COMP2421—DATA STRUCTURES AND ALGORITHMS

Sorting Algorithms

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

جامعة بيرزيت
BIRZEIT UNIVERSITY

INSERTION SORT

Insertion Sort

- One of the simplest sorting algorithms. Consists of n-1 passes over n items.
- For pass p=1 through n-1, it ensures that element in position 0 to p are in sorted order.
- Each pass has k comparisons, where k is pass number. So that 1^{st} pass 1 comparison, the 2^{nd} pass 2 comparisons, …, to (k-1).

Insertion Sort

```
void InsertionSort( int arr[], int n) {
    int i, key, j;
    for( i=1; i<n; i++){
        key = arr[i];
        j = i-1;
        while( j>=0 && arr[j] > key){ //shift elements
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Insertion Sort

- Total number of comparisons is
  $F(n) = 1 + 2 + 3 + \ldots + (n-2) + (n-1)$
  $= n * (n-1) / 2$
  $= O(n^2)$
- Worst case: elements are not sorted → $O(n^2)$
- Average case: $O(n^2)$
- Best case: elements are sorted $O(n)$
  - Because the inner loop won't enter
- Other $O(n^2)$ sorting algorithms include Bubble sort and Selection Sort.

SELECTION SORT

Selection Sort

- Sorts an array by repeatedly finding the minimum element (ascending in this case) from unsorted part & putting it at the beginning.

Selection Sort

```
void SelectionSort(int arr[], int n)
{
    int i, j, temp;

    for (i = 0; i < n-1; i++)
    {
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[i]){
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
    }
}
```

Selection Sort

- Notice that the time complexity is O(n^2) always.
- Good for small arrays (small data size).
- Inefficient for large data.
- Performs all comparisons on sorted data.

RADIX SORT

Radix Sort

- Linear non-comparative sorting algorithm
- O(k.n) time complexity
- Usage
  - Very fast
  - Easy to understand and implement
- Not to use
  - If you are not sure about the data (e.g., if all integers, then ok. If there might be some float of character values, then don’t use it).
  - Requires additional space.

MERGE SORT

Merge Sort

- Average time complexity O(n log n).
- Divide and conquer technique.
- Recursively sort each half of the array.
- Merge 2-halves.
- Code (the merge routine is too large to fit in slides!): https://www.geeksforgeeks.org/merge-sort/

Merge Sort

```
void MergeSort(int arr[], int p, int q)
{
    if (p < q)
    {
        /* Same as (p+q)/2, but avoids overflow for
           large p and h */
        int m = p+(q-p)/2;

        // Sort first and second halves
        mergeSort(arr, p, m);
        mergeSort(arr, m+1, q);

        merge(arr, p, m, q);
    }
}
```


Merge Sort

- Suitable for very large lists.
- Fast recursive algorithm.
- Useful for both internal and external sort.

SHELL SORT

Shell Sort

- It works by comparing elements that are distant.
- The comparison of elements decreases as the algorithm runs until the last phase, in which adjacent elements are compared.
- Motivation: since insertion sort runs fast on nearly sorted data, then do several passes of insertion sort on different subsequence of elements.

Shell Sort - Example

- Step 1: set up increment gap variable.
- Step 2: mark each element that comes in inc. gap. E.g., if we have 10 elements, set up inc. gap to 3

89	46	99	**12**	33	14	**69**	41	33	**28**

- Step 3: sort marked elements such that the smallest goes to the 1st place.

12	46	99	**28**	33	14	**69**	41	33	**89**

Shell Sort - Example

- Step 4: reduce inc. gap by 1.
- Step 5: repeat steps 2, 3, 4 till all elements area sorted.

12	46	**99**	28	**33**	14	**69**	41	**33**	89

12	46	**33**	28	**33**	14	**69**	41	**99**	89

12	**46**	**33**	**28**	**33**	**14**	**69**	**41**	**99**	**89**

12	14	28	33	33	41	46	69	89	99

Shell Sort

```
void shellSort(int arr[], int m) {
    int inc, j, k, temp;
    for(inc = n/2; inc>0; inc /= 2){
        for(j=inc; j<num; j++){
            for(k=j-inc; k>=0; k-=inc){
                if(arr[k+inc] >= arr[k])
                    break;
                else{
                    temp = arr[k];
                    arr[k] = arr[k + inc];
                    arr[k + inc] = temp;
                }
            }
        }
    }
}
```


Shell Sort

- The average complexity of Shell sort depends on the gap.
