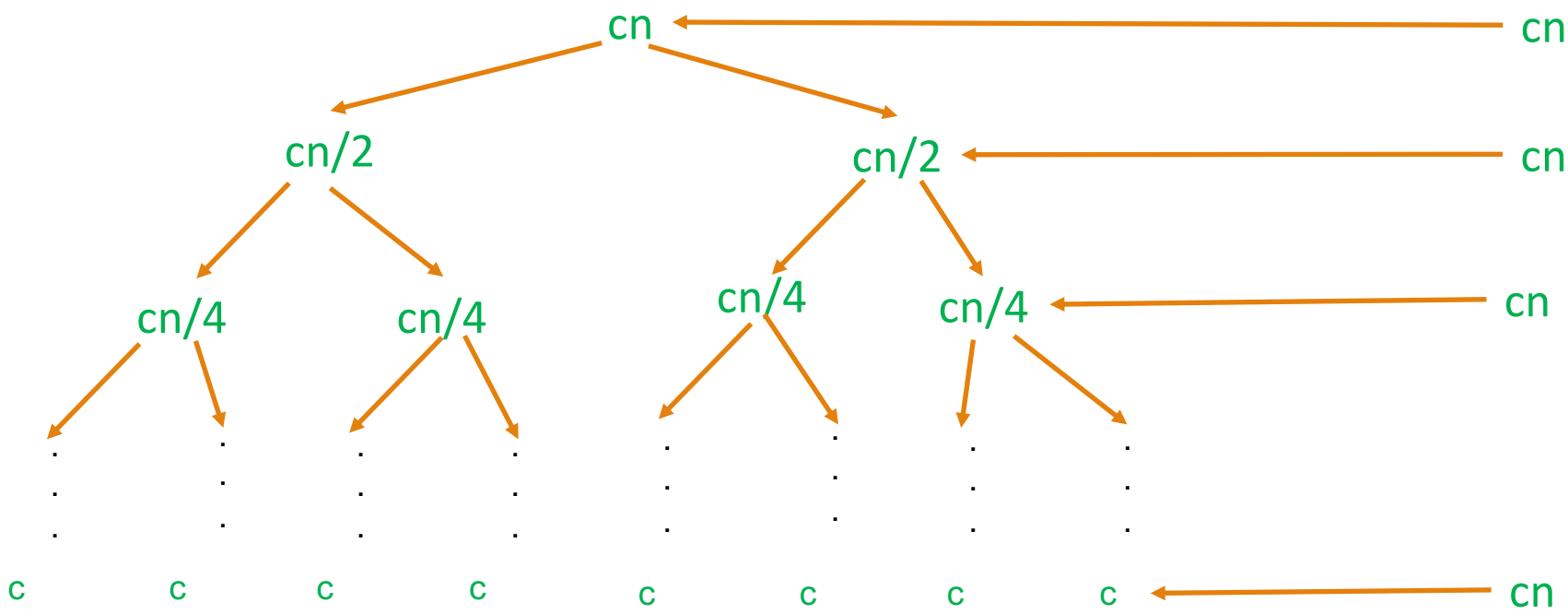
Merge Sort: Complexity

Solving the *recurrence relation.* :



Merge Sort: Complexity

Solving the *recurrence relation.* :



n/ 2^k = 1 imnplies k=$\log_2 n$. So, T(n) = cn $\log_2 n$ and complexity of mergesort is O(n$\log_2 n$)

Quick Sort

Quick sort facts:

- Quicksort algorithm has a worst-case running time of $\boldsymbol{\Theta}$(n2) and average expected running time is $\boldsymbol{\Theta}$(nlogn)
- It's performs very well in practice as the constant factors hidden inside $\boldsymbol{\Theta}$(nlogn) are very small
- It also has the advantage of sorting in place unlike mergesort
- Quicksort also applies the divide and conquer approach

Quick Sort

The three-step divide-and-conquer process of quick sort for sorting the array `A[p..r]`

- **Divide:** *Partition* (rearrange) the array `A[p..r]` into two (possibly empty) subarrays `A[p..q –1]` and `A[q+1..r]` with the following property:
  - each element of `A[p..q–1]` is less than or equal to `A[q]`
  - each element of `A[q+1..r]` is greater than or equal to `A[q]`
  - compute the index `q` as part of this partitioning procedure.

