1 Introduction

Sorting algorithms on modern computers have been playing an important role in Computer Science and can date back to the early 50s and 60s. It is the great efforts that humans have put into the field of developing faster sorting algorithms that shapes the vital parts that formed all the significant infrastructures of the vast computer world like databases, data centers, cluster networks etc. In the realm of sorting algorithms, Quicksort, introduced by computer scientist Tony Hoare [1] to the public, has made its way into the libc of the GNU/Linux system and stood the test of time as one of the most classical and widely used methods for sorting. Its divide-and-conquer strategy has been widely used and taken into research to fully utilize the performance of sorting. The basic Quicksort picks an arbitrary pivot element, partitions the array into 2 segments, one with elements less than the pivot and the other greater. It then recursively sorts these segments using the same method. While the original Quicksort method has proven effective, the quest of further optimizations and the endless pursuit of efficiency has led to many variations to explore, one of which is the Multi-Pivot Quicksort.

This thesis is my shallow attempt to unveil the benefits of Multi-Pivot Quicksort, exploring the effects more pivots can bring, followed by some potential implementations and the analysis on the benefits that arise from leveraging multiple pivots in the Quicksort process.

1.1 History and Related work

When Quicksort was first invented a great number of variations and modifications were put into research over the years, such as sorting with a better partition method and a more optimal strategy to select the arbitrary pivot element. But the innovative approach of Multi-pivot Quicksort, exemplified by Vladimir Yaroslavskiy [2] in 2009, Bentley and Bloch's algorithm, which was also highlighted by Wild and Nebel, adopted in Sun's Java 7 runtime library, introduces the use of two pivots. Contrary to initial expectations, studies have revealed that employing multiple pivots can enhance the performance, challenging prior assumptions derived from the dual-pivot proposal by Sedgewick [3] in 1978 where he analyzed a potential dual-pivot approach, which was deemed inferior to the classical Quicksort in his research. Furthermore, Kushagra presented a 3-pivot Quicksort [4] algorithm at ALENEX, has attracted significant attention, further pushing the researches of multi-pivot strategies.

One common pitfall in the performance downgrade of the conventional Quicksort is the degenerated cases, which could often arise when the input array is already mostly sorted or when certain patterns in the data cause the algorithm to consistently choose poor pivots that fail to split the data into evenly sized partitions. In such cases, the partitioning process may result in highly unbalanced partitions, leading to suboptimal performance and potentially degrading the time complexity to $\mathcal{O}(n^2)$ instead of the expected $\mathcal{O}(n \log n)$ for Quicksort. Wiser choices of pivot selections could decrease the possiblities of such events from happening, such as the median-of-3 method and its variations of median-of-5 and median-of-7. But on

mostly ascending or descending arrays, the median-of-3 method still has a chance of choosing the smallest or largest element as the pivot, which is not the best ideal approach to prevent the worst case from happening. It also brings to our concerns that in real world applications, duplicated elements are not rare, and the median-of-3 method could be easily affected by the duplicated elements, which could lead to the same worst case as the original Quicksort. There are some other variations of Quicksort that have been proposed to address these specific issues, such as the Introsort algorithm [5] by David Musser, which switches to Heapsort when the recursion depth exceeds a certain threshold. Thus the worst running time is guaranteed to be $\mathcal{O}(n \log n)$ by doing so.

