CMPT 225

Lectures 10 and 11 – Efficient Sorting Algorithms: Quick Sort

Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - describe the behaviour of and implement simple sorting algorithms:
    - selection sort
    - insertion sort
  - describe the behaviour of and implement more efficient sorting algorithms:
    - quick sort
    - merge sort
  - analyze the best, worst, and average case running time (and space) of these sorting algorithms

Last Lecture

- Quick Review of Searching algorithms
- Simple Sorting
  - Selection sort
  - Insertion sort
  - Analyze their best, worst, and average case running time and space efficiency of these sorting algorithms

Today's menu

- Looking at
  - Describing how quick sort works
  - Analyzing the time efficiency of quick sort
  - Discussing how to improve quick sort's time efficiency
  - Analyzing its space efficiency
  - Discussing how to improve quick sort's space efficiency

Question from last lecture:

1 + 2 + … + n-2 + n-1 = n(n-1)/2 ?

Summation S = $\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ \ldots + n - 3 + n - 2 + n - 1$

We can rewrite S: $S = n - 1 + n\ - 2 + n\ - 3 + \ \ldots + 3 + 2 + 1$

And re-rewrite S: $S = [(n) - 1] + [(n) - 2] + \cdots + [(n) - (n-2)]\ + [(n) - (n-1)]$

$= \sum_{i=1}^{n-1} n - i$ (in which we isolate n in each operand)

We add both versions of S above: $S + S = \sum_{i=1}^{n-1} i\ + \sum_{i=1}^{n-1} n - i$

factor out sigma: $2S = \sum_{i=1}^{n-1}(i + (n - i))$

simplify: $2S = \sum_{i=1}^{n-1}(n)$ (n is a constant)

expand, i.e., add n (n-1) times: $2S = (n) + (n) + (n) + \ldots + (n) + (n)$

$2S = (n)(n\ - 1)$

divide by 2 on both sides: $S = \frac{(n)(n-1)}{2}$ QED (quod erat demonstrandum)

Question from last lecture: How can we tell whether a sorting algorithm is stable or not?

- Let's observe Selection Sort in action with duplicated data

How Quick Sort works

- **Divide and conquer** algorithm
- **Partitioning** method
- **Recursive** in nature
- If array has **n** elements

So, the actual **sorting** is done while **partitioning** elements around a pivot

1. *Pivot*: we select an element to be the pivot
2. *Partitioning*: we partition the array around this pivot by swapping elements such that elements < pivot are in one partition and elements > pivot are in the other

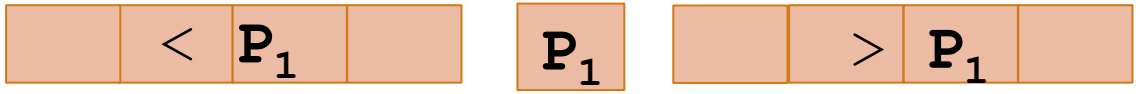
   Repeat partitioning each partition until the array is completely sorted

* What about when elements == pivot?

Quick Sort – Partitioning step (in a nutshell)

- **Problem statement** -> sort this array:

- So we partition this array such that … (P_1 is pivot)

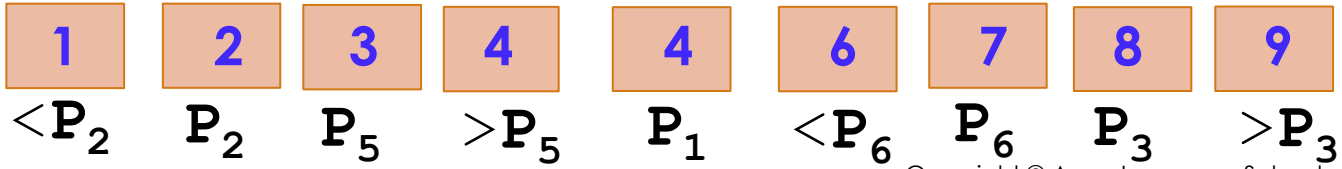


- Then the problem becomes -> sort these 2 smaller arrays:

- So we partition these 2 smaller arrays such that …

- Then the problem becomes -> sort these even smaller arrays:

- So we continue solving our **sort problem** by partitioning previously partitioned partitions into smaller and smaller arrays until each **sort problem** is solved



Many different versions of Quick Sort

- Different ways of selecting the pivot
  - Will be discussed later on in these lecture notes
- Different ways of partitioning the array using the pivot
  - **Lomuto** partition scheme
  - **Hoare** partition scheme

Lomuto partition scheme – thank you, wiki!

