CSCI 132: Basic Data Structures and Algorithms

Sorting (Quick Sort)

Reese Pearsall

Fall 2023

https://www.cs.montana.edu/pearsall/classes/fall2023/132/main.html



Announcements

Program 5 and Lab 12 will be posted very soon (sorry)

hope you are all doing well

When your CS instructor is teaching you about recursive functions and you think you found one





Quick Sort is a sorting algorithm that works by <u>partitioning</u> an array around a certain element in the array, called a <u>pivot</u>. This is a recursive method that then sorts the sections of the array to the left of the pivot, and to the right of the pivot.

Quick sort is a **Divide and Conquer** algorithm, which involves dividing the problem into smaller sub-problems (divide), recursively solving the smaller problems (conquer), and combining the sub problems to get the final solution for the original problem



3

Quick sort is rather complex. I don't expect you to memorize the code, and if you don't fully understand the code, *that is fine!*

You should, however, be able to describe how quick sort works from a high level, and be able to draw out the steps if given an example array

You should also know the time complexity of the sorting algorithms that we talk about





8	2	4	7	1	3	9	6	5

The first step of Quick Sort is to select a pivot, and to get the pivot sorted to is correct position

→ This step is known as **partitioning**





The first step of Quick Sort is to select a pivot, and to get the pivot sorted to is correct position

→ This step is known as **partitioning**

There are many ways to select a pivot, but to keep things simple, the last element of the array will be the pivot





We are going to define two pointers i and j, that will help us get the pivot sorted correctly





j will be defined to be the starting point of the array (0), and i will be defined to be (j-1)



8





9



We will check if index $\ensuremath{ j}$ is less than the pivot





We will check if index j is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

```
If not, increase j by 1
```





j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index j is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```





We will check if index $\ensuremath{ j}$ is less than the pivot

```
If not, increase j by 1
```























Our pivot is now in the correct spot!

Everything to the left is less than the pivot, everything to the right is greater than the pivot



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---

This step is known as partitioning



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---

Now, we will recursively call quick_sort on the left section of the array, and the right section of the array


2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---

Now, we will recursively call quick_sort on the left section of the array, and the right section of the array



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---

Now, we will recursively call quick_sort on the left section of the array, and the right section of the array



Unlike Merge Sort, these are **not** new arrays, these are just "sections" of the original array



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---

Now, we will recursively call quick_sort on the left section of the array, and the right section of the array

Unlike Merge Sort, these are **not** new arrays, these are just "sections" of the original array

Due to how we call our recursive methods, we will always prioritize the "left tree" of the array

2	4	1	3	5	7	9	6	8
								1

Now we partition this section!



2	4	1	3	5	7	9	6	8
								1





2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
---	---	---	---	---	---	---	---	---



We will check if index j is less than the pivot

If so, we will increase \pm by 1, and then swap the elements located at index \pm and \pm



2	4	1	3	5	7	9	6	8
								1

2 1 3 4





2	4	1	3	5	7	9	6	8

2 1	3	4
-----	---	---







2	4	1	3	5	7	9	6	8

2 1	3	4
-----	---	---







2	4	1	3	5	7	9	6	8

2	1	3	4
---	---	---	---







2	4	1	3	5	7	9	6	8

2	1	3	4
---	---	---	---







2	4	1	3	5	7	9	6	8

2 1	3	4
-----	---	---

1 2





2	4	1	3	5	7	9	6	8



1	2







2	4	1	3	5	7	9	6	8

2 1	3	4
-----	---	---

1 2



2



2	4	1	3	5	7	9	6	8

2 1	3 4
-----	-----

1	2





2	4	1	3	5	7	9	6	8

1 2	3	4
-----	---	---





2	4	1	3	5	7	9	6	8

1 2	3	4
-----	---	---





2	4	1	3	5	7	9	6	8
								1

1 2	3	4
-----	---	---



1 2 3 4 5 7 9 6 8





	1	2	3	4	5	7	9	6	8
--	---	---	---	---	---	---	---	---	---



1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





|--|




1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---





Call quick sort, and give it the section of the array to the left of the pivot, and to the right of the pivot



1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---









1 2 3 4 5 7 9 6 8









1	2	3	4	5	7	9	6	8	
---	---	---	---	---	---	---	---	---	--









1 2 3 4 5 7 9 6 8









1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---









1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---

7	6	8	9
---	---	---	---







1	2	3	4	5	7	9	6	8
---	---	---	---	---	---	---	---	---

7	6	8	9
---	---	---	---









|--|

7 6 8	9
-------	---





1	2	3	4	5	7	9	6	8
								1

6 7 8	9
-------	---





1	2	3	4	5	7	9	6	8
								1

6 7	8	9
-----	---	---





1	2	3	4	5	7	9	6	8
								1

6 7	8	9
-----	---	---



1	2	3	4	5	6	7	8	9
								1

All done!! 🖌



1	2	3	4	5	6	7	8	9
								1 .

All done!!

Let's code this!!!



```
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}</pre>
```

```
public static int partition(int[] array, int start, int end) {
    int pivot = array[end];
    int i = start - 1;
    for(int j = start; j <= end - 1; j++) {
        if(array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
}</pre>
```



```
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}</pre>
```

```
public static int partition(int[] array, int start, int end) {
    int pivot = array[end];
    int i = start - 1;
    for(int j = start; j <= end - 1; j++) {
        if(array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        Running
}
</pre>
```

Running time?



```
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}</pre>
```

```
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) { O(n)
        if(array[j] < pivot) { O(1)
            i++; O(1)
            int temp = array[i]; O(1)
            array[i] = array[j]; O(1)
            array[j] = temp;O(1)</pre>
```



```
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}</pre>
```

```
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) { O(n)
        if(array[j] < pivot) { O(1)
            i++; O(1)
            int temp = array[i]; O(1)
            array[i] = array[j]; O(1)
            array[j] = temp;O(1)</pre>
```

Running time of partition subroutine = O(n) where n= # of elements in array



```
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing O(1)
        return array; O(1)
    }
    int pivot = partition(array, start, end); O(n)
    quick_sort(array, start, pivot-1); O(1)
    quick_sort(array, pivot + 1, end); O(1)
    return array; O(1)
    Running time of quick sort method = O(n)
</pre>
```

```
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) { O(n)
        if(array[j] < pivot) { O(1)
            i++; O(1)
            int temp = array[i]; O(1)
            array[i] = array[j]; O(1)
            array[j] = temp;O(1)</pre>
```

Running time of partition subroutine = O(n) where n= # of elements in array

We must now evaluate how often we recursively call the method, and the size of the problem we give that method



MONTANA 93



Ideally, we want **balanced** partitions.

That way we are dividing the problem size by 2 (which gives us log n running time!)





Because our array is random, we don't know how balanced our partitions will be





Because our array is random, we don't know how balanced our partitions will be

(This is a less ideal case)





This is the ideal situation...

This will give us **O(N * log(n))** running time





This is worst case scenario...

This will give us O(N²) running time





How balanced our partitions are depends on how we select the pivot





If we select the **median** value of the array as the pivot, that will always give us the optimal recursion tree and **O(n logn)** running time





If we select a **random value** as the pivot, that will actually give us a much better chance of **O(nlogn)** running time



Running time of Quick Sort

O(n²) WORSE CASE SCENARIO (n recursive calls, O(n) work at each level)

O(n * log n) on average

(logn recursive calls, O(n) work at each level)

