# Quick Sort

Dr. Tanvir Ahmed

## Quick Sort

- Probably Most common sorting technique used in practice
- Usually fastest in practice
- It utilizes the idea of a partition (that can be done without an auxiliary array like merge sort) with recursion to achieve this efficiency.
- It is also a divide and conquer algorithm

## Pivot

- The algorithm picks an element as pivot
- Partitions the given array around the picked pivot
- There are many different versions of quickSort that pick pivot in different ways:
    - Always pick first element as pivot.
    - Always pick last element as pivot
    - Pick a random element as pivot.
    - Pick median as pivot.

## Recursion and Making partitions

- Given an array with n elements
- Select one element as pivot
- Compare all other elements in the array with the selected element
- If an element in smaller,
    - then put it on the left of this element
- If greater,
    - put it on the right
- After one partition run, we know that the selected element (pivot) is in its correctly sorted location
- All the items smaller than the pivot are in the left of pivot
- All the items larger than the pivot are in the right side of the pivot
- Note that, the left array and right array still need to be sorted
- Now if we sort these left side and right side, our entire array will be sorted

## Algorithm

Thus, we have a situation where we can use a partition to break down the sorting problem into two smaller sorting problems. Thus, the code for quick sort, at a real general level looks like:

1) Partition the array with respect to a pivot

2) Sort the left part of the array using Quick Sort

3) Sort the right part of the array using Quick Sort

**Notice that there is no merge step**

- Quick Sort is recursive, so we need a base case (or terminating condition that will not result in more recursive call):
    - When the array size is one, i.e., we have only one element, that array is already sorted. So, no sorting is necessary

## Pseudo Code

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        k = partition(arr, low, high);

        quickSort(arr, low, k - 1);  // Before k
        quickSort(arr, k + 1, high); // After k
    }
}
```

# Example: How to Partition in Place

- **Consider the following list of values to be partitioned:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| data | 5 | 3 | 6 | 9 | 2 | 4 | 7 | 8 |
| | | ^ | | | | | | ^ |

- **Let us assume, our pivot is the first element of the array, *5***
- **Here is how we will partition:**
- **Start two counters, one at array index 1 and the other at array index 7, (which is the last element in the array.)**
- **Advance the left counter forward until a value greater than the pivot is encountered.**
- **Advance the right counter backwards until a value less than the pivot is encountered.**

# Example: How to Partition in Place

**Remember our current pivot is 5**

- **After the two steps discussed in the last slide, we have:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 6 | 9 | 2 | 4 | 7 | 8 |
| | | ^ | | | ^ | | |

- **Now, <u>swap</u> these two elements, since we know that they are both on the "wrong" side:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 9 | 2 | 6 | 7 | 8 |
| | | ^ | | | ^ | | |

- **Now, continue to advance the counters as before (left one first and then the right one):**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 9 | 2 | 6 | 7 | 8 |
| | | | ^ | ^ | | | |

- **Now, swap as we did before**

# Example: How to Partition in Place

**Remember our current pivot is 5**

- **Now, swap as we did before**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 2 | 9 | 6 | 7 | 8 |
| | | | ^ | ^ | | | |
| | | | L | R | | | |

- ***When both counters cross over each*** **other,** ***swap*** **the value stored in the original** ***right counter*** **with the** ***pivot element*****.**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 2 | 9 | 6 | 7 | 8 |
| | | | ^ | ^ | | | |
| | | | R | L | | | |

- **Now, swap the 2 and 5 to yield:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 9 | 6 | 7 | 8 |
| | | | ^ | ^ | | | |
| | | | R | L | | | |

# Example: How to Partition in Place

**Remember our current pivot is 5**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 9 | 6 | 7 | 8 |
| | | | | ^ | ^ | | | |
| | | | | **R** | **L** | | | |

- **Return the index the 5 is stored in to indicate where the partition element ended up in the array.**
- **See in the partition point:**
  - **5 (the pivot we worked on in this iteration) is in the correct position**
  - **Everything before 5 is smaller than 5 *[Remember, the left items might not be sorted. Somehow in this example they got sorted, but it will not be the case always]***
  - **Everything in the right side of 5 is larger than 5. [the right elements are not sorted automatically]**
- **What would you do next?**
  - **Similarly take a pivot in the left array and place it in the same process discussed above**
  - **Similarly take a pivot in the right array and place it**
  - **Recursively do it**

## Code implementation

The full quicksort and test code is available in the webcourses. Here we just copied the quicksort and the partition functions.
We will go thorough the code now.