  - Different gap sizes change the complexity of the sort.
  - E.g., using Shell's gap $(n/2^k)=O(n^2)$, Hibbard's method $(2^k-1)=O(n^{3/2})$, $k>=0$ and $< n$.
- Average & best case: $O(n \log n)$.
- Worst case: $O(n^2)$.
- Not stable.
- Efficient for large lists.
- Requires relatively small amount of memory as it is extension of insertion sort.
- As it has more constraints, it is not very stable sort algorithm.

QUICK SORT

Quick Sort

- One of the fastest sorting algorithms on average.
- Divide & conquer technique.
- Consists of the following steps:
  - If the number of elements in the array is 0 or 1, return.
  - Pick an element (pivot) P
  - Re-arrange the elements into 3-sub-blocks:
    - Those less than or equal to P (left-block S1)
    - P (the only element in the middle)
    - Those greater than or equal to P (right-block S2)
  - Return {quicksort(S1), P, quicksort(S2)}

Quick Sort

- Quick sort does not perform well on small arrays as insertion sort for example.

- Selecting Pivot:
  - Randomly
  - Element at position n/2
  - Take the median of (first, n/2, last)

Quick Sort - Example

4	4	8	0	8	9	7	3	7	6

- Step 1: pick the pivot (mid-element)

Quick Sort - Example

4	4	8	0	8	9	7	3	7	6

- Step 1: pick the pivot (mid-element)

4	4	8	0	**8**	9	7	3	7	6

Quick Sort - Example

4	4	8	0	8	9	7	3	7	6

- Step 1: pick the pivot (mid-element)

4	4	8	0	**8**	9	7	3	7	6

- Step 2: Swap pivot with last element.

Quick Sort - Example

4	4	8	0	8	9	7	3	7	6

- Step 1: pick the pivot (mid-element)

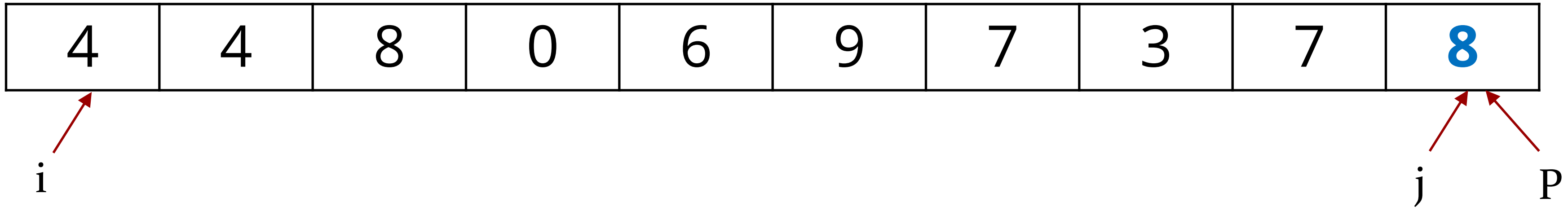
4	4	8	0	**8**	9	7	3	7	6

- Step 2: Swap pivot with last element.

