CMSC 351: QuickSort

Justin Wyss-Gallifent

March 24, 2025

1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 Overview

QuickSort works by first choosing an element in the list called the pivot key (at some initial pivot index) and then rearranging the list (including probably moving the the pivot key) so that every element smaller than pivot key is to the left of it and every element larger than the pivot key is to the right of it. This is called the partitioning process.

We then apply QuickSort recursively to the left and right sublists.

When QuickSort is applied to a single element it does nothing, since a single element is always sorted.

The choice of pivot key is nuanced. For now we will consistently choose the final key in the list. If another key is chosen it is simply first exchanged with the final key in the list before proceeding.

3 Partitioning Overview

The encoding of the partitioning process can seem a bit convoluted so it's worth summarizing what is effectively happening in the following way which clarifies that it does what we claim it does.

Pick the leftmost element which is greater than the pivot key and swap it with the first subsequent element which is less than or equal to the pivot key. Repeating until there are no subsequent elements left. The final swap will be with the actual pivot key and the result will be that all elemements to the left of the pivot key will be less than or equal to the pivot key and all elements to the right of the pivot key will be greater than the pivot key.

Example 3.1. Consider the list A with the pivot key $p = 3$.

index	0	1	2	3	4
A	5	2	4	1	3

The leftmost element greater than $p = 3$ is $A[0] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[1] = 2$ so we swap those two:

index	0	1	2	3	4
A	2	5	4	1	3

The leftmost element greater than $p = 3$ is $A[1] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[3] = 1$ so we swap those two:

index	0	1	2	3	4
A	2	1	4	5	3

The leftmost element greater than $p = 3$ is $A[2] = 4$. The first subsequent element less than or equal to $p = 3$ is $A[4] = 3$ so we swap those two:

index	0	1	2	3	4
A	2	1	3	5	4

The leftmost element greater than $p = 3$ is $A[3] = 5$. There are no subsequent elements less than or equal to $p = 3$. Thus we are done.

Observe that the pivot keys is such that all elements to the left are less than or equal to it and all elements to the right are greater than it.

4 Pseudocode

The actual algorithmic implementation is a bit more nuanced and has a small quirk.

```
\\ PRE: A is a list of length n.
\\ Note that A is considered global here.
function quicksort(A,L,R)
    if L < R then
         resultingpivotindex = partition(A,L,R)
         quicksort(A,L,resultingpivotindex-1)
         quicksort(A,resultingpivotindex+1,R)
    end
end
function partition(A,L,R)
    \\ To use a different pivotkey
    \\ swap it with A[R] here.
    pivotkey = A[R]
    t = L
    for i = L to R-1
        if A[i] <= pivotkey
            A[t] <-> A[i]
            t = t + 1
        end
    end
    A[t] <-> A[R]
    return t
end
quicksort(A,0,n-1)
\\ POST: A is sorted.
```

Loosely speaking `t` keeps track of the leftmost key larger than the pivot key and `i` hunts down the subsequent key less than or equal to the pivot key. We say "loosely" because if the list starts with keys less than or equal to the pivot key then the algorithm will swap them with themselves for a while.

Example 4.1. Let's trace the implementation on the following list with $n = 7$ elements. Note that the initial call to `quicksort` is the call `quicksort(A,0,7-1)` which then calls `partition(A,0,6)` which runs `for i = 0 to 5`. We have `L=0` and `R=6`.

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5

The pivot key is `p=5` at index 6.

We set `t=0` and iterate from `i=0`:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
	t,i						p=5

We see `A[i]=A[0]=2<=5` is true and swap `A[i]<->A[t]` which effectively swaps `A[0]<->A[0]`. We increase `t` so now `t=1`. We now have `i=1`:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
		t,i					p=5

We see `A[i]=A[1]=1<=5` is true and swap `A[i]<->A[t]` which effectively swaps `A[1]<->A[1]`. We increase `t` so now `t=2`. We now have `i=2`:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
			t,i				p=5

We see `A[i]=A[2]=6<=5` is false and we do nothing. We now have `i=3`:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
			t	i			p=5

Notice that finally `t` indicates the location of the leftmost element greater than the pivot key. This is when things actually start to happen.