And this is where Multi-pivot quicksort comes into play. Assuming we are dealing with randomly generated arrays, since we don't know the exact size of each partition, the more pivots we choose, the more partitions we divide the array into, the less chance it will be that we encounter imbalanced sizes of partitions. Moreover, choosing more pivots will significantly reduce the chance we access the same elements again and the maximum depth of recursion, which can translate into better running time. With n pivots we are dividing the array into n+1 partitions, the same things happen on the sub-partitions in the next level of recursion and we can conclude the maximum depth of recursion would be $\log_{n+1} L$ given $\mathbf{n}$ as the number of pivots and $\mathbf{L}$ for the length of array. Greater the n is, less the depth. Better memory behaviors can bring effects that can not be achieved by any of other traditional means such as choosing a better pivot value among samples extracted from different parts of the array, but the drawbacks are also obvious, branch misprediction could be terrible as the different outcomes of comparisons can largely impact on the runtime performance. A delay of 14 to 19 stages on a typical Intel desktop CPU or 20 to 25 stages on AMD's Zen CPUs could lead to huge efficiency loss for our algorithm, and this number only grows as the number of pivots increases and on more complicatad architectures. These are the trade-offs we have to consider when we are choosing the number of pivots, although these side effects can be mitigated by the use of branchless comparisons and modern CPUs are able to run conditional move operations (cmove) to avoid branches, these overheads still remain as concerns. Apart from that, it could be really tricky when it comes to moving the elements to the right places among multiple partitions, especially when the number of pivots is large. Additionally, comparisons are the second expensive things next to swaps we are dealing with, and the more pivots we choose, the more comparisons we have to make. Even for the case of using only one pivot, the average number of comparisons remain unoptimial, being $1.38n \log n + \mathcal{O}(n)$ compared with the $n \log n + \mathcal{O}(n)$ of the Merge Sort, but the overall instructions are much less than the average of Merge Sort and with proper pivot selection strategy, it can still be reduced to a certain extent.

Fortunately, an efficient approach has been put forward by Stefan Edelkamp and Armin Weiß, in their thesis of BlockQuickSort [6] which evens the odds of both branch mis-prediction and cache misses which uses one or more buffer blocks of constant sizes on the stack to store the index offsets of out-of-order elements, swap them altogether in a second pass to rearrange them in order after scanning a large chunk of array with the size of the block. Their approach realized a significant speedup up to 80% faster compared with the GCC implementation of std sort with only one

single pivot. The BlockQuickSort has been adapted to the Multi-Pivot Quicksort and the results are also promising, the improvements on contiguous memory access significantly improves the spatial locality and reduces both the cache misses and branch mis-predictions. Branchless comparisons are also adapted which plays an important role in contributing to the huge performance boost by reducing the overhead when increasing the counter of elements stored in the block buffer(s). This novel approach has also developed variations that could effectively make the use of different partitioning methods and multiple pivots selected.

Meanwhile, in the pursuit of advancing the efficiency of the Quicksort algorithm, my investigation led me to explore various modifications, with a particular focus on the pattern-defeating Quicksort (PDQSort) proposed by Orson Peters. PDQSort leverages the average-case performance of randomized Quicksort with the rapid worst-case execution of heapsort, while concurrently achieving linear time complexity for slices exhibiting specific patterns. Notably, PDQSort employs a judicious application of randomization to prevent the degenerated case from happening. When it does, PDQSort will strategically shuffle or reverse the entire array to simplify the sorting and increase the speed. If all the counter measures fail, it also has the final backup plan of switching to Heapsort when bad choices of pivots or the depth of recursions exceed the limit. In terms of pivot selection, it uses a fixed seed when generating a pseudo-random index to ensure the reproducibility and deterministic behavior. Furthermore, in instances where randomly selected elements from the original array appear in a descending order. The combination of these optimizations has demonstrated a significant enhancement in performance, showcasing PDQSort as up to twice faster than the original Quicksort algorithm. This outcome illuminates the transformative impact that a comprehensive analysis on the pattern of arrays has on the speed of sorting algorithms, not to mention it is a successful combination of BlockQuickSort's partition method, Heap sort when dealing with worst cases and Insertion sort when it comes to small sub-arrays of sizes less than the cache line size. Inspired by PDQSort's success, I dive into the diverse variations of Quicksort, motivating a search for novel patterns that may be leveraged to enhance the existing Quicksort algorithms.

1.2 Contributions

- Present the variants of Multi-Pivot QuickSort and implement a 4-Pivots Quicksort and evaluate N-Pivots Quicksort from N = 1, 2, 3, 4 experimentally. Comparing the implementations, times, instructions, cache misses and branch mis-predictions with a duration of 3 seconds warm-up time.