4	4	8	0	6	9	7	3	7	**8**

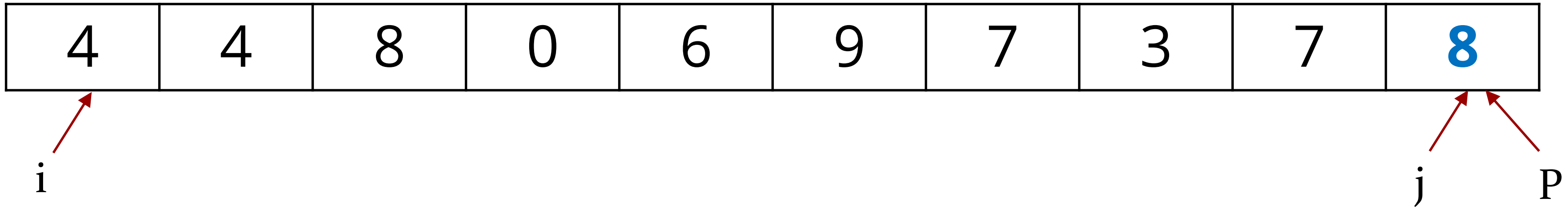
i j P

Quick Sort - Example

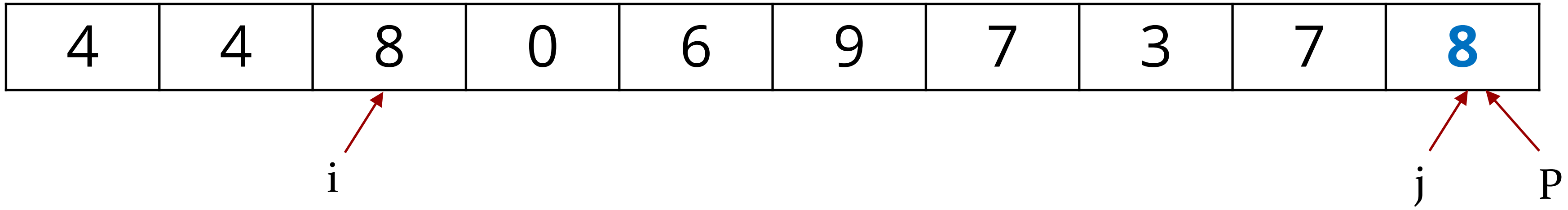


- Increment i to reach the first item greater than or equal to the pivot

Quick Sort - Example



- Increment i to reach the first item greater than or equal to the pivot



- Decrement j to reach the element that is less than or equal to the pivot.

Quick Sort - Example

4	4	8	0	6	9	7	3	7	**8**
		i						j	P

- Swap i & j

4	4	7	0	6	9	7	3	8	**8**
		i						j	P

- Increment i to reach the next greater item.

Quick Sort - Example

4	4	7	0	6	9	7	3	8	**8**
					i			j	P

- Decrement j to reach the next element less than the pivot

4	4	7	0	6	3	7	9	8	**8**
					i		j		P

- Swap i & j

Quick Sort - Example

4	4	7	0	6	3	7	9	8	**8**

i j P

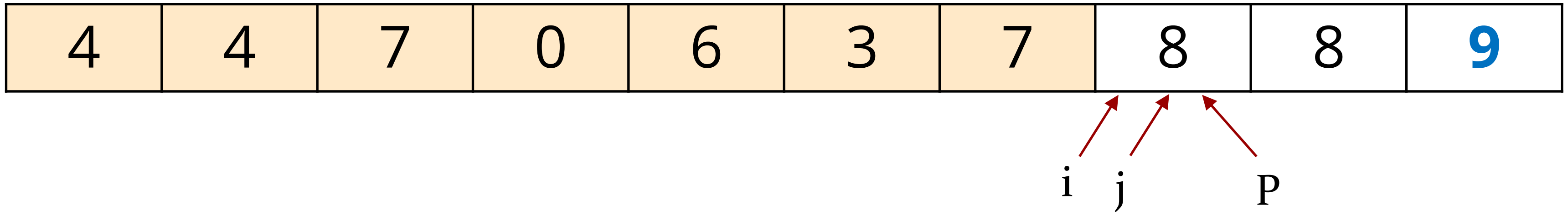
- Increment i to reach the next element greater than or equal to the pivot

4	4	7	0	6	3	7	9	8	**8**

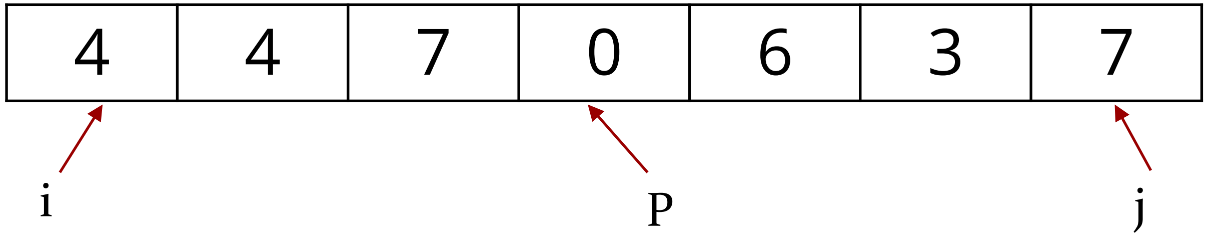
i j P

- i & j have crossed.
- Swap with the pivot.