- **Conquer:** Sort the two subarrays `A[p..q–1]` and `A[q+1..r]` by recursive calls to quicksort.

- **Combine:** Because the subarrays are already sorted, the entire array A[p..r] is now sorted. So, nothing to be done here.

Quick Sort

Algorithm:

```
quicksort(A,p,r)
     if p<r
          q=partition(A,p,r)
          quicksort(A,p,q-1)
          quicksort(A,q+1,r)
```

- To sort array A, we call quicksort(A,0,n-1) where n is the size of the array A.
- The key to the algorithm is the *partition procedure*

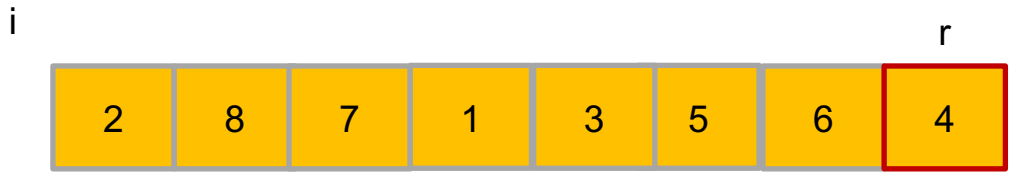
Partitoning in Quick Sort

partition(A,p,r)

- `x =A[r]`
  We call `x` a *pivot* element

Partitoning in Quick Sort

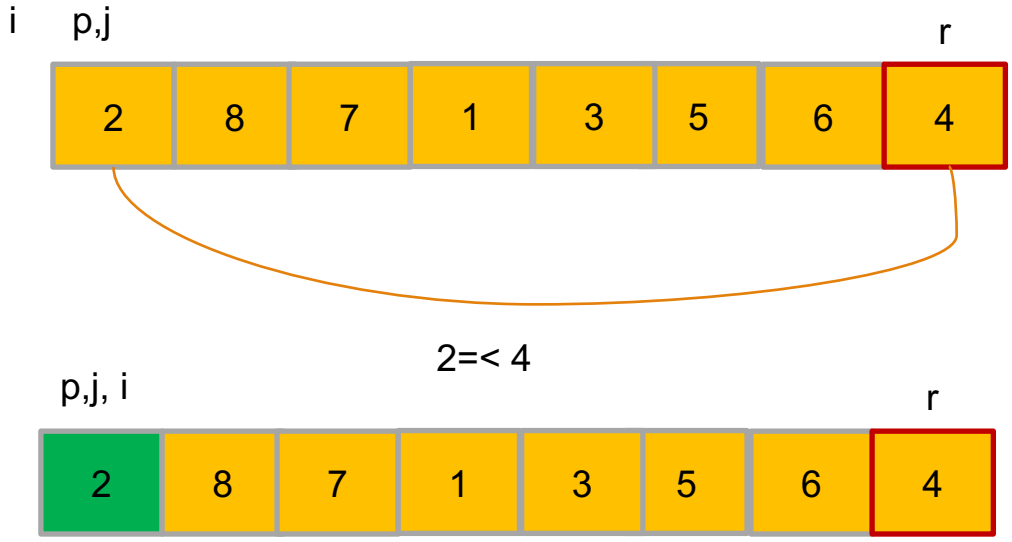
partition(A,p,r)



- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
  At the end of partition, x will be placed at i+1 index
- A[p..i] subarray will contain elements that are smaller than or equal to x
- A[i+2..r] will contain elements that are greater than or equal to x
- A[i+1]=x and i+1 is the right index for x.