- Present a new layout of Multi-Pivot BlockQuickSort that could potentially reduce the memory access and increase the block buffer usage rate. Analyze this method with all other available variants with block partitioning and compare the pros and cons.

2 Preliminaries

Algorithm 1 QuickSort with Hoare Partition

```
1: procedure QUICKSORT(A, l, r)
2:     if l < r then
3:         p ← HOAREPARTITION(A, l, r)
4:         QUICKSORT(A, l, p-1)
5:         QUICKSORT(A, p+1, r)
6:     end if
7: end procedure
8: procedure HOAREPARTITION(A, l, r)
9:     P ← A[l]                                          ▷ Pivot
10:    i ← l − 1
11:    j ← r + 1
12:    while i < j do
13:        repeat
14:            j ← j − 1
15:        until A[j] < P
16:        repeat
17:            i ← i + 1
18:        until A[i] > P
19:        SWAP(A[i], A[j])
20:    end while
21:    SWAP(A[l], A[j])                                  ▷ Put pivot in the middle
22:    return j
23: end procedure
```

For Quicksorts with single pivot value, the partitioning procedure splits the array into 2 parts where the left part are elements less than the pivot and the right part being greater, separated by the pivot value in the middle. Then the main body recursively sorts the left and right part to rearrange all the elements in the correct order. The hoare's partition method uses double pointers starting at the leftmost and the rightmost element in the array and scanning towards the middle until they meet. Each pointer scans for mis-placed elements which belong to the other partition by comparing the current element to the pivot. Out-of-order elements are swapped in pairs to the right place and the scanning procedure continues until all elements are moved to the places where they belong.

Algorithm 2 QuickSort with Lomuto Partition

```
1: procedure QUICKSORT(A, l, r)
2:     if l < r then
3:         p ← LOMUTOPARTITION(A, l, r)
4:         QUICKSORT(A, l, p-1)
5:         QUICKSORT(A, p+1, r)
6:     end if
7: end procedure
8: procedure LOMUTOPARTITION(A, l, r)
9:     P ← A[r]                                   ▷ Pivot
10:    i ← l
11:    for j ← l to r − 1 do
12:        if A[j] < P then
13:            SWAP(A[i], A[j])
14:            i ← i + 1
15:        end if
16:    end for
17:    SWAP(A[i], A[r])                           ▷ Put pivot in the middle
18:    return i
19: end procedure
```

While the Lomuto partition method uses another way, it scans the array in sequential order from left to right using one bare pointer and swaps all the elements less than the pivot to the left side regardless, thus duplicated swaps are introduced due to not checking if the elements are already in the right places. Both of these 2 partition methods are straightforward, and their main structures are similar: scan, find mis-placed elements, swap and recursively repeat the same procedure until the whole array is sorted.

Algorithm 3 Dual-Pivot QuickSort

```
1: procedure DUALPIVOTPARTITION(A, l, r)
2:     P1, P2 ← A[l], A[r]                                   ▷ Pivots
3:     if P1 > P2 then
4:         SWAP(P1, P2)
5:     end if
6:     less ← l + 1
7:     greater ← r − 1
8:     k ← l
9:     while k ≤ greater do
10:        if A[k] < P1 then                  ▷ If the current element is less than P1
11:            SWAP(A[k], A[less])            ▷ Shift the element to the leftmost partition
12:            less ← less + 1
13:        else if A[k] > P2 then             ▷ If the current element is greater than P2
14:            while A[greater] > P2 and k < greater do
15:                greater ← greater − 1      ▷ Find the first element less than P2
16:            end while
17:            SWAP(A[k], A[greater])         ▷ Swap to the rightmost partition
18:            greater ← greater − 1
19:            if A[j] < P1 then              ▷ Double check if another swap is needed
20:                SWAP(A[k], A[less])
21:                less ← less + 1
22:            end if
23:        end if
24:        k ← k + 1                          ▷ Move to the next element
25:    end while
26:    less ← less − 1
27:    greater ← greater + 1
28:    SWAP(A[l], A[less])                    ▷ Put Pivot1 in the middle
29:    SWAP(A[r], A[greater])                 ▷ Put Pivot2 in the middle
30:    return less, greater
31: end procedure
```