```
void quicksort(int* numbers, int low, int high) {

    // Only have to sort if we are sorting more than one number
    if (low < high) {
        int split = partition(numbers,low,high);
        quicksort(numbers,low,split-1);
        quicksort(numbers,split+1,high);
    }
}
```

## Code implementation

Randomly generating an index and we will choose the data in that index as the pivot

Swapping and putting our pivot in the low position

lowpos is the index where your pivot is now

Our work starts after lowpos, which is low++
Remember, the first figure of the example in previous slides

We keep working until low and high cross each other. Breaking this loop means, we found our position for pivot to be placed at high (see slide 9).

low index keep going until it finds an item is greater than the pivot (see slide 7). It can break too if low crosses high.

high index keep decreasing until it finds something smaller than or equal to your pivot (see slide 7). It can break too if high crosses low

After completing the loops for low and high, if still (low<high), it means they have not crossed yet. The above loops stopped because they found a value which should not be there. So, we swap the items at low and high. (see slid 8)

You are here because your low and high crossed, now swap your pivot with the high position and then return the portion point, high

```
int partition(int* vals, int low, int high) {
        int temp;
        int i, lowpos;
       // Pick a random partition element and swap it into index low.
        i = low + rand()%(high-low+1);
        swap(&vals[low], &vals[i]);
       // Store the index of the partition element.
       lowpos = low;

        // Update our low pointer.
       low++;

       // Run the partition so long as the low and high counters don't cross.
       while (low <= high) {

              // Move the low pointer until we find a value too large for this side.
              while (low <= high && vals[low] <= vals[lowpos]) low++;

              // Move the high pointer until we find a value too small for this side.
              while (high >= low && vals[high] > vals[lowpos]) high--;

              // Now that we've identified two values on the wrong side, swap them.
              if (low < high)
                 swap(&vals[low], &vals[high]);
       }

       // Swap the partition element into it's correct location.
       swap(&vals[lowpos], &vals[high]);

