Algorithm Adventures: Sorting, Searching, and Solving the Teacher's Nightmare!

Minor in AI, IIT Ropar
14th February, 2025

Quick Sort Algorithm: A Step-by-Step Guide for Beginners

Welcome back to the world of sorting algorithms! In this module, we'll explore a particularly clever and powerful sorting algorithm called **Quick Sort**.

We will take things slowly, with plenty of examples, and show you how to implement Quick Sort in Python. Finally, we'll discuss how to measure how efficient an algorithm is using a concept called Time Complexity. Let's begin!

The Teacher's Dilemma - An Intuitive Introduction to Quick Sort

Imagine two school teachers tasked with arranging students in a line according to their heights. The students, naturally, prefer standing with their friends and end up in a random order every day. The teacher's challenge is to quickly sort them into ascending order of height.

Let's say the students are initially standing in this order (representing their heights):

7, 6, 1, 5, 4, 9, 2, 3

Now, these teachers came up with a unique method. One teacher, let's call her Teacher A, stands at the beginning of the line. The other, Teacher B, stands at the end.

Teacher A's job is to send the taller students to the *back* of the line. Teacher B, on the other hand, sends the shorter students to the *front*. The idea is that by constantly shuffling students, we can progressively sort the entire line.

The Problem:

Initially, Teacher A looks at the first student (height 7) and tells him to move to the back, as he's tall. But the student protests, "Tall compared to whom?". Teacher B faces a similar problem with the last student (height 3). They need a common reference point.

The Solution:

The teachers decide to pick a student from the middle as a reference point. In our example, there are 8 students, so the "middle" student is the one at position 4, with height 5. They call this the **Pivot**.

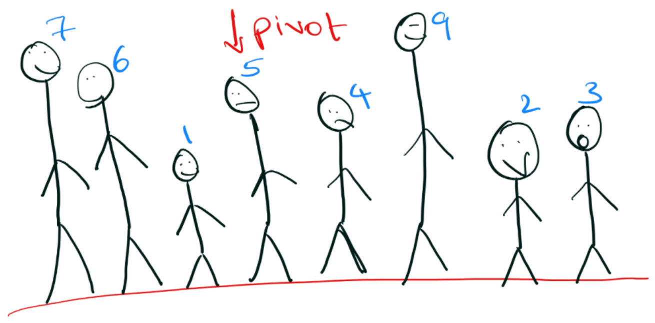

Figure 1: A diagram showing the students lined up, with an arrow pointing to the middle student (height 5) labelled ”Pivot”

Now, Teacher A compares everyone to the Pivot (5). If a student is taller than 5, they’re sent to the back. Teacher B compares everyone to the Pivot as well. If a student is shorter than 5, they go to the front.

The Shuffling Begins:

1. Teacher A tells the first student (7) to go to the back, and Teacher B tells the last student (3) to go to the front. They swap places, giving us:

   3, 6, 1, 5, 4, 9, 2, 7

2. Teacher A moves one position to the right, looking at the student with height 6. Teacher B moves one position to the left, looking at the student with height 2. Both are taller and shorter respectively compared to Pivot (5). They swap places, giving us:

   3, 2, 1, 5, 4, 9, 6, 7

3. Teacher A moves to the student with height 1, noting that it is not taller compared to Pivot (5). So, the teacher moves to the next student with height 5. Teacher A tells the student with height 5 to go to the back and Teacher B tells the student with height 4 to go to the front. They swap places, giving us:

   3, 2, 1, 4, 5, 9, 6, 7

4. As teachers move positions one after the other, Teacher A looks to the position of Teacher B & Teacher B looks at the position of Teacher A. Since both teachers find out that Teacher A is at the left half where all elements are smaller than pivot and Teacher B is at the right half where all elements are bigger than pivot, they realize that their job is done. Teacher A stops at position 4, the left half and Teacher B stops at position 5, the right half.

The Result:

Notice something interesting! All the students shorter than 5 are now on the left side of the line, and all the students taller than 5 are on the right!