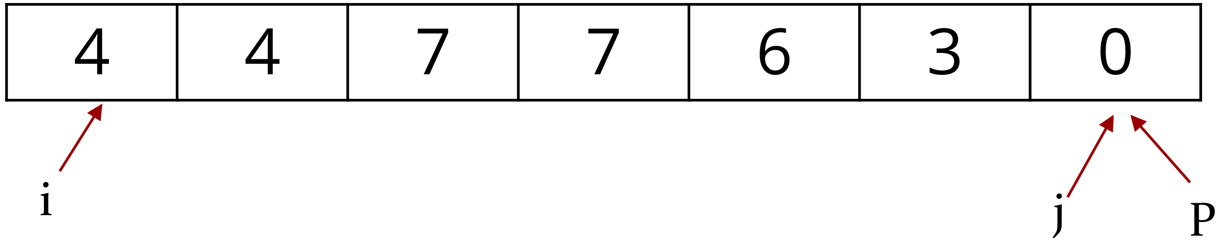
Quick Sort - Example



- Apply recursion on the left sub-list



- Pick pivot and swap with the last element



Quick Sort - Example

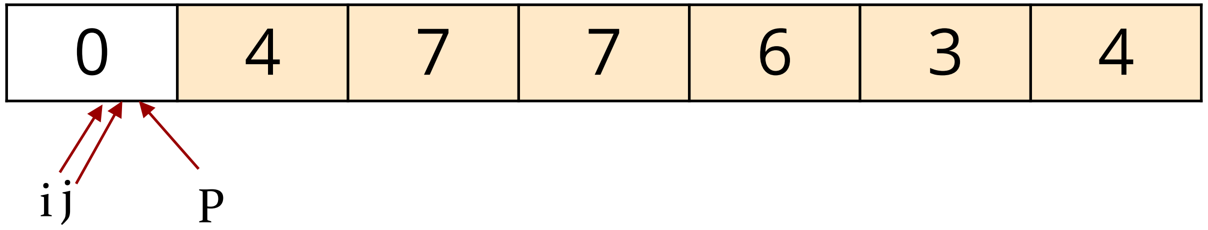
4	4	7	7	6	3	0
i						j P

- Nothing is less than the current pivot. So j is moved all the way to the first item.

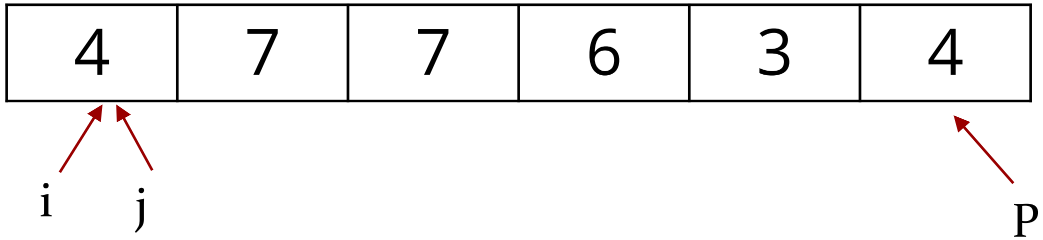
4	4	7	7	6	3	0
i j						P

- Swap with the pivot.

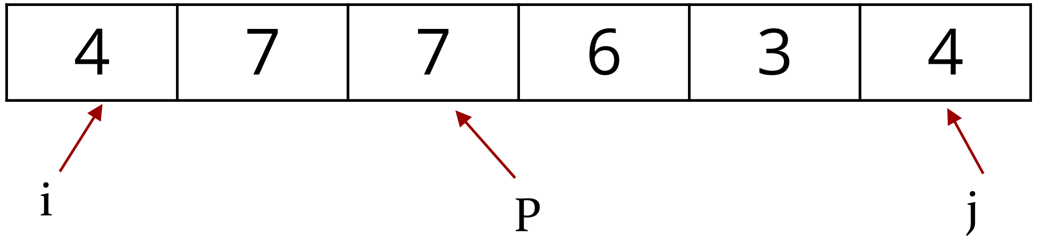
Quick Sort - Example



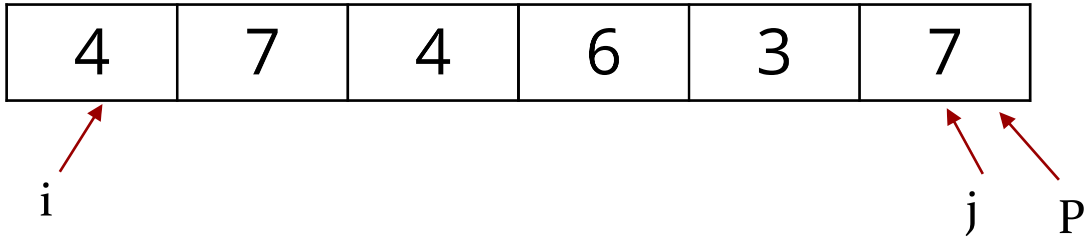
- Recursion on the right sub-list.