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)
```

```
algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Demo - Let's have a look at Quick Sort

- **Lomuto** partition scheme

Hoare partition scheme – thank you, wiki!

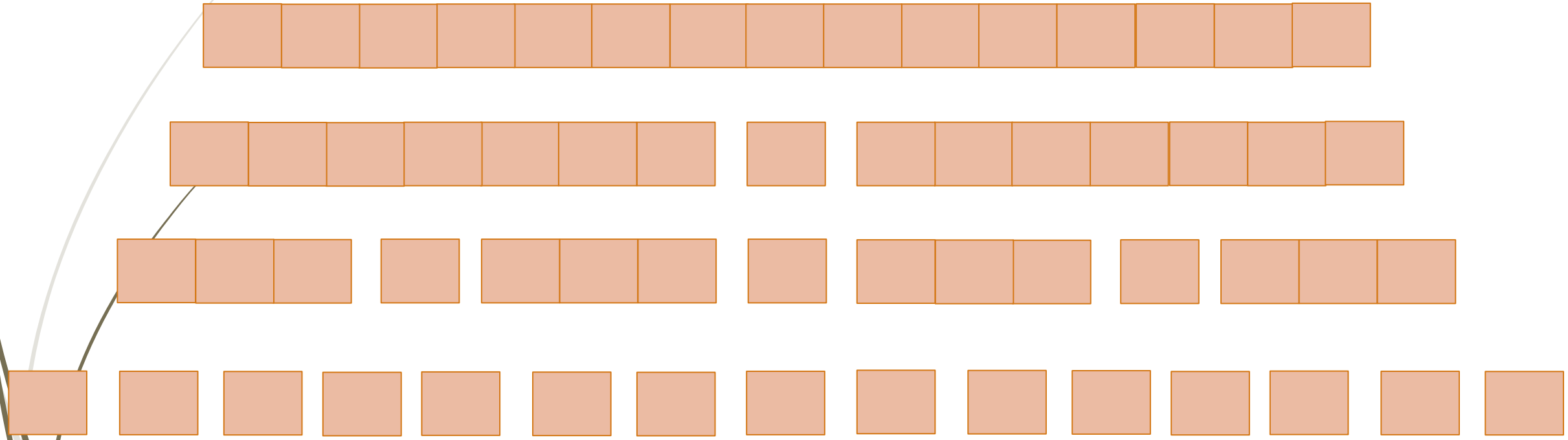
```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)
```

```
algorithm partition(A, lo, hi) is
    pivot := A[(lo + hi) / 2]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot

        do
            j := j - 1
        while A[j] > pivot

        if i >= j then
            return j

        swap A[i] with A[j]
```

Demo - Let's have a look at Quick Sort

- **Hoare** partition scheme

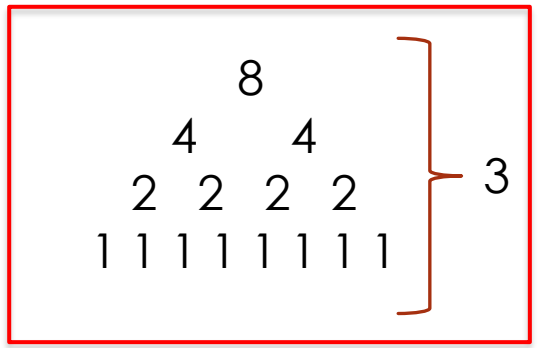
Quick Sort – Pattern of the Best Case Scenario