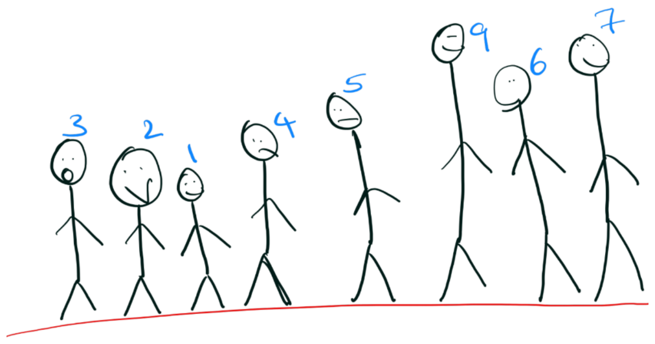

Figure 2: Result of the partition!

This is what Quick Sort does at its core. It picks a pivot, divides the list into two parts (smaller and larger than the pivot), and then repeats the process on each of those parts. By breaking down the problem into smaller and smaller chunks, we can efficiently sort the entire list.

Breaking it Down - The Quick Sort Algorithm

Now that we've got the intuitive understanding down, let's formalize the Quick Sort algorithm.

1. **Choose a Pivot:** Select an element from the list to serve as the pivot. In our teacher example, we picked the middle element. In practice, pivots can be chosen in various ways (first element, random element, etc.).

2. **Partition:** Rearrange the list so that all elements smaller than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot ends up in its final sorted position. The same thing the teachers did in previous section. This step is known as **Partitioning**.

3. **Recursion:** Recursively apply steps 1 and 2 to the sub-lists of smaller and larger elements. This means we repeat the whole process on the left side elements of Pivot and right side elements of Pivot.

Python Implementation - Bringing Quick Sort to Life

Let's write some Python code to implement the Quick Sort algorithm. We'll start by defining a `partition` function, mirroring the teacher's actions of dividing the list around the pivot.

Listing 1: Partition function in Python

```python
def partition(my_list, start, end):
    t1 = start
    t2 = end
    pivot = my_list[(start+end)//2]

    while True:
        while my_list[t1]<pivot:
            t1 += 1
        while my_list[t2]>pivot:
            t2 -= 1
        if t1>=t2:
            return t2

        #swap
        my_list[t1], my_list[t2] = my_list[t2], my_list[t1]
        t1 += 1
        t2 -= 1
```

This `partition` function takes the list (`my_list`), a `start` index, and an `end` index as input. It selects the middle element as the `pivot`. Then, using a `while` loop, it iterates through the list, shifting elements according to whether they are less than or greater than the `pivot`.

Next, let's implement the `quicksort` function, which uses the `partition` function recursively to sort the list.

Listing 2: Quicksort function in Python

```python
def quicksort(my_list, start, end):
    if start<end:
        t2 = partition(my_list, start, end)
        quicksort(my_list,start,t2)
        quicksort(my_list,t2+1,end)
```

This function checks if there's more than one element in the current sub-list (`if start < end`). If so, it calls the `partition` function to divide the sub-list around the pivot. Then, it recursively calls itself on the two resulting sub-lists.

Let's try it out with our initial student height list:

Listing 3: Example usage of quicksort

```python
my_list = [7,6,1,5,4,9,2,3]
quicksort(my_list,0, len(my_list)-1)
print(my_list) # Output: [1, 2, 3, 4, 5, 6, 7, 9]
```

Congratulations! You've successfully implemented Quick Sort in Python!

Measuring Efficiency - Time Complexity

Now we know how Quick Sort works, but how do we compare it to other sorting algorithms? This is where the concept of **Time Complexity** comes in.

What is Time Complexity?

Time complexity is a way of describing how the *number of operations* an algorithm performs grows as the *input size* grows. We are talking about how many instructions (or operations) the computer has to execute to complete the task. Instead of measuring time in seconds (which can vary based on the computer), we count the number of instructions.

Consider these example instructions:

Listing 4: Time complexity Example 1

```
print("Hello") #1 instruction

print("Hello") #2 instructions
print("Hello")

for i in range(0,10):
    print("Hello") #10 instructions

for i in range(0,10):
    print("Hello") #20 instructions
    print("Hello")
```

So, number of instructions can be measured as above.