Partitoning in Quick Sort

partition(A,p,r)

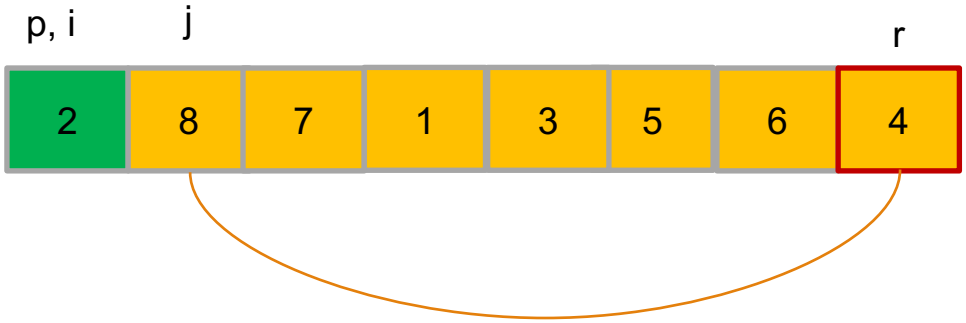


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

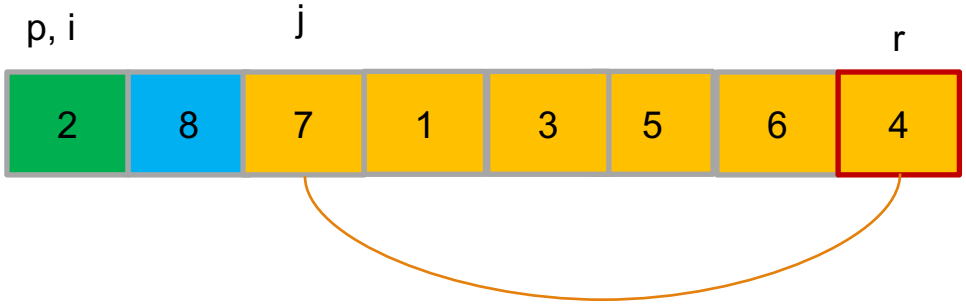


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
       i=i+1
       swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