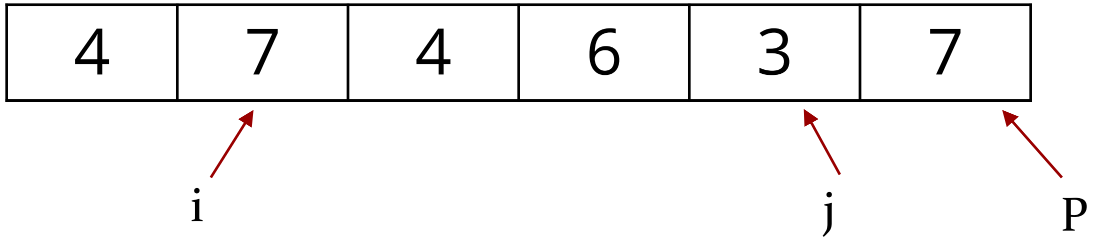
- Select pivot and swap with last.



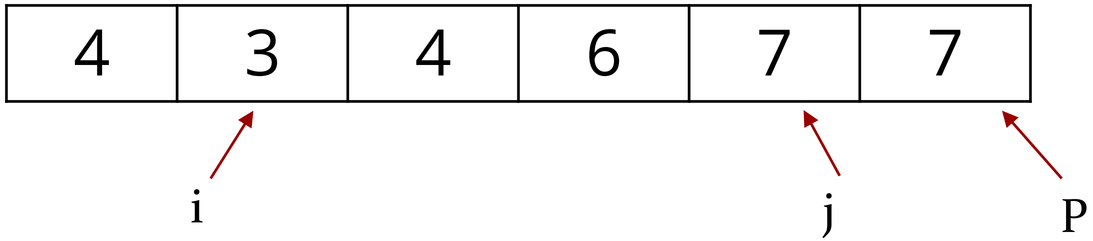
Quick Sort - Example



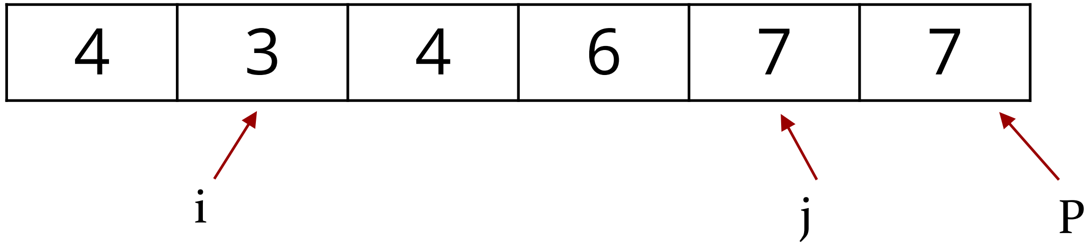
- Increment i & decrement j



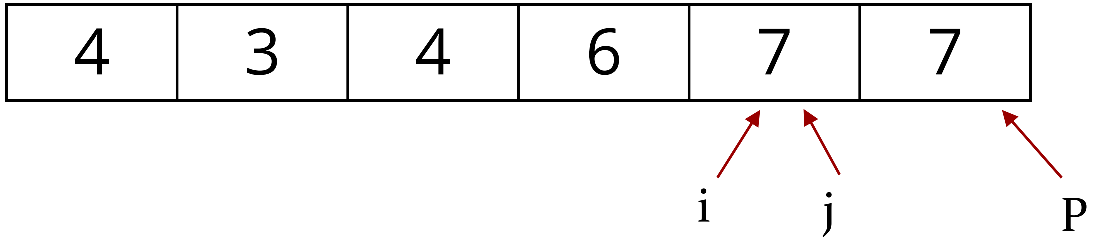
- Swap.