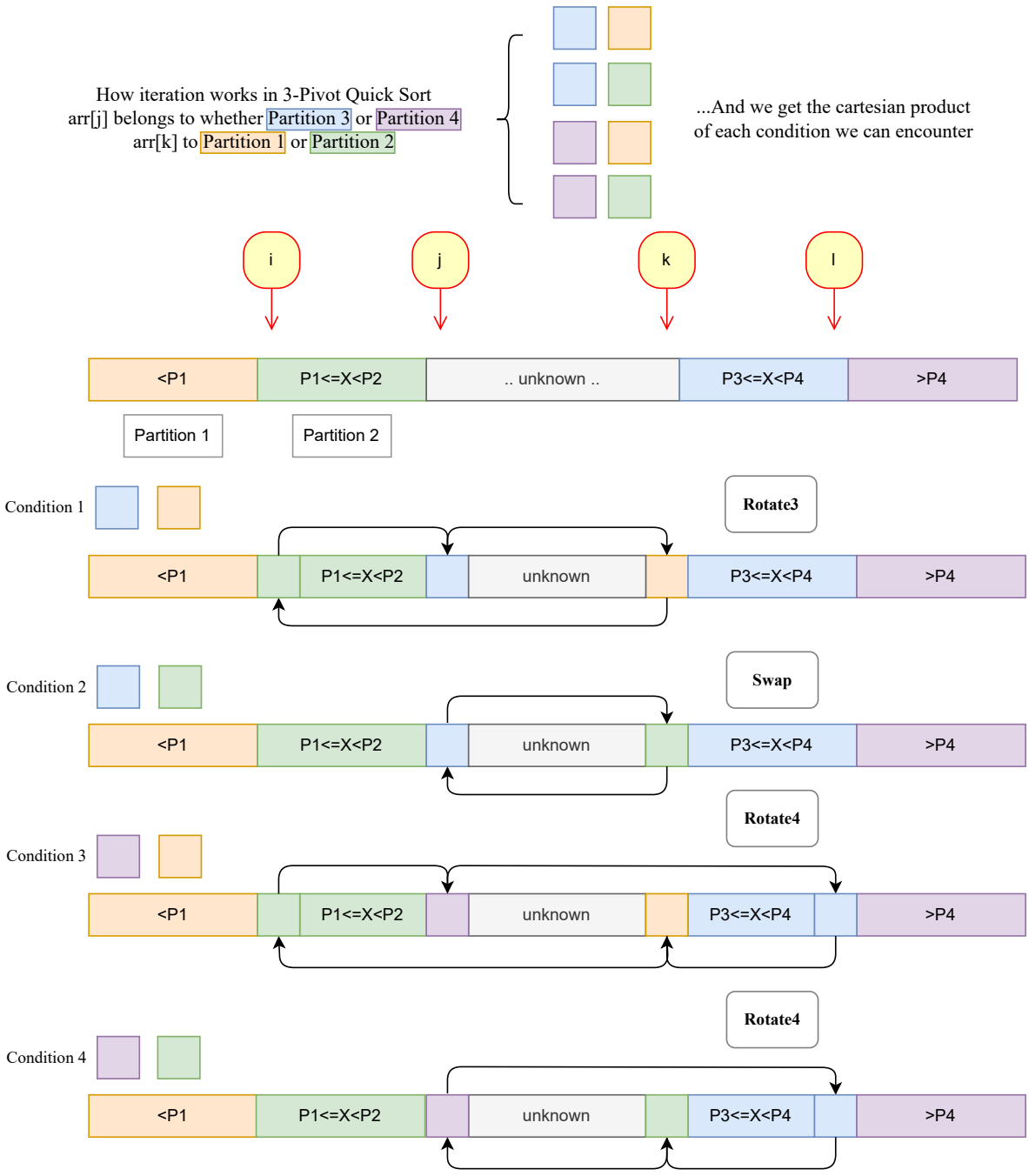
With one more pivot added, the two-pivot Quicksort method is introduced by Vladimir Yaroslavskiy in 2009. It uses 2 pivots selected from each end to split the array into 3 parts. K stands for the current element being scanned, it will be swapped to the left part, which is accumulated by the less pointer, if it is less than the first pivot or do nothing if it is in the middle part. Otherwise, if it is greater than the second pivot, it will be swapped to the right part, which is accumulated by the greater pointer. We will double check the current element at index k after the swap to see if it is less than the first pivot, if so, we will swap it to the left part and move the less pointer to the right. Hereby, we can see that the Dual-Pivot Quicksort is a generalization of the original Quicksort, and the partitioning method is also a generalization of the original Hoare's partition method but with 2 pivots.