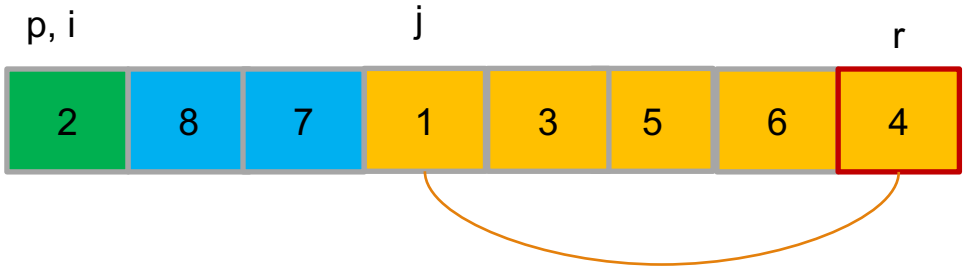


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

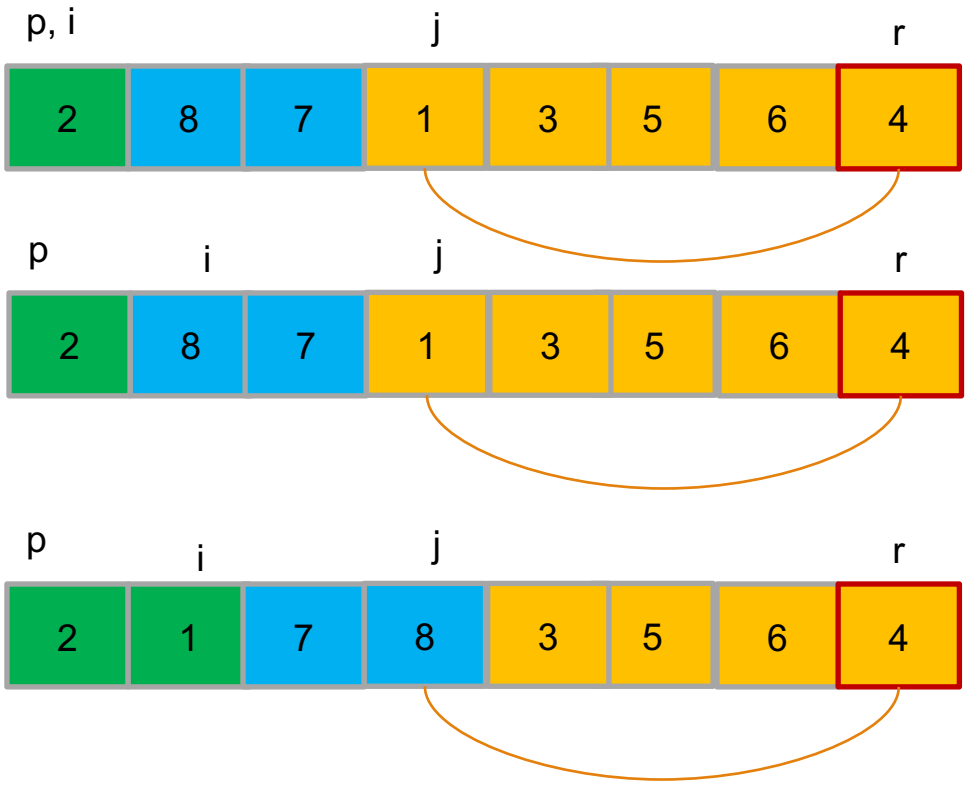


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

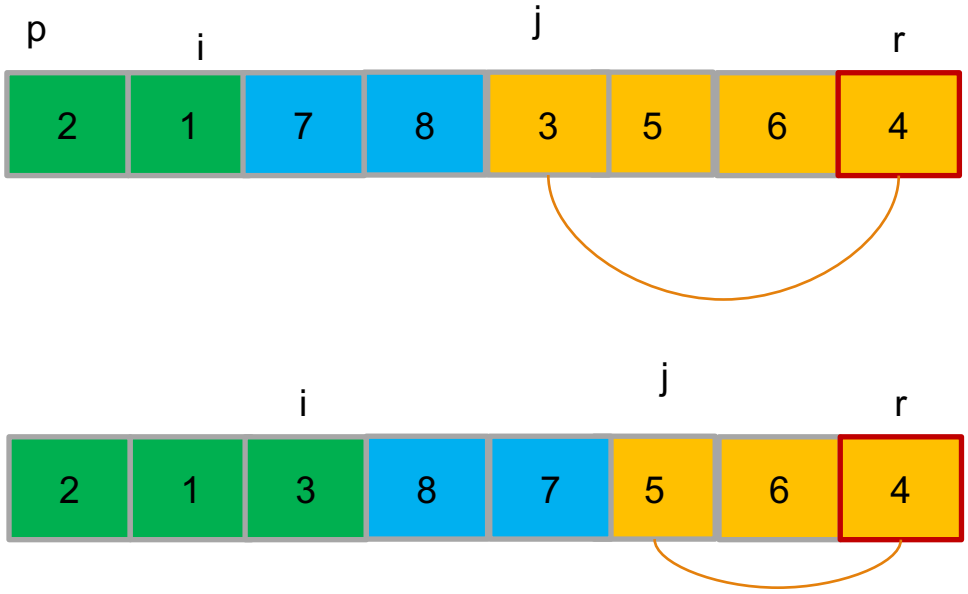


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
       i=i+1
       swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