We see `A[i]=A[3]=1<=5` is true and swap `A[i]<->A[t]` which effectively swaps `A[3]<->A[2]`. We increase `t` so now `t=3`. We now have `i=4`:

index	0	1	2	3	4	5	6
A	2	1	1	6	8	4	5
				t	i		p=5

We see `A[i]=A[4]=6<=5` is false and we do nothing. We now have `i=5`:

index	0	1	2	3	4	5	6
A	2	1	1	6	8	4	5
				t		i	p=5

We see `A[i]=A[5]=4<=5` is true and swap `A[i]<->A[t]` which effectively swaps `A[5]<->A[3]`. We increase `t` so now `t=4`. There is no `i` since the loop has ended:

index	0	1	2	3	4	5	6
A	2	1	1	4	8	6	5
					t		p=5

The loop has ended and we do a final `A[t]<->A[R]` which effectively swaps `A[4]<->A[6]`, putting the pivot key in location `t` so that finally we are done this partition process:

index	0	1	2	3	4	5	6
A	2	1	1	4	5	6	8
					t		

Now we are done.

Example 4.2. Let's trace the implementation on the following list with $n = 12$ elements. Note that the initial call to `quicksort` is the call `quicksort(A,0,12-1)` which then calls `partition(A,0,11)` which runs `for i = 0 to 10`. We have `L=0` and `R=11`.

For example consider this list:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
												p=5

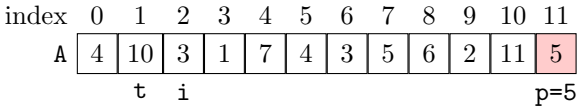
Set `t=0` and `i=0`:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
	t,i											p=5

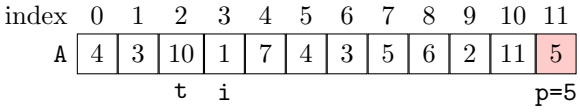
No swap, iterate `i=1`:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
	t	i										p=5

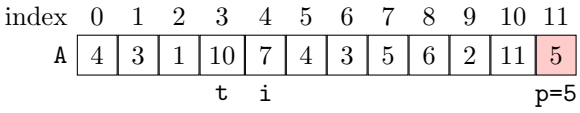
Swap, iterate `t=1` and iterate `i=2`:



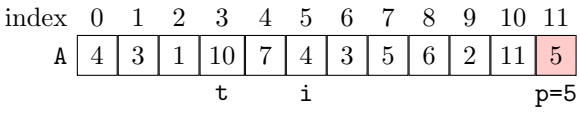
Swap, iterate `t=2` and iterate `i=3`:



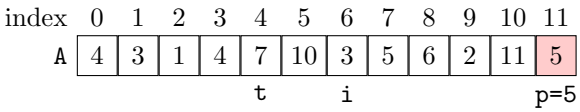
Swap, iterate `t=3` and iterate `i=4`:



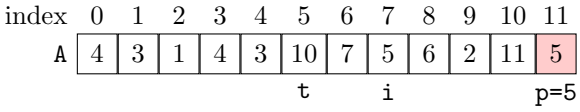
No swap, iterate `i=5`:



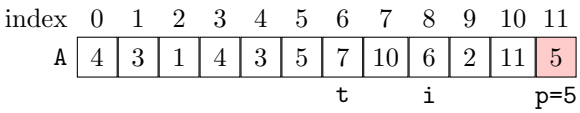
Swap, iterate `t=4` and iterate `i=6`:



Swap, iterate `t=5` and iterate `i=7`:



Swap, iterate `t=6` and iterate `i=8`:



No swap, iterate `i=9`:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	7	10	6	2	11	5
							t			i		p=5

Swap, iterate `t=7` and iterate `i=10`:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	2	10	6	7	11	5
								t			i	p=5

Loop done. Swap `A[t]<->A[R]`:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	2	5	6	7	11	10

5 Pivot Key Choice

For simplicity we choose the last key of the sublist as the pivot but this is perhaps less than idea. For example if the list is already sorted then choosing the last element as the pivot key results in the algorithm taking as long as possible and not actually changing anything - try it on some data and see!

Intuitively if the input is close to sorted then choosing the last element (or in fact the first element) as the pivot will result in very slow running time.

In order to prevent this there are other ways we can choose the initial pivot key. One way is simply randomly. This adds very little time to the process since typically random number generation is $\Theta(1)$. Whether it helps or not is something we'll have to see.

It might be tempting to choose the initial pivot index so that the pivot key is the median, since that results in a nice balanced partition, but this is in and of itself challenging as we'll see later.

6 Pseudocode Time Complexity

Let $T(n)$ be the time complexity of a call to QuickSort.