Algorithm 4 3-Pivot QuickSort

```
1: procedure ThreePivotPartition(A, l, r)
2:     P1, P2, P3 ← A[l], A[l+1], A[r]          ▷ And sort 3 Pivots in ascending order
3:     i, j ← l+2
4:     k, l ← r−1
5:     while j ≤ k do
6:         while A[j] < P2 do                    ▷ Put left side elements less than P2 in order
7:             if A[j] < P1 then
8:                 Swap(A[i], A[j])
9:                 i ← i+1
10:            end if
11:            j ← j+1
12:        end while
13:        while A[k] > P2 do                    ▷ Put right side elements greater than P2 in order
14:            if A[k] > P3 then
15:                Swap(A[k], A[l])
16:                l ← l−1
17:            end if
18:            k ← k−1
19:        end while
20:        if j ≤ k then
21:            if A[j] > P3 then                 ▷ Deal with 2 branches when A[j] is greater than P3
22:                if A[k] < P1 then
23:                    Rotate3(A, [j, i, k])
24:                    i ← i+1
25:                else
26:                    Swap(A[j], A[k])          ▷ A[j] > P3 and P2 > A[k] ≥ P1
27:                end if
28:                Swap(A[k], A[l])
29:                l ← l−1
30:            else                              ▷ Deal with 2 branches when A[j] is less than or equal to P3
31:                if A[k] < P1 then
32:                    Rotate3(A, [j, i, k])
33:                    i ← i+1
34:                else
35:                    Swap(A[j], A[k])
36:                end if
37:            end if
38:            j ← j+1, k ← k−1
39:        end if
40:    end while
41:    i ← i−1, j ← j−1, k ← k+1, l ← l+1
42:    Rotate3(A, [left+1, i, j])
43:    Swap(A[left], A[i])
44:    Swap(A[right], A[l])
45:    return i, j, l
46: end procedure
```

The three-pivot Quicksort, introduced by Shrinu Kushagra, uses the similar hoare-like partition method, is a further generalization of the Dual-Pivot Quicksort. In the original implementation, the 3 pivots are selected from the leftmost, the second element and the rightmost element of the array, and are sorted in ascending order. The scanning procedure is similar to the Dual-Pivot Quicksort, but with 3 pivots, the array is divided into 4 parts, and is repeated until all elements are set. The i, j, k, l pointers are used to scan the array and the elements are swapped to the right places according to the comparison results with the 3 pivots, where i stands for the leftmost part where elements less than Pivot 1 belongs, j stands for the middle-left part, k stands for the middle-right part and l stands for the rightmost part. Between j and k stands the unknown region where elements are not yet compared with any of the pivots. If the elementa at index j is less than Pivot 2, it is whether swapped to the leftmost part if it is less than Pivot 1, or do nothing otherwise, since the middle-left part is for elements greater than Pivot1 and less than or equal to Pivot 2. Same logic applies to the rightmost part, where elements are greater than Pivot 2 and less than or equal to Pivot 3. These two pre-conditions are checked before the main body of the swapping and rotating procedure, which corresponds to the 2 while loops applied at the top of the main scanning loop. Afterwards, the real swapping and rotating procedure is applied to the elements at index j and k, which are on the left and right boundaries of the unknown region, need to go to the other side of the array and moved to the right places. We can simplify this part into a 2 if-else nested block, where the outter if-else block is for the case at index j and the inner one is for the case at index k. Both of them have 2 branches, one for the case where the current element belongs to the leftmost or the rightmost part, and the other for the case where it belongs to the middle-left or the middle-right part. It is not hard to tell there are $2 \times 2 = 4$ branches in total, and the swapping and rotating procedure is applied to each of them. The following chart * illustrates the 4 branches and the corresponding actions.