Time Efficiency Analysis of Quick Sort – Best Case Scenario

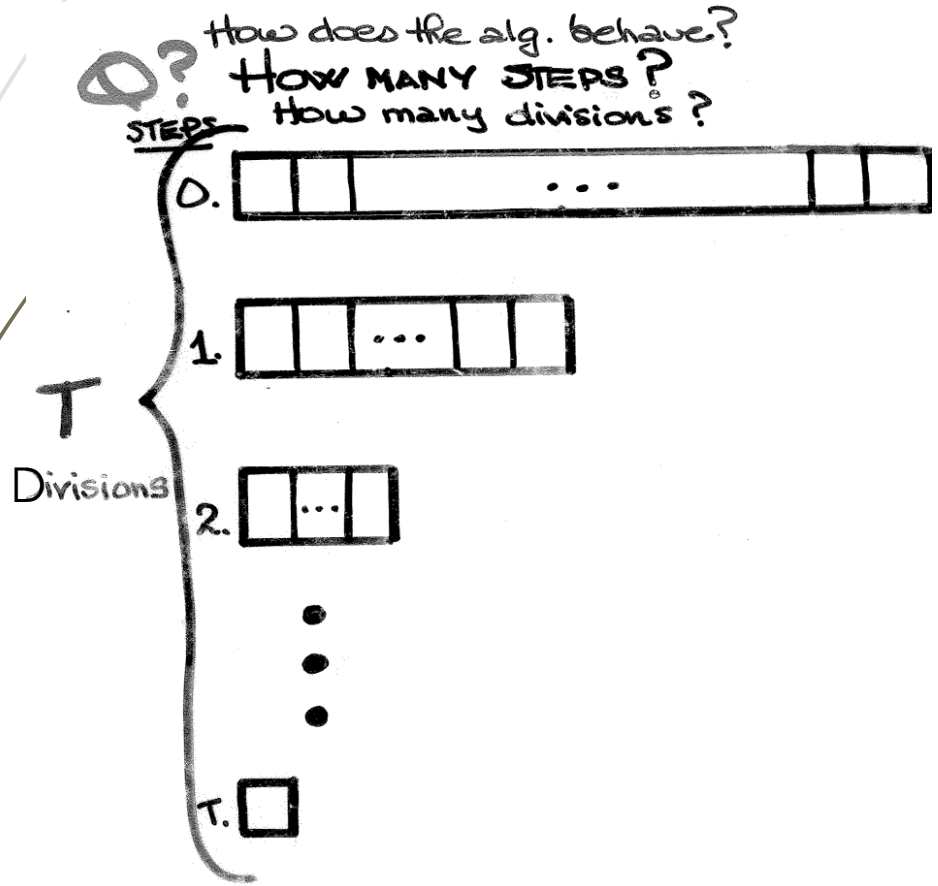
- Each time we partition …
  - We divide the data collection in half
    - Selected pivot is such that its value is (more or less) the median value of data collection
    - Effect of this is that the partition with elements < pivot is about the same size as the partition with elements > pivot
  - At most **n - 1** comparisons are made at each level
- How many times do we partition?
  - We divide the data collection in half until the partitions have size 1
  - How many times does **n** have to be divided in half before the result is 1?
  - Answer: $\log_2$ **n** times
- Quick sort performs **n - 1** * $\lceil \log_2 \mathbf{n} \rceil$ operations in its best case scenario

For example:



Details of time efficiency analysis: how many times do we partition?