A call to QuickSort on a list of length n invokes both a partitioning call and two subsequent recursive calls to QuickSort. If the `resultingpivotindex` returns index k then one of them to a sublist of length k and one of them to a sublist of length $n-k-1$

Those recursive calls take time $T(k)$ and $T(n-k-1)$ respectively.

The partition call on a list of length n does some constant time c_1 work and also iterates over $n-1$ elements, say it take time c_2 for each iteration, thus in total it takes $c_1 + c_2(n-1)$.

Hence we have the recurrence relation:

$$T(n) = T(k) + T(n-k-1) + c_2 n + (c_1 - c_2)$$

1. **Worst Case:** The worst-case occurs when the `resultingpivotindex` is the first or last element in the sublist.

   This results in the sublist being only one element smaller than the list itself and the other sublist being length zero. Without loss of generality if $k = 0$ in the above relation we have

   $$T(n) = T(n-1) + c_2 n + (c_1 - c_2)$$

We cannot solve this with the Master Theorem but observing that $T(1) = c_3$ for some constant c_3 we can derive a pattern:

$$
\begin{aligned}
T(1) &= c_3 \\
T(2) &= T(1) + c_2(2) + (c_1 - c_2) = c_1 + c_2 + c_3 \\
T(3) &= T(2) + c_2(3) + (c_1 - c_2) = (c_1 + c_2 + c_3) + 3c_2 + (c_1 - c_2) = 2c_1 + 3c_2 + c_3 \\
T(4) &= T(3) + c_2(4) + (c_1 - c_2) = (2c_1 + 3c_2 + c_3) + 4c_2 + (c_1 - c_2) = 3c_1 + 6c_2 + c_3 \\
T(5) &= T(4) + c_2(5) + (c_1 - c_2) = (3c_1 + 6c_2 + c_3) + 5c_2 + (c_1 - c_2) = 4c_1 + 10c_2 + c_3 \\
T(6) &= T(5) + c_2(6) + (c_1 - c_2) = (4c_1 + 10c_2 + c_3) + 6c_2 + (c_1 - c_2) = 5c_1 + 15c_2 + c_3 \\
\vdots &= \quad \vdots \\
T(n) &= (n-1)c_1 + \frac{n(n+1)}{2}c_2 + c_3
\end{aligned}
$$

This results in $T(n) = \Theta(n^2)$.

2. **Best-Case:** The best-case occurs when the `resultingpivotindex` is in the middle of the sublist.

   This results in the sublists being of equal size. then in the above relation we have

   $$T(n) = 2T(n/2) + c_2 n + (c_1 - c_2)$$

   which results in $T(n) = \Theta(n \lg n)$ by the Master Theorem.

3. **Average-Case:**

   As per earlier discussions we need to understand what "average" means.

   QuickSort is interesting in that the underlying partition method has time complexity $\theta(n)$ which does not depend upon the order of the elements. This is because $n - 1$ comparisons are made in every case, and then some other constant stuff.

   As a consequence the only thing that influences the time complexity of QuickSort is the resulting pivot position after each partition which dictates the sizes of the sublists on which the recursive calls are made.

   Thus one perfectly reasonable way to understand the average case is to not worry about the lists themselves but instead to look at all possible pivot positions and average the time complexity of all of them. In this case we get the following recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-k-1) + \Theta(n)] & \text{otherwise} \end{cases}$$

This is not easy to solve but essentially the idea is to show that this is $\mathcal{O}(n \lg n)$. This is done by constructive induction. I am omitting the proof for now.

7 Auxiliary Space

Suppose $S(n)$ is the auxiliary space required for a list of length n. Consider then:

- In the best case the pivot key ends up in the middle of the list each time. Since the first recursive call to Quicksort frees the memory before the second call and since `partition` requires $\Theta(1)$ auxiliary space the total auxiliary space satisfies the recurrence relation:

  $$S(n) = S(n/2) + \Theta(1)$$

  This yields $S(n) = \Theta(\lg n)$ by Case 2 of the Master Theorem.

- In the worst case the pivot ends up at one end each time. In this case the the total auxiliary space satisfies the recurrence relation:

  $$S(n) = S(n-1) + \Theta(1)$$

  This yields $S(n) = Theta(n)$ by digging down.

8 Stability

QuickSort is not stable. This is because the final swap `A[t] <-> A[R]` could place the pivot key to the left of an equal key.

9 In-Place

QuickSort is in-place.

10 Notes

A few notes:

1. Ideally (mathematically) at each step the median of the keys would be used as the pivot element. The exercises ask you to think about why. There is a method we'll see later called the Median of Medians which can

find the median in $\mathcal{O}(n)$ time. However it's still slow, relatively speaking, and thus...