Figure 1: 3-Pivot Quick Sort



Algorithm 5 4-Pivot QuickSort

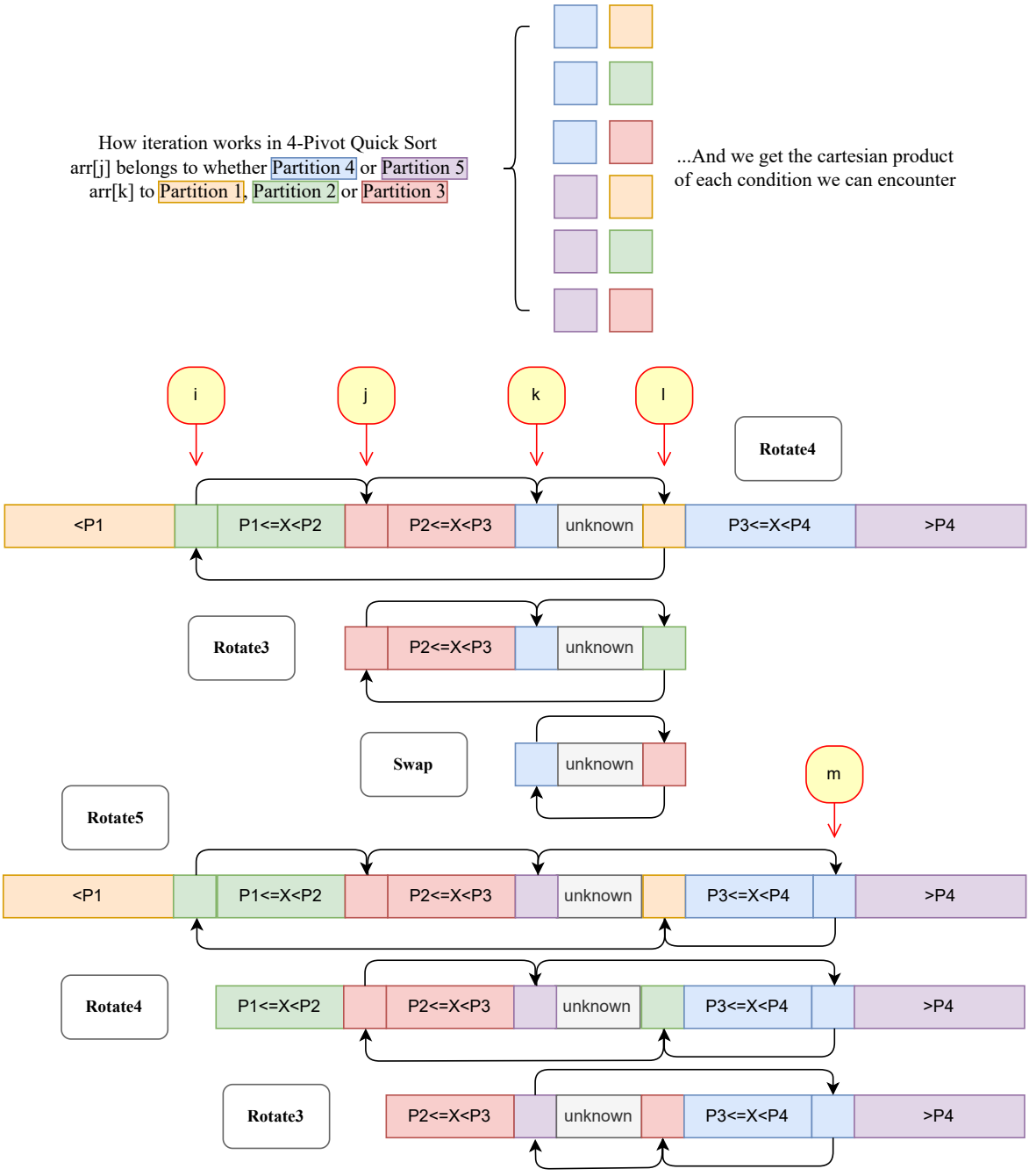
```
1: procedure FourPivotPartition(A, l, r)
2:     P1, P2, P3, P4 ← A[l], A[l+1], A[r−1], A[r]        ▷ And sort 4 Pivots in order
3:     i, j, k, l, m ← l+2, l+2, l+2, r−2, r−2
4:     while k ≤ l do
5:         while A[k] < P3 do                    ▷ Put left side elements less than P3 in order
6:             if A[k] < P1 then
7:                 Rotate3(A, [k, j, i])
8:                 i ← i+1, j ← j+1
9:             else if A[k] < P2 then
10:                Swap(A[k], A[j])
11:                j ← j+1
12:            end if
13:            k ← k+1
14:        end while
15:        while A[l] > P3 do          ▷ Put right side elements greater than P3 in order
16:            if A[l] > P4 then
17:                Swap(A[l], A[m])
18:                m ← m−1
19:            end if
20:            l ← l−1
21:        end while
22:        if k ≤ l then    ▷ Start manipulation on the boundaries of unknown region
23:            if A[k] < P4 then        ▷ Deal with 3 branches when A[k] is less than P4
24:                if A[l] < P1 then
25:                    Rotate4(A, [k, j, i, l])
26:                    i ← i+1, j ← j+1
27:                else if A[l] < P2 then
28:                    Rotate3(A, [k, j, l])
29:                    j ← j+1
30:                else
31:                    Swap(A[k], A[l])
32:                end if
33:            else                    ▷ Deal with 3 branches when A[k] is greater than P4
34:                if A[l] > P2 then
35:                    Rotate3(A, [k, l, m])
36:                else if A[l] > P1 then
37:                    Rotate4(A, [k, j, l, m])
38:                    j ← j+1
39:                else
40:                    Rotate5(A, [k, j, i, l, m])
41:                    i ← i+1, j ← j+1
42:                end if
43:                m ← m−1
44:            end if
45:            k ← k+1, l ← l−1
46:        end if
```