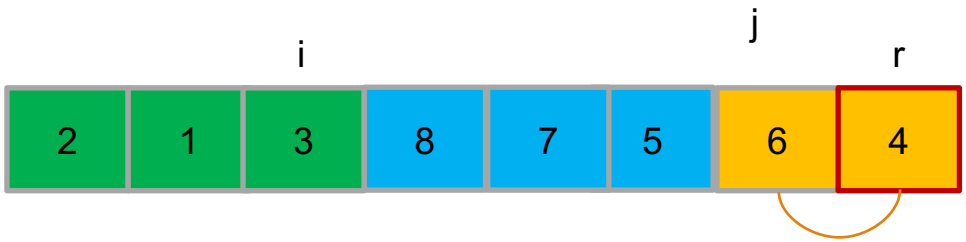


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

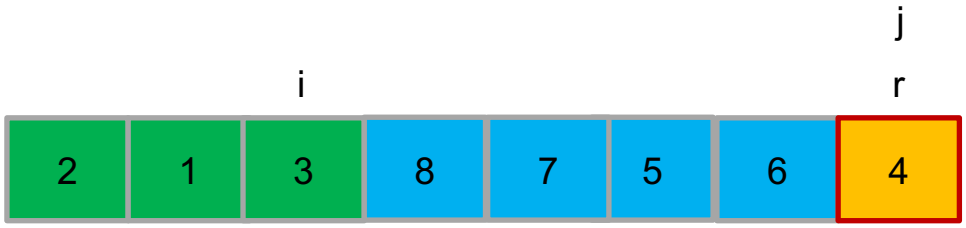


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
   if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)

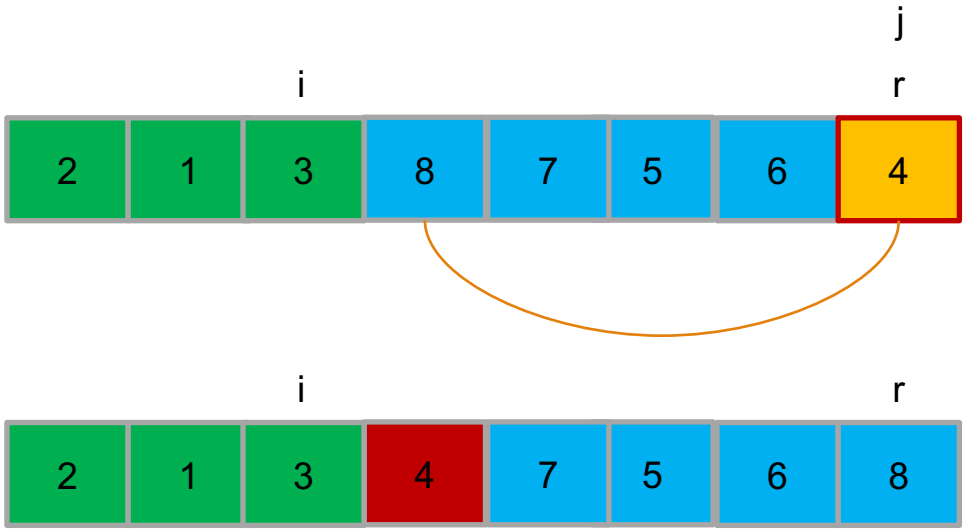


- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
  if A[j]=< x
     i=i+1
     swap(A[i],A[j])
```

Partitoning in Quick Sort

partition(A,p,r)



- `x =A[r]`
  We call `x` a *pivot* element
- `i = p-1`
- We rearrange array elements as follows:

```
for j= p to r-1
  if A[j]=< x
      i=i+1
      swap(A[i],A[j])
```

- `swap(A[i+1],A[r])`
- `Return i+1`

Performance of Quick sort

- The running time of quicksort depends on whether the partitioning is *balanced* or *unbalanced*
- which in turn depends on which elements are used for partitioning.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge.
- If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

Performance of Quick sort

Worst case partitioning

So, after the partition, one subarray has element 0 and the other subarray has element n-1

So, we can use the following recurrence relation:

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Partitioning cost

Performance of Quick sort

Best case partitioning

After the partition, the size of each subarray is no more than n/2

So, we can use the following recurrence relation:

T(n) = 2T(n/2) + Θ(n) = Θ(nlogn)

Balanced Partioning

The average-case running time of quicksort is much closer to the best case than to the worst case.

(see Introduction to Algorithms, Coreman et al, chapter 7)

Selection Sort

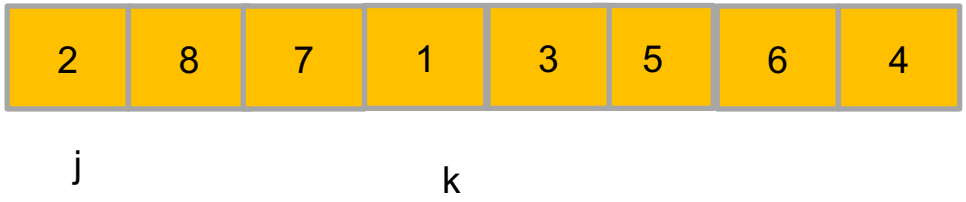
- Selection sort is a simple sorting method that has a quadratic running time complexity (i.e. $O(n^2)$)
- It is thus inefficient to be used on large lists and performs worse than insertion sort algorithm
- It has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Selection Sort

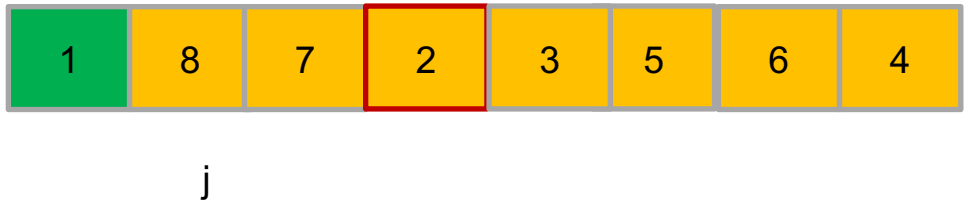
Algorithm `selectionsort(A,i,n):`