Each time we partition, we divide the data collection in half



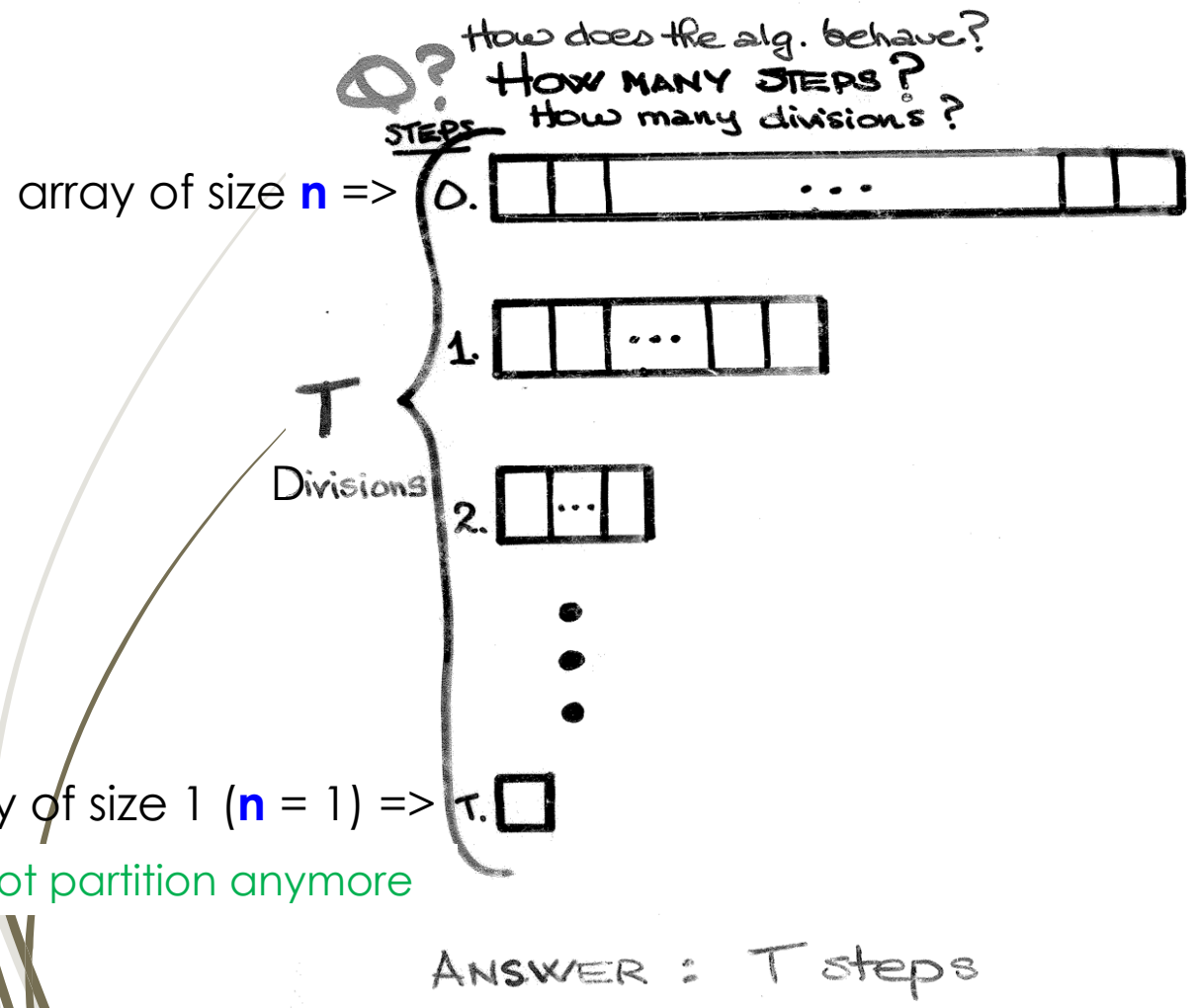
Size of data in 1st iteration: **n**

Size of data in 2nd iteration : **n**/2

Size of data in 3rd iteration : **n**/4

Size of data in Tth iteration : 1

Details of time efficiency analysis: how many times do we partition?



Express T as a function of **n**:

Before we divide: initially : $\frac{n}{2^0}$ = n <= array of size **n**

After 1st division : $\frac{n}{2} = \frac{n}{2^1}$ <= array of size $\frac{n}{2}$

After 2nd division : $\frac{n}{4} = \frac{n}{2^2}$ <= array of size $\frac{n}{4}$