Algorithm 6 4-Pivot QuickSort (Continued)

```
47:     end while
48:     i ← i − 1, j ← j − 1, k ← k − 1, l ← l + 1, m ← m + 1
49:     ROTATE3(A, [left + 1, i, j])
50:     i ← i − 1
51:     SWAP(A[left], A[i])
52:     ROTATE3(A, [right − 1, m, l])
53:     m ← m + 1
54:     SWAP(A[right], A[m])
55: end procedure
```

As long as the number of pivots increases, the number of branches will grow linearly, and the complexity of the partitioning method will also increase. Thankfully, for 4-Pivot QuickSort *, we are just adding two more branches (As showed in the blue-red and purple-red branches in the chart below) into considerations and the branching logic is still under control. It is for sure that we are going to have 9 branches for 5-Pivot QuickSort (3 partitions on each side, $3 \times 3 = 9$) and 12 branches for 6-Pivot QuickSort (3 partitions on one side and 4 on the other, $3 \times 4 = 12$). The complexity of the partitioning method is growing way too far and thus only 4-Pivot QuickSort will be discussed in this thesis.

Figure 2: 4-Pivot Quick Sort



The next part of the thesis will be the implementation of the Multi-Pivot Quicksort and the BlockQuickSort, followed by the analysis of the results and the comparisons on the performance of these algorithms.