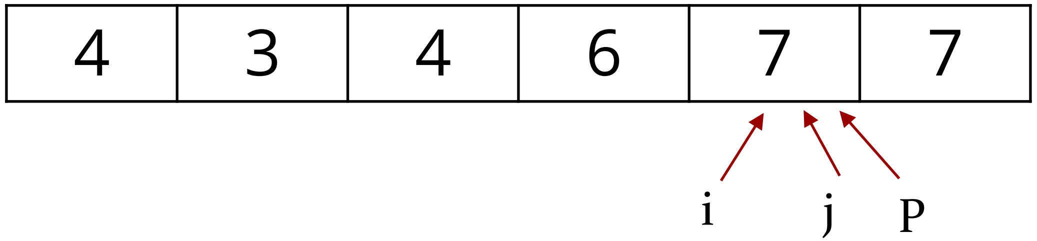
Quick Sort - Example



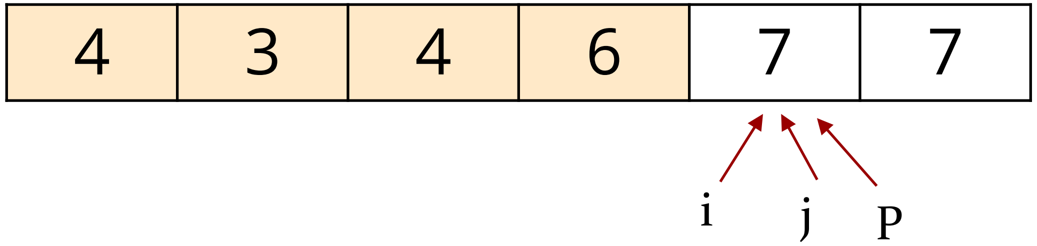
- Increment i & decrement j



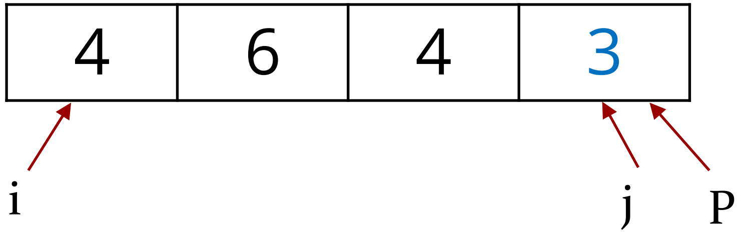
- Swap pivot.



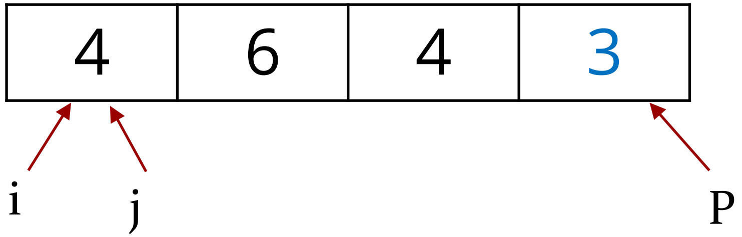
Quick Sort - Example



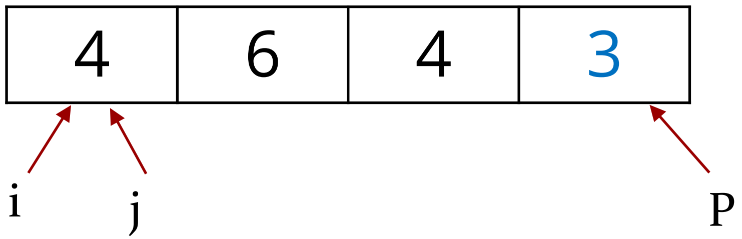
- Recursion. Select pivot and swap with last.