```
j=i
While elements exists in unsorted array (j =< n)
        k=findIndexOfMinValue(A,j,n)
        swap(A[j], A[k])
        j = j+1
```

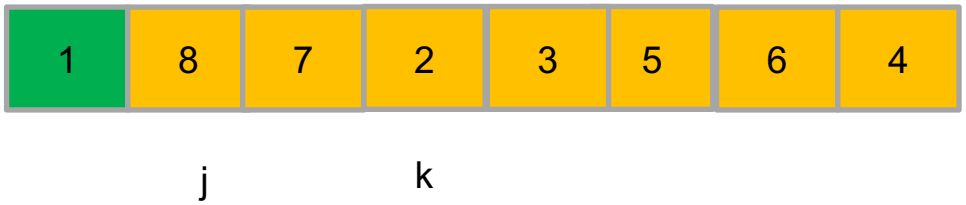
Selection Sort



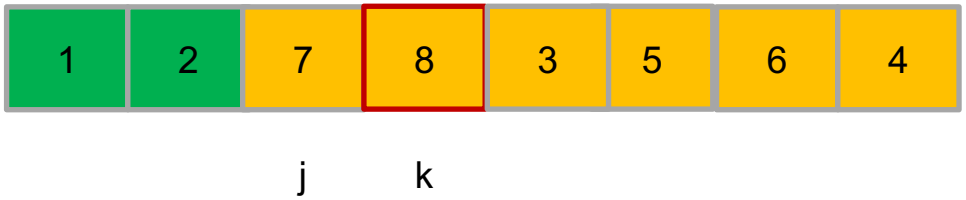
- j=0
- k=`findIndexOfMinValue(A,j,n)`
- Swap (A[k],A[j])
- j++;



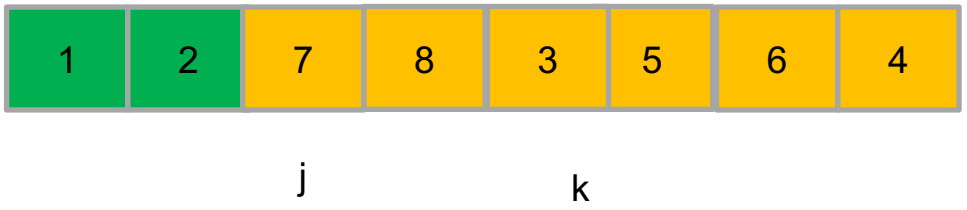
Selection Sort



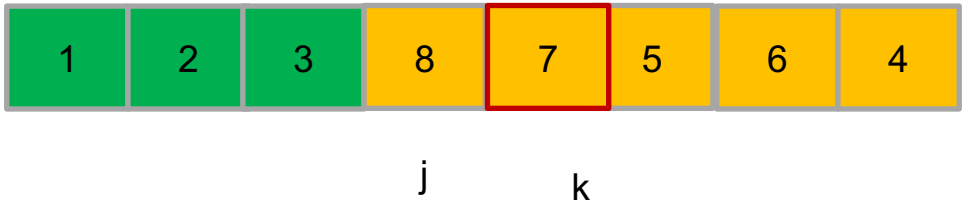
- j=0
- k=`findIndexOfMinValue(A,j,n)`
- Swap (A[k],A[j])
- j++;



Selection Sort

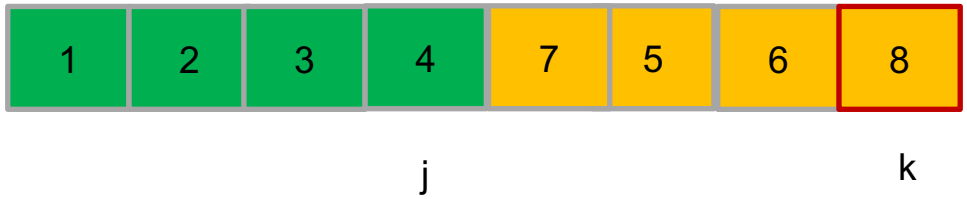


- j=0
- k=`findIndexOfMinValue(A,j,n)`
- Swap (A[k],A[j])
- j++;

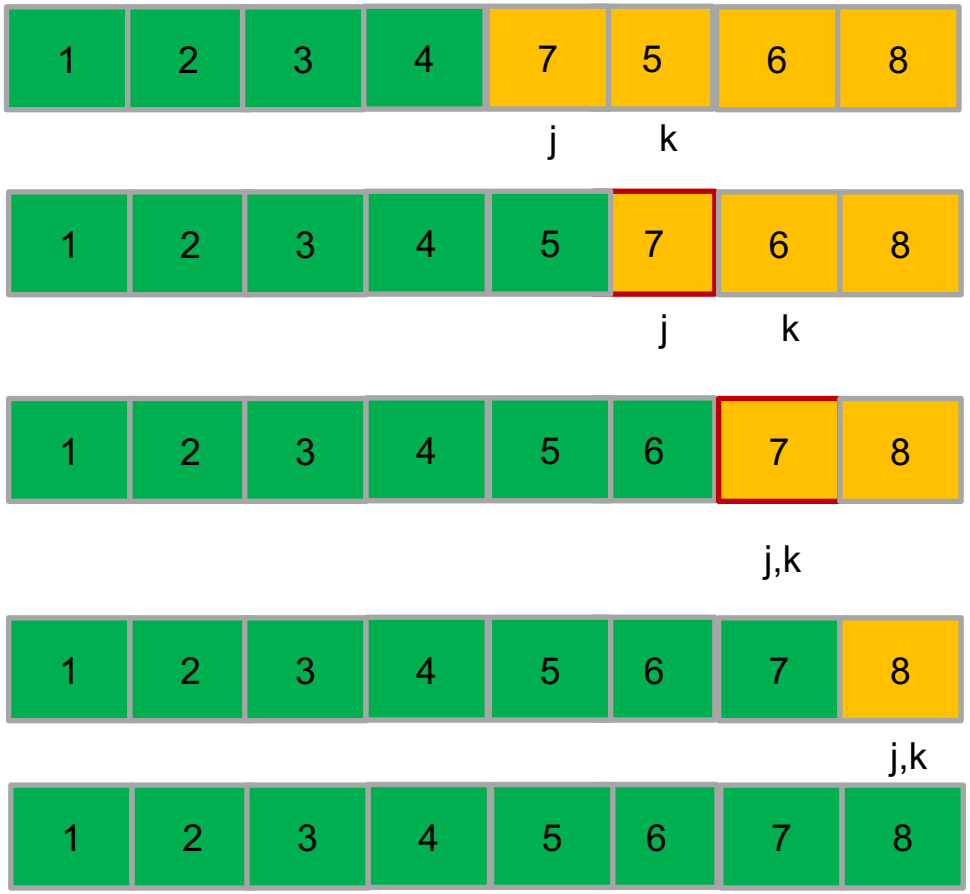


Selection Sort

- j=0
- k=`findIndexOfMinValue(A,j,n)`
- Swap (A[k],A[j])
- j++;



Selection Sort



- j=0
- k=`findIndexOfMinValue(A,j,n)`
- Swap (A[k],A[j])
- j++;

Selection Sort

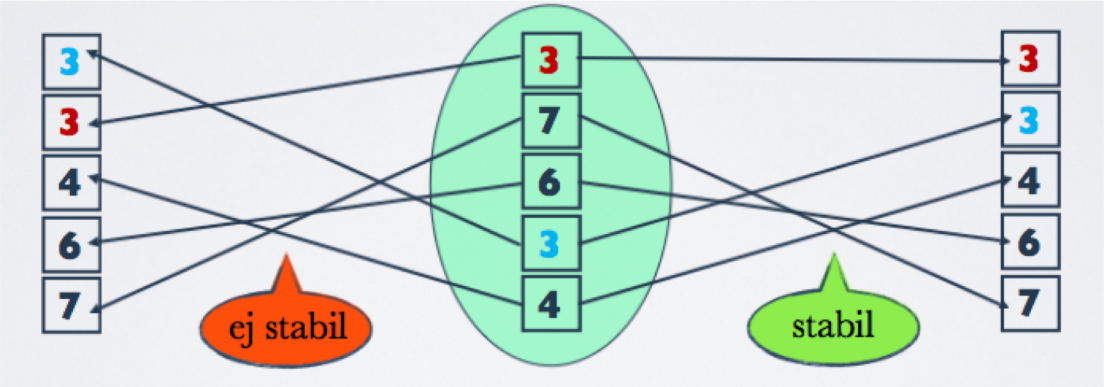
Complexity of Selection Sort

- In Pass 1, `findIndexOfMinValue` requires scanning all n elements.
  thus, n–1 comparisons are required in the first pass.
- In Pass 2, `findIndexOfMinValue` requires scanning all n-1 elements.
  thus, n–2 comparisons are required in the second pass
- and so on
- We have a total of n steps
- Thus, the number of comparison (and possible swaps) are
  $(n - 1) + (n - 2) + \ldots + 2 + 1 + 0$
  $= n(n - 1) / 2 = O(n^2)$ comparisons

- Best and worst case time complexity is $O(n^2)$

More Properties

Stable - *A stable sorting algorithm* is the one that sorts the identical elements in their same order as they appears in the input, whilst unstable sorting *may not* satisfy the case.



- Memory In-Place - Sorting occurs without a copy of the array. (you do not have to use any extra memory to perform the sorting).