Branch Misses	Size	Mean	SD	Median
1-Pivot Hoare	100	211.5502	0.0608	211.5546
	10k	3442.7053	2.1831	3442.5371
	10M	47572.1385	78.0503	47582.3149
1-Pivot Lomuto	100	208.4046	0.2538	208.3322
	10k	3427.8026	2.7327	3427.9686
	10M	47567.6013	71.8560	47559.8385
2-Pivot Yaro	100	218.2533	0.1106	218.2404
	10k	3626.2804	1.4727	3626.4159
	10M	50123.8209	72.2654	50129.0911
3-Pivot Kush	100	230.1718	0.0892	230.1822
	10k	3819.6887	1.9794	3819.1637
	10M	52642.3847	54.1679	52656.3930
4-Pivot	100	231.3465	0.0875	231.3237
	10k	3818.6932	2.2639	3818.4127
	10M	52185.5931	64.8748	52183.4357

Cache Misses	Size	Mean	SD	Median
1-Pivot Hoare	100	0.0818	0.1691	0.0184
	1000	0.7063	0.8911	0.3544
	10000	12.4844	17.9061	7.6416
1-Pivot Lomuto	100	0.0202	0.0301	0.0095
	1000	0.4810	1.0869	0.1625
	10000	6.5238	15.5592	2.5491
2-Pivot Yaro	100	0.0891	0.1247	0.0557
	1000	0.6540	1.1752	0.3613
	10000	2.5734	1.4644	2.1219
3-Pivot Kush	100	0.0244	0.0087	0.0223
	1000	0.4488	0.9355	0.1930
	10000	5.2864	5.4446	3.6768
4-Pivot	100	0.0263	0.0416	0.0159
	1000	0.3894	0.6758	0.1691
	10000	10.9905	19.5457	5.1953

CPU Cycles	Size	Mean	SD	Median
1-Pivot Hoare	100	8833.5322	3.7385	8832.9757
	1000	133660.4883	54.1056	133657.3592
	10000	1811140.8549	54157.2600	1793886.9439
1-Pivot Lomuto	100	8831.1909	18.8702	8827.8629
	1000	135551.1548	672.3558	135278.5342
	10000	1823471.6121	1392.4438	1823461.0815
2-Pivot Yaro	100	8737.9368	68.0570	8722.9634
	1000	135167.3287	923.0637	134889.7150
	10000	1814578.1631	2082.3356	1813979.9786
3-Pivot Kush	100	8580.3270	3.2015	8579.4317
	1000	129564.1759	71.3810	129560.2397
	10000	1720134.7766	1718.4817	1719931.7412
4-Pivot	100	8724.2381	1.7123	8724.0691
	1000	131284.4391	37.8241	131282.8758
	10000	1740851.2642	1393.4360	1740909.0583

Time (µs)	Size	Mean	SD	Median
1-Pivot Hoare	100	2.0344	8.9433	2.0323
	1000	31.126	122.78	31.068
	10000	412.78	2.5557	411.99
1-Pivot Lomuto	100	2.0476	10.402	2.0428
	1000	31.478	193.95	31.369
	10000	426.16	1.4271	425.74
2-Pivot Yaro	100	8737.9368	68.0570	8722.9634
	1000	135167.3287	923.0637	134889.7150
	10000	1814578.1631	2082.3356	1813979.9786
3-Pivot Kush	100	8580.3270	3.2015	8579.4317
	1000	129564.1759	71.3810	129560.2397
	10000	1720134.7766	1718.4817	1719931.7412
4-Pivot	100	8724.2381	1.7123	8724.0691
	1000	131284.4391	37.8241	131282.8758
	10000	1740851.2642	1393.4360	1740909.0583

References

[1] Hoare, C.A.R. (1962). Quicksort. *The Computer Journal*, 5, 10-15.

[2] Yaroslavskiy, V. (2009). Dual-Pivot Quicksort. *Proceedings of the 6th International Conference on Computer Science and Education*, 13-27.

[3] Sedgewick, R. (1978). Implementing Quicksort Programs. *Communications of the ACM*, 21(10), 847-857.

[4] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. *Multi-pivot quicksort: Theory and experiments*. In ALENEX, pages 47-60, 2014.

[5] David R. Musser. *Introspective Sorting and Selection Algorithms*. Software: Practice and Experience, 27(8):983-993, 1997.

[6] BlockQuicksort: How Branch Mispredictions don't affect Quicksort. *Stefan Edelkamp and Armin Weiß*. 2016.