2. In practice choosing an element randomly is the usual approach, even though this the actual implementation is slightly slower. The exercises ask you to think about why.

3. After k iterations we can draw some conclusions about which elements are correctly placed. The exercises discuss this further.

11 Thoughts, Problems, Ideas

1. Show the steps of the first partition of QuickSort on the list `[10,6,7,2,4,3]`. Use the final index as the initial pivot index.
2. Show the full steps of QuickSort on the list `[10,6,7,2,4,3]`. Use the final index as the initial pivot index.
3. Consider QuickSort used on the list `[11,6,5,45,23,7,2]`. Which initial pivot index and pivot key should be used as the first pivot to ensure that the result of the first partition is equally balanced? Show the result of doing this first `partition`.
4. Suppose the `resultingpivotindex` somehow ended up consistently one third of the way through the list. What would be the correponding recurrence relation? Can the Master Theorem be used to solve this? Can you say anything about the time complexity?
5. Consider the theoretical case where the `resultingpivotindex` consistently ends up 1/2 of the way through the list and the theoretical case where the `resultingpivotindex` consistently ends up 1/4 of the way through the list. Essentially the corresponding recurrence relations would be something like:

$$T_{1/2}(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n)$$
$$T_{1/4}(n) = T(\lfloor n/4 \rfloor) + T(\lfloor 3n/4 \rfloor) + \Theta(n)$$

   Suppose in addition that:

   - The $\Theta(n)$ term is actually $5n + 2$.
   - You know that $T(0) = 0$ and $T(1) = 2$.

   (a) Find each time complexity for keys $n = 0, 10, 20, ..., 100$. (You are welcome to do this in recursive code and if you do, include your code. It's not hard - each is six simple lines of Python, for example.)

   (b) Plot the corresponding data and connect with smooth lines.

   (c) Which of these does your data suggest has a better Θ time complexity? Explain.
6. Give a specific example which illustrates the fact that QuickSort is not stable. Illustrate where in the process the stability breaks down. You don't need to show the entire implementation, just enough to justify.

7. Consider these two approaches to pivot key selection:

   - Obtain index of the median, use as initial pivot index.
   - Choose random index, use as initial pivot index.

   In practice both of these are average case $\Theta(n \lg n)$.

   (a) Mathematically, using the median is better. Why?

   (b) In practice, using a random element is the standard approach. Why not the median?

8. Suppose a list has $n = 2^k - 1$ elements for some k and suppose that somehow, magically, the index of the median is chosen at every stage for the initial pivot index. In this case we can explicitly calculate the number of calls to `quicksort`, the length of each list it is called on, and the number of subsequent calls to `partition`. Remember that `quicksort` only calls `partition` in the case `l<r` so when `l==r` (list of length 1) `quicksort` exits.

   Consider the case where $n = 2^4 - 1 = 15$. Observe there will be:

   - 1 initial call to `quicksort` with a list of length 15, resulting in 1 call to `partition` and then ...
   - a total of 2 calls to `quicksort` with lists of length 7, resulting in a total of 2 calls to `partition` and then ...
   - a total of 4 calls to `quicksort` with lists of length ???, resulting in a total of ??? calls to `partition` and then ...
   - and so on, until it ends.

   (a) Complete the counting argument above - finish the third bullet point and then the remaining bullet points until the argument ends. There should be only four bullet points total.

   (b) What would the counting argument be for $n = 2^k - 1$ for an arbitrary k? It's not hard, just generalize the pattern in (a).

   (c) Suppose that each call to `partition` on a list of length i takes time $c_1 i$. Ignoring all other time requirements (the call to `partition` is the important one) write down and evaluate the sum which gives the total time requirement of the algorithm. Does the time complexity of this result correspond to the best-case analysis?

9. Modify the QuickSort pseudocode so that it chooses the first element as the pivot key.

10. Modify the QuickSort pseudocode so that it randomly chooses a pivot.

11. Modify the QuickSort pseudocode so that it sorts the list in decreasing order.

12 Python Test - Pivot on Final Element

Code:

```
import random
A = []
for i in range(0,15):
    A.append(random.randint(0,100))

A = [5,8,3,4,10,7]


n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        resultingpivotindex = partition(l,r,indent+2)
        quicksort(l,resultingpivotindex-1,indent+2)
        quicksort(resultingpivotindex+1,r,indent+2)
        print(indent*'_' + 'Recombine: '+str(A[l:r+1]))

def partition(l,r,indent):
    print(indent*'_' + 'Subarray: ' + str(A[l:r+1]))
    # To use a different pivotvalue
    # swap it with A[r] here.
    pivotvalue = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivotvalue:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
            print(A)
    temp = A[t]
    A[t] = A[r]
    A[r] = temp
    print(indent*'_' + 'Pivot around final element.')
    print(indent*'_' + 'Result: ' + str(A[l:r+1]))
    return(t)

quicksort(0,n-1,0)
print(A)
```