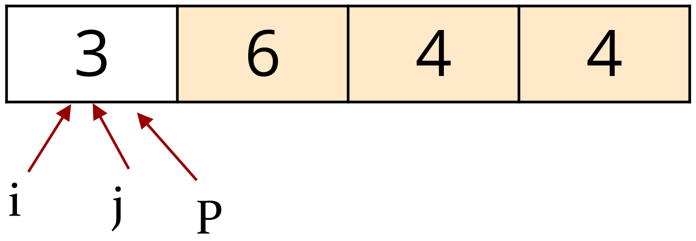
- Increment i & decrement j



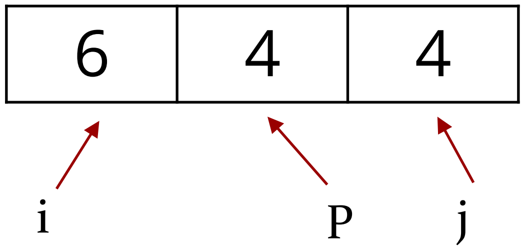
Quick Sort - Example



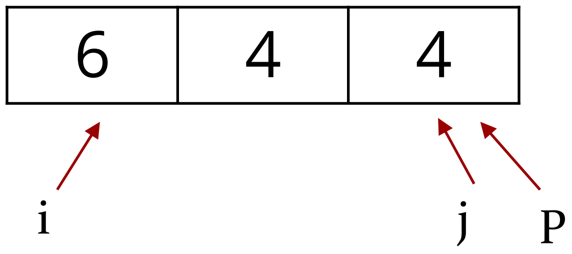
- Swap pivot.



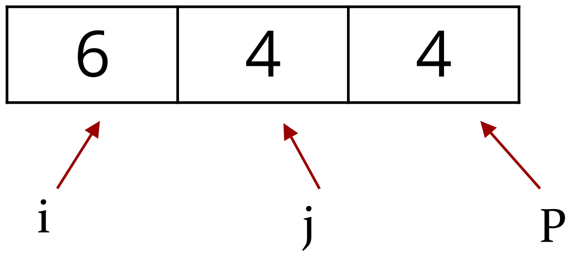
- Recursion. Select pivot and swap.



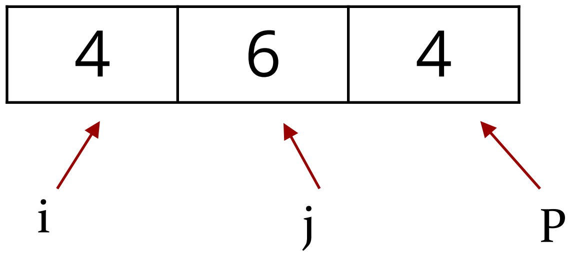
Quick Sort - Example



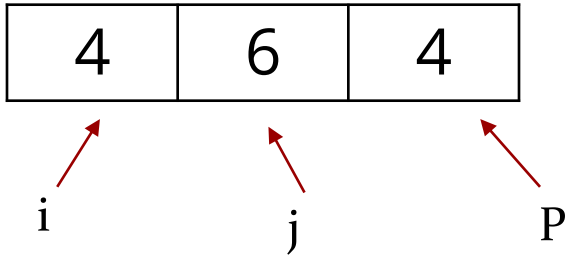
- Decrement j



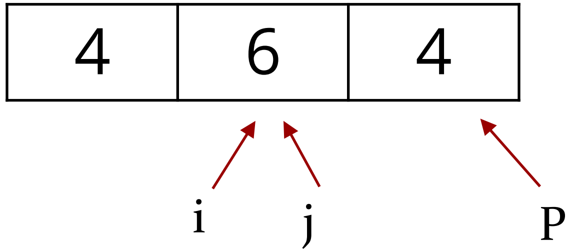
- Swap i & j



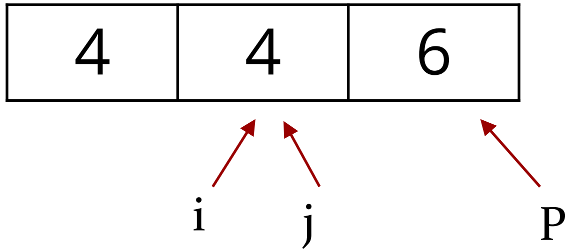
Quick Sort - Example



- Decrement j and increment i

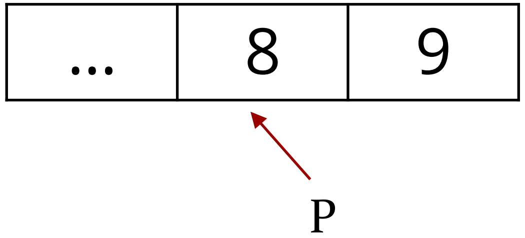


- Swap the pivot

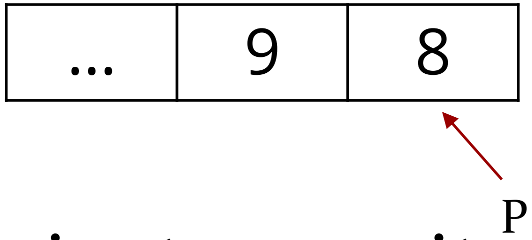


Quick Sort - Example

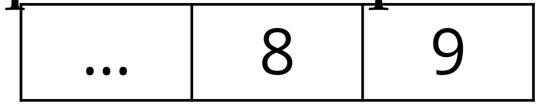
- Sorting 4 & 4
- Then return to the right-sublist



- Swap with the last element



- Least item is the pivot, swap it, and then the array is sorted



Quick Sort - Example

0	3	4	4	6	7	7	8	8	9

- Best-case runtime O(n log n)
- Average-case runtime O(n log n)
- Worst-case runtime $O(n^2)$
- Fast & efficient for large amount of input data.