The Big O Notation

We use something called **Big O notation** to express time complexity. The ”O” stands for ”order of,” and it gives us a general idea of how the algorithm scales.

For example, **O(n)** (read as ”order of n”) means the number of operations grows linearly with the input size (n). If you double the input size, you roughly double the number of operations. An example is the linear search algorithm.

Listing 5: Linear search algorithm

```
key = 5
n = len(my_list) #number of instructions is n
for i in range(0,len(my_list)):
    if(my_list[i]==key):
        print("Found")
```

The code has linear time complexity of O(n).

Understanding O($\mathbf{n^2}$)

If the algorithm executes as follows:

Listing 6: O(n squared)

```
for i in range(0,n):
    for j in range(0,n):
        print(my_list[i])

    print(my_list[i])
```

Then time complexity will be n * n + n = n^2 + n. However, as n becomes very large, n^2 dwarfs n. For example:

If n = 10, the number of instructions will be: 10 * 10 + 10 = 110 instructions

If n = 100, the number of instructions will be: 100 * 100 + 100 = 10100 instructions

If n = 1000, the number of instructions will be: 1000 * 1000 + 1000 = 1001000 instructions

You can see that as n becomes very large, the + n part is relatively insignificant compared to n^2. So, we ignore it and say the time complexity is $\mathbf{O(n^2)}$.

Logarithmic Time - O(log n)

Consider the following algorithm: binary search. The code has logarithmic time complexity. It turns out that the time complexity of the binary search algorithm is O(log n).

The reason binary search achieves O(log n) complexity lies in its divide-and-conquer approach. With each comparison, the algorithm effectively halves the search space. Let's say you have a list of 'n' items.

1st comparison: You check the middle element. If it's not the target, you eliminate half of the list. You're left with approximately n/2 elements to search.

2nd comparison: You check the middle element of the remaining half. Again, you eliminate half of what's left. You're now looking at roughly n/4 elements.

3rd comparison: You're down to about n/8 elements.

This halving continues until you either find the target element or the search space is exhausted (meaning the element is not in the list).

So, how many times can you divide 'n' by 2 until you get down to 1? This is essentially what the logarithm (base 2) is asking. In mathematical terms, we're looking for 'k' such that:

$$\frac{n}{2^k} = 1$$

This can be rewritten as:

$$n = 2^k$$

Taking the logarithm base 2 of both sides:

$$\log_2(n) = k$$

Therefore, the number of comparisons k is proportional to $\log_2(n)$. Since the base of the logarithm is a constant, we express the time complexity in Big O notation as O(log n). This means that as the size of the input (n) increases, the number of operations required by binary search grows logarithmically, making it very efficient for searching large sorted lists.

Listing 7: Binary search algorithm

```
#Binary Search - REVISION OF PREVIOUS CLASS
my_list = [3,4,6,7,9,12,16,17]
target = 13
ans = -1
start = 0
end = len(my_list)-1

while(start<=end):
    mid = (start+end)//2
    if(my_list[mid]==target):
        ans = mid
        break
    elif(my_list[mid]>target):
        end = mid-1
    elif (my_list[mid]<target):
        start = mid+1

print(ans)
```

Time Complexity of Quick Sort

The time complexity of Quick Sort is a bit more nuanced. In the **best-case** and **average-case**, Quick Sort has a time complexity of $\mathbf{O(n \log n)}$. However, in the **worst-case** (when the pivot is consistently the smallest or largest element), it degrades to $\mathbf{O(n^2)}$.

The **best-case** and **average-case** occur when the partitioning step divides the list into roughly equal halves. Then, since teachers are working together to find the correct position in Quick Sort, it saves time in sorting.

That's why, it is named as `quick` sort.

Despite the possibility of a worst-case scenario, Quick Sort is often favored in practice due to its excellent average-case performance and is often used in implementations.

Congratulations on completing this journey through Quick Sort! We started with an intuitive example of teachers sorting students, translated that intuition into a Python implementation, and finally, learned how to measure the efficiency of algorithms using Time Complexity. With this knowledge, you're well-equipped to tackle more complex sorting problems and appreciate the power of algorithmic thinking!