Output:

```
[50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
__Subarray: [50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
__Pivot around final element.
__Result: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28, 80, 92, 81, 84, 90]
____Subarray: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28]
____Pivot around final element.
____Result: [4, 24, 27, 13, 2, 28, 50, 44, 62, 33]
______Subarray: [4, 24, 27, 13, 2]
______Pivot around final element.
______Result: [2, 24, 27, 13, 4]
________Subarray: [24, 27, 13, 4]
________Pivot around final element.
________Result: [4, 27, 13, 24]
__________Subarray: [27, 13, 24]
__________Pivot around final element.
__________Result: [13, 24, 27]
________Recombine: [13, 24, 27]
______Recombine: [4, 13, 24, 27]
____Recombine: [2, 4, 13, 24, 27]
______Subarray: [50, 44, 62, 33]
______Pivot around final element.
______Result: [33, 44, 62, 50]
________Subarray: [44, 62, 50]
________Pivot around final element.
________Result: [44, 50, 62]
______Recombine: [44, 50, 62]
____Recombine: [33, 44, 50, 62]
__Recombine: [2, 4, 13, 24, 27, 28, 33, 44, 50, 62]
____Subarray: [92, 81, 84, 90]
____Pivot around final element.
____Result: [81, 84, 90, 92]
______Subarray: [81, 84]
______Pivot around final element.
______Result: [81, 84]
____Recombine: [81, 84]
__Recombine: [81, 84, 90, 92]
Recombine: [2, 4, 13, 24, 27, 28, 33, 44, 50, 62, 80, 81, 84, 90, 92]
[2, 4, 13, 24, 27, 28, 33, 44, 50, 62, 80, 81, 84, 90, 92]
```

13 Python Test - Pivot on Random Element

Code:

```
import random
A = []
for i in range(0,13):
    A.append(random.randint(0,100))
n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        pivotindex = partition(l,r,indent+2)
        quicksort(l,pivotindex-1,indent+2)
        quicksort(pivotindex+1,r,indent+2)
        print(indent*'_' + 'Recombine: '+str(A[l:r+1]))
def partition(l,r,indent):
    print(indent*'_' + 'Subarray: ' + str(A[l:r+1]))
    p = random.randint(l,r)
    temp = A[p]
    A[p] = A[r]
    A[r] = temp
    pivot = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivot:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
    temp = A[t]
    A[t] = A[r]
    A[r] = temp
    print(indent*'_' + 'Pivot around index ' + str(p-l))
    print(indent*'_' + 'Result: ' + str(A[l:r+1]))
    return(t)
quicksort(0,n-1,0)
print(A)
```

Output:

```
[92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
__Subarray: [92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
__Pivot around index 2
__Result: [3, 6, 79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
____Subarray: [79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
____Pivot around index 3
____Result: [43, 30, 37, 24, 43, 66, 63, 55, 92, 92, 79]
______Subarray: [43, 30, 37, 24]
______Pivot around index 3
______Result: [24, 30, 37, 43]
________Subarray: [30, 37, 43]
________Pivot around index 2
________Result: [30, 37, 43]
__________Subarray: [30, 37]
__________Pivot around index 1
__________Result: [30, 37]
________Recombine: [30, 37]
______Recombine: [30, 37, 43]
____Recombine: [24, 30, 37, 43]
______Subarray: [66, 63, 55, 92, 92, 79]
______Pivot around index 1
______Result: [55, 63, 66, 92, 92, 79]
________Subarray: [66, 92, 92, 79]
________Pivot around index 0
________Result: [66, 92, 92, 79]
__________Subarray: [92, 92, 79]
__________Pivot around index 1
__________Result: [92, 79, 92]
____________Subarray: [92, 79]
____________Pivot around index 0
____________Result: [79, 92]
__________Recombine: [79, 92]
________Recombine: [79, 92, 92]
______Recombine: [66, 79, 92, 92]
____Recombine: [55, 63, 66, 79, 92, 92]
__Recombine: [24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
Recombine: [3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
[3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
```