- No additional memory is required.

Quick Sort - Example

- Worst-case scenario: when the partition sizes are unbalanced. E.g., the pivot always the smallest or largest element in the n-element. Then one partition will contain no elements & the other will contain (n-1) elements.
- So the total partitioning time for all sub-problems of this size is: $cn + c(n-1) + c(n-2) + \ldots + 2c + 0$

  $= c((n+1)(n/2)-1)$

  $= O(\ n^2\ )$

HEAP SORT

Heap Sort

- O(n log n) in all cases.
- Even though Quick sort is O(n^2) in the worst case, it is still a better choice than Heap sort.
  - More difficult to implement (entire data structure and its operations) and requires to build a tree.
- Uses heaps data structure to sort elements in the array.
- Requires extra array so that memory requirements are doubled.

Heap Sort

- Idea: turn the array of data into heap. DeleteMin and insert it into a sorted array until the heap is empty.
- Each DeleteMin will take O(log n) time, and we will delete all N elements, so O(n log n).

EXTERNAL SORT

External Sort

- Is used to sort data that resides on different storage device.
- The basic external sorting algorithm uses the merge routine from Merge Sort.
- Suppose we have four tapes: T_{a1}, T_{a2}, T_{b1}, T_{b2}; which are two input and two output tapes.
- Suppose the data is initially on Ta1. Suppose further that the internal memory can hold (and sort) m records at a time. The first step is to read m records from the input tape, sort them, and write the sorted records alternately to T_{b1} and T_{b2}.

External Sort

- Each of the sorted records is called a *run*.
- Suppose m = 3, then the sort will be

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

External Sort

- Now T_{b1} & T_{b2} contain groups of runs. We take the first run from each tape & merge them, which runs twice as long onto T_{a1}.

T_{a1}					
T_{a2}					
T_{b1}	11 81 94	17 28 99	15		
T_{b2}	12 35 96	41 58 75			

External Sort

- Now T_{a1} & T_{a2} contain groups of runs. We take the first run from each tape & merge them, which runs twice as long onto T_{a1}.

T_{a1}	11	12	35	81	94	96	15	
T_{a2}	17	28	41	58	75	99		
T_{b1}								
T_{b2}								

External Sort

- Now T_{a1} & T_{a2} contain groups of runs. We take the first run from each tape & merge them, which runs twice as long onto T_{b1}

T_{a1}	11	12	35	81	94	96	15	
T_{a2}	17	28	41	58	75	99		
T_{b1}								
T_{b2}								

External Sort

- Now T_{a1} & T_{a2} contain groups of runs. We take the first run from each tape & merge them, which runs twice as long onto $T_{b1.}$

T_{a1}												
T_{a2}												
T_{b1}	11	12	17	28	35	51	58	75	81	94	96	99
T_{b2}	15											

External Sort

- Now T_{b1} & T_{b2} contain groups of runs. We take the first run from each tape & merge them, which runs twice as long onto $T_{a1.}$

T_{a1}	11	12	15	17	28	35	51	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

External Sort

- If we add more tapes it will make the sort faster; instead of 2-way merge it becomes a k-way merge.
- This algorithm will require log(n/m) passes, plus the initial run-constructing pass. For instance, if you have 10M records of 128 bytes each and a 4 megabytes of internal memory, then the first pass will create 320 runs. We will need then 9 more passes to complete the sort. Our example needed log(13/3) = 3 more passes to finish the sort.