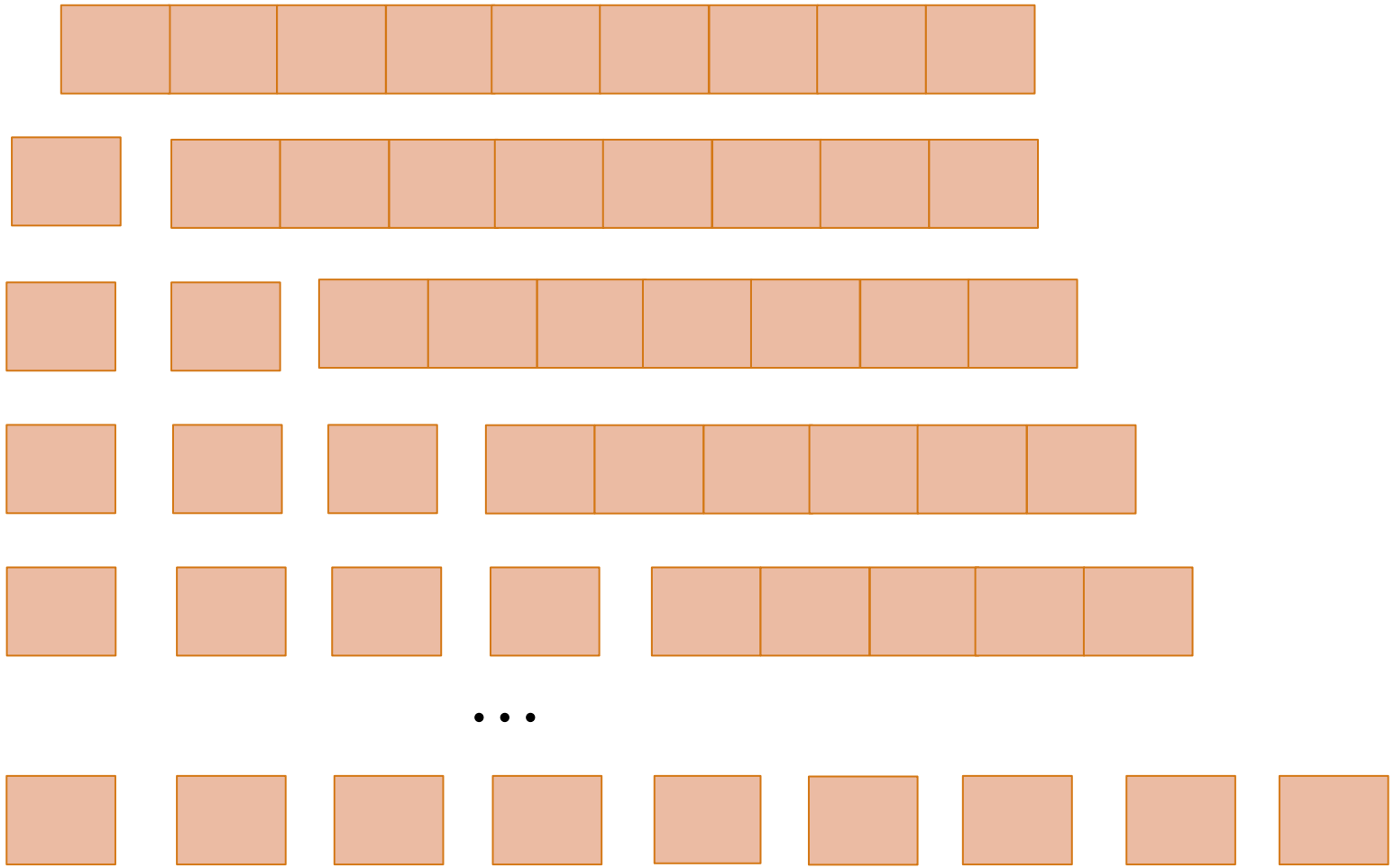
:

After Tth division : $\frac{n}{2^T} = 1 \rightarrow n = 2^T \rightarrow \log_2 n = \log_2 2^T$

$T = \log_2 n$

Number of times we partition: **$O(\log_2 n)$**

Quick Sort – Pattern of the Worst Case Scenario



Time Efficiency Analysis of Quick Sort – Worst Case Scenario

- Each time we partition …
  - We do not divide the data collection in half
    - Selected pivot is such that its value is either the smallest or largest of data collection, so when, when swapped, pivot lands at either end of the partition
    - Effect is that either the partition with elements < pivot is empty or the partition with elements > pivot is empty
  - At most **n - 1** comparisons are made at each level
- How many times do we partition?
  - Every time we partition, all we do is remove the pivot from the partition
    - So every time we partition, we decrement the size of the partition by 1
  - How many times does **n** have to be decremented by 1 before it reaches 1?
  - Answer: **n - 1** times (or **n - 1** partitions)
- Quick sort performs **n - 1** * **n - 1** operations -> $\mathbf{n^2}$ operations in its worst case scenario

For example:

If **n** = 8:

8
7
6
5
4
3
2
1

7

Time Efficiency Analysis of Quick Sort – Average Case Scenario

- Average case scenario of quick sort is closer to its best case scenario than to its worst case scenario

Time Efficiency Analysis of Quick Sort

- Time efficiency of
  - Best case:
  - Average case:
  - Worst case:

- Does the organization of the data in the array to be sorted affect the amount of work done by quick sort?

Improving Quick Sort – Choice of Pivot

- Different ways of selecting the pivot:
  - Way 1: first element
  - Way 2: last element
  - Advantage of ways 1 and 2: simple to implement and time efficient **O(1)**
  - Disadvantage of ways 1 and 2: may lead to worst case scenario

Improving Quick Sort – Choice of Pivot

- Way 3: choose three elements "Median-of-3"
  - First, last and middle elements of data collection
  - Sort order them
  - Pick their median as pivot

To be used if problem statement leads us to believe that the data collection may be (partially) sorted

- Advantage:
  - Still simple to implement and time efficient **O(1)**
  - Reduce risk of creating worst case scenario
- Disadvantage:

Check: https://www.techiedelight.com/iterative-implementation-of-quicksort/ using a Stack
Also check quicksort on Wiki!