       return high; // Return the index of the partition element.
}
```

# Analysis of Quick sort run time

- For a correctly sorted array, we can choose any pivot.
- For efficiency, one of following is best case, the other worst case:
  - pivot partitions the list roughly in half
  - pivot is greatest or least element in list

- Which case above is best? Clearly, a partition element in the middle is ideal
  - As it splits the list roughly in half
  - But we don't know where that element is
- So we have a few ways of choosing pivots

## Analysis of Quick sort run time

- Choosing a Partition Element
- first element:
  - bad if input is sorted or in reverse sorted order
  - bad if input is nearly sorted
  - Variation: particular element (e.g. middle element)
- Random element:
  - You could get lucky and choose the middle element
  - You could be unlucky and choose the smallest or greatest element
    - Resulting in a partition with ZERO elements on one side
- median of three elements:
  - choose the median of the left, right, and center elements
  - There is extra expense as picking three values, doing three comparisons
  - But for a large array, this extra work will be small compared to the gain of a better partition
- Median of 5 or 7 elements is also possible with extra work

## Analysis of Quick sort run time

- It is more difficult compared to merge sort as it is not exactly halves at each step.
- Each recursive call of Quick sort could have a different sized set of numbers to sort
- Location of partition element determines difficulty:
  - In the best case, partition element will be always in the middle
  - In the worst case we always choose the first and last element the list as our partition
    - As we will have not really sorted anything
    - We simply reduced the to be sorted amount by 1.
  - In average case:
    - We will get some good partition sometimes
    - We will get some bad partition sometimes
    - We will also get some okay partitions
- So, we will analyze for best case, average case (learn it in CS2), and worst case.

# Analyze Best Case Scenario

- Best case means perfect partition every single time. It means the algorithm becomes:
  - Partition the element
  - Quick sort first half (recursive)
  - Quick sort second half (recursive)
- So, if we have n elements and
  - luckily pick the middle element as the partition element
  - We end up with n/2 – 1 elements on each side
- Let T(n) be the running time of Quick sort of n elements
- Partition function works in O(n) time
- So, T(n) = 2*T(n/2) + O(n) [same recurrence as merge sort and we already solved it]
- So, in best case it is O(nlogn)

## Analyzing Worst Case Scenario

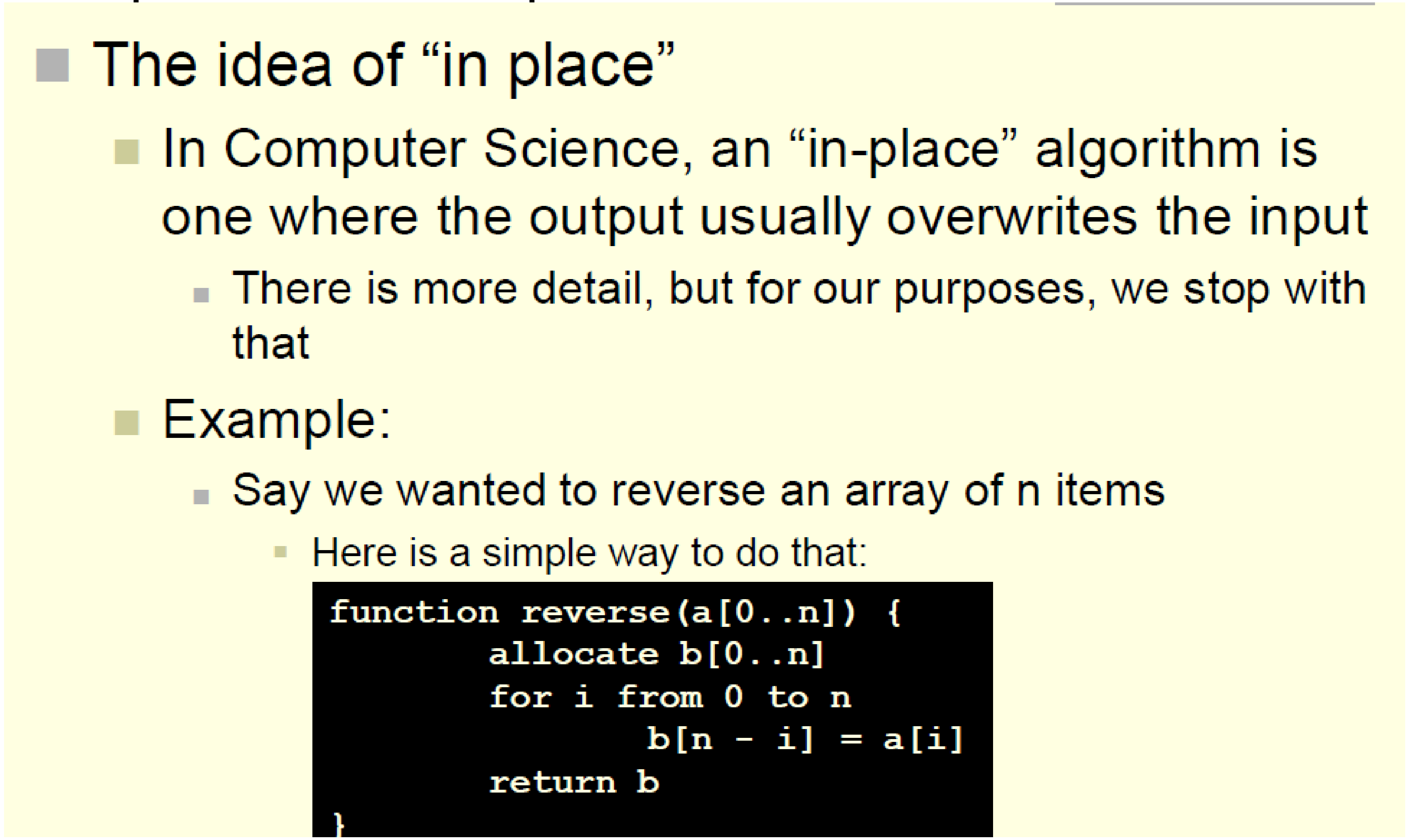
- If we are horribly unlucky the partition will be chosen as the largest or smallest element. So it is not dividing the array at all.
- How many times the partition function will run?
    - If we choose the largest element, then after partition:
    - There will be zero element in the right side
    - All the other items will be in the left side
    - So, Partition will run n-1 times
    - First time results in comparing n-1 values,
    - Second time comparing n-2 values
    - Third time n-3 values, etc.
- Summing: 1 + 2 +3 +….+(n-1)
- n(n-1)/2
- $O(n^2)$

## Summary Analysis

- Quick Sort Analysis Summary:
- Best Case: O(nlogn)
- **Average Case: O(nlogn)**
- Worst Case: O($n^2$)

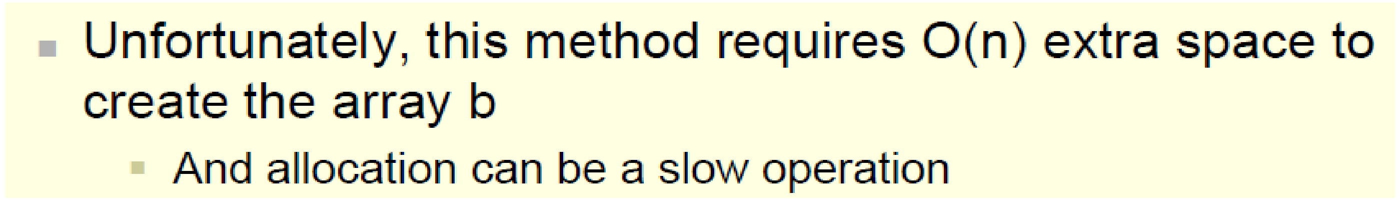
- Compare Merge Sort and Quick Sort:
- Merge Sort: guaranteed O(nlogn)
- Quick Sort: best and average case is O(nlogn) but worst case is O($n^2$)

## Concept about in place



- The idea of “in place”
  - In Computer Science, an “in-place” algorithm is one where the output usually overwrites the input
    - There is more detail, but for our purposes, we stop with that
  - Example:
    - Say we wanted to reverse an array of n items
      - Here is a simple way to do that:

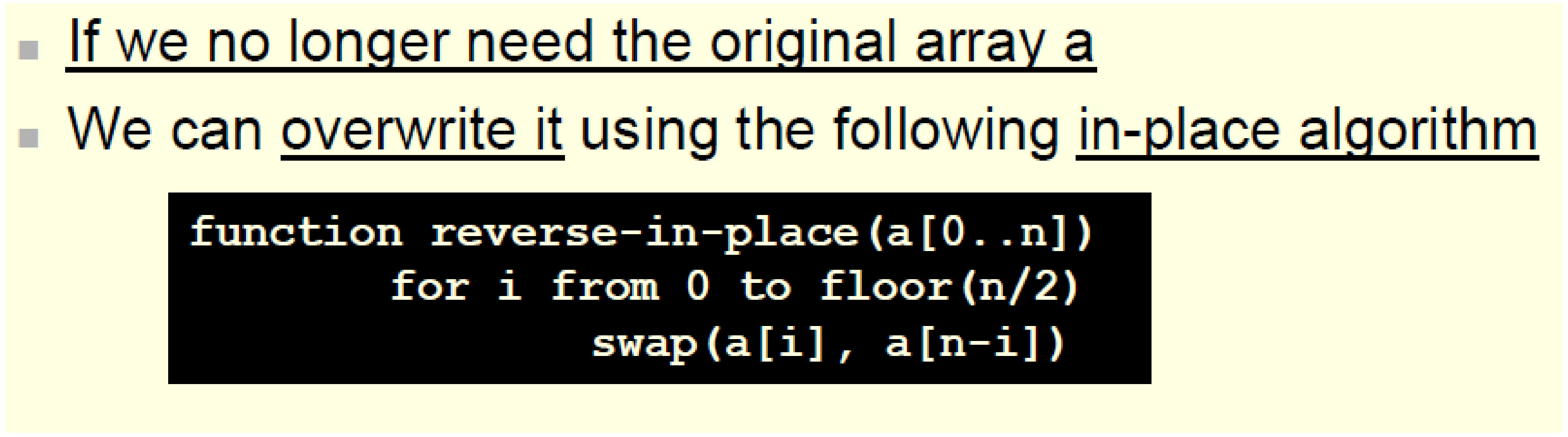
```
function reverse(a[0..n]) {
        allocate b[0..n]
        for i from 0 to n
                b[n - i] = a[i]
        return b
}
```



- Unfortunately, this method requires O(n) extra space to create the array b
  - And allocation can be a slow operation

Slide reference:
https://www.cs.ucf.edu/courses/cop3502/spr2012/notes/COP3502_28_QuickSort.pdf

# Concept about in place



- If we no longer need the original array a
- We can overwrite it using the following in-place algorithm

```
function reverse-in-place(a[0..n])
        for i from 0 to floor(n/2)
                swap(a[i], a[n-i])
```

- In our partition algorithm of the quicksort, we have used in place
- It is actually changing the original array.

## Other ways of partitioning in quick sort

- Reference and reading (Arup’s note): http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/QuickSort.doc
- Full code with automatic testing: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/sampleprogs/quicksort.c
- Acknowledgement:
    - Some slides and text were taken from : https://www.cs.ucf.edu/courses/cop3502/spr2012/notes/COP3502_28_QuickSort.pdf

## Some easy practice

- Choosing pivot from the beginning
- Choosing pivot from right side
- Choosing pivot from a random index

- Do the exercises for all the sorting.