Space Efficiency Analysis of Quick Sort

- Quick sort is **in-place** sort algorithm
- Quick sort is often implemented as a recursive function
  - Must keep the amount of required **stack frames** in mind
- How much space (memory) does quick sort (recursive implementation) require to execute?

Space efficiency:

- How much space (memory) does quick sort (iterative implementation) require to execute?

Space efficiency:

Improving Quick Sort – Space efficiency

- To improve quick sort, we could reduce the number of recursive calls
  - Improvement 1: Pivot selection using median-of-3 strategy
  - Improvement 2:
    - Stop using **quick sort** when size of partition small (i.e. < 10)
    - Use **insertion sort** for these small partitions
  - Improvement 3:
    - Use tail recursion
    - Then replace tail recursion with a loop

Is Quick Sort *stable*?

√ Learning Check

- We can now …
  - Describe how quick sort works
  - Analyze the time efficiency of quick sort
  - Improve quick sort's time efficiency
  - Analyze its space efficiency

recursive implementation

  - Improve quick sort's space efficiency

Next Lecture

- Let's have a look at another *efficient* sorting algorithm

Extra Material

Example of Improvement 1 and 2 in Java

```
public void recQuickSort(int left, int right)
{
    int size = right-left+1;

    if (size < 10)    // Insertion Sort if small
       insertionSort(left, right);
    else              // Quick Sort if large
    {
       double median = medianOf3(left, right);
       int partition = partitionIt(left, right, median);
       recQuickSort(left, partition-1);
       recQuickSort(partition+1, right);
    }
}  // end recQuickSort()
```

Improvement 3 - Tail Recursion

- Tail recursive method is a method in which …
  - there is no computation done once the recursive call has returned (recursive call is the last statement of the method), and
  - the entire state of computation is represented in the parameter(s) of the method

Example of **Non-Tail** Recursive Factorial Method

in Java

```
private static int fac( int n ) {
    // Base case
    if ( ( n == 0 ) || ( n == 1 ) )
        return 1;

    // Recursive case
    return ( n * fac( n - 1 ) );
}
```

```
Call: fac( n );
```

Multiplying the return value of the recursive **fac** method with **n** is done after the recursive call has returned (recursive call is not the last statement of the method).

Example of **Tail** Recursive Factorial Method

in Java

```
private static int fac( int n,
                        int result ) {
   // Base case
   if ( ( n == 0 ) || ( n == 1 ) )
        return result;

   // Recursive case
   return fac( n - 1, n * result );
}

Call: fac( n, 1 );
```

- there is no computation done once the recursive call has returned (recursive call is the last statement of the method), and
- the entire state of computation is represented in the parameter(s) of the method

Advantages of Tail Recursion - 1

- Can be transformed into the equivalent iterative implementation of the method
- This is done by replacing the actual recursive call in the method by a loop

Transforming **Tail** Recursive Method into Iterative Implementation of Method

1. Transform "the answer" (here `result`) into a local variable and initialize it to the value in the base case, then remove base case

```
int result = 1;
```

2. Replace the recursive call with a loop which builds the answer:

```
for ( ; n > 1; n-- ) {
    // compute factorial
    result = result * n;
}
```

3. Return the answer

```
return result;
```

```
private static int fac( int n,
                        int result ) {
   // Base case
   if ( ( n == 0 ) || ( n == 1 ) )
        return result;

   // Recursive case
   return fac( n - 1, n * result );
}
```

Example of Iterative Implementation of Factorial Method

in Java

```
// PRE condition: n must be non-negative
public static int fac( int n )
{
   int result= 1;

   // Ensure PRE condition is satisfied

   for ( ; n > 1; n-- ) {
      // compute factorial
      result = result * n;
   }
   return result;
}
```

Advantages of Tail Recursion - 2

Tail Recursion Optimization

- A compiler may recognize "tail recursion" and optimize the code by replacing the calling statement with the called method (with some modifications), so that instead of nesting the stack deeper, the current stack frame is reused when the code execute, i.e., no other stack frame than the first one is needed
- This allows a software developer to write (tail) recursive methods without worrying about space inefficiency during execution
  - The tail recursive code is then as efficient as its iterative version

Source: http://c2.com/cgi/wiki?TailRecursion & http://en.wikipedia.org/wiki/Tail_call