Applied Algorithms 2020 (KSAPALG1KU): Final Project

Ewa Agata Habierska Marta Palade Alexandra Waldau
{ewha, marpa, alew}@itu.dk

December 18, 2020

1 Part 1

1.1 Task 1

The Binary Search implementation can be found under the following path: part1/BinarySearch.java. We have implemented the algorithm by initially setting the range from 0 (variable called left) up to the length of the array (variable called right). The range is continuously updated, based on if the query integer y is bigger or less than the middle index. If no predecessor is found, the implementation returns -1. Otherwise, the algorithm should return the index of the searched element or an index of a first smaller number in the array if the searched element cannot be found. For this purpose the main method that performs the binary search is called 'getFloor'.

In order to speed up binary search with tabulation we have created a 'Tabluation' class. The key changes that have been made in this implementation comparing to the regular binary search is an array of Nodes (Nodes is an object created by us for the purpose of this task) of size 2^k. Each entry in the array represents the first k bits of a potential query integer y and can store a node whose left and right index reference other entries in the array. The indices indicate the range of the array that the algorithm should start the binary search from. The 'table' array is filled up by calling the 'precompute' method. This method iterates over the input array, which has been sorted, and once two of its entries differ in their top k bits, sets the Nodes in the 'table' array. If the table array does not have an entry for the query integer y, then the 'findPreceedingBucket' method searches for a preceeding bucket that the algorithm will start the binary search method from.

1.2 Task 2

For task 2 we designed two different experiments. Both experiments were run on a MacBook Pro with a i5-5257U CPU, with a nominal frequency of 2.70GHz. The results can be found in part1/results. The first experiment was designed with the goal to determine the average running time per query as a function of n. Hence, we chose n as the factor for this experiment ($levels$=100000, 250000, 500000, 750000, 1000000,..., 5000000). Parameters k and q were fixed to 10 and 9.000.000, respectively. The input (S and q) was generated using Java's *Random* class and contained positive and distinct elements only. To account for the influence of randomness in the experiment, 10 trials were run per design point. For every trial the average running time per query (avg=running time/q) was calculated and the mean of the sum of indicators was used to plot. To ensure reproducible we fixed the seed from the outside to $seed$=4321. The results of this experiment are depicted in Figure 1.

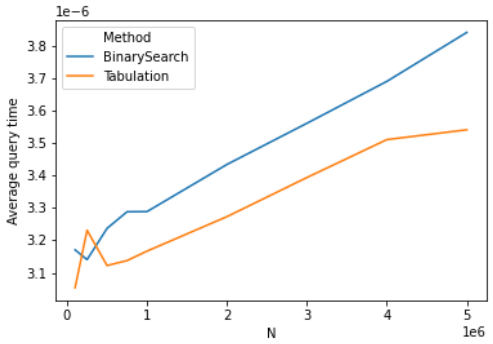

Figure 1: average running time/query as a function of n

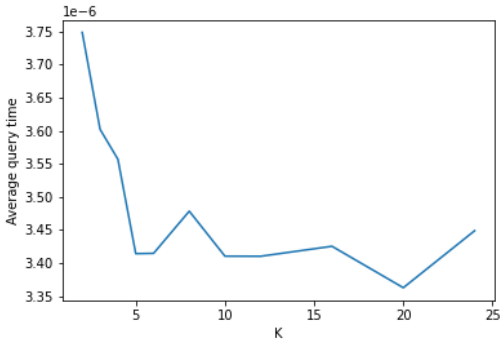

Figure 2: average running time/query as a function of k

The experiment was run over a range of small and large n in order to show on what input sizes both algorithms perform best. Our hypothesis was that Binary Search with tabulation would outperform the classic Binary Search algorithm on large input sizes due to the table being filled out more evenly. In fact, the results show that the average running time per query is substantially less for the second implementation while for smaller input sizes, the classic Binary Search implementation performs better.

The second experiment was designed with the goal to show the relation between the parameter k and average running time per query. Hence, the same experiment set up was used but parameters and q were fixed and k was chosen as the factor. To ensure the reproductibility, the seed was again fixed from the outside. The results of this experiment are depicted in Figure 2. It can be observed that a larger k reduces the average running time per query. For n=2.000.000 and q=5.000.000, k=20 performed the best. Overall, it can be said that Binary Search with tabulation can outperform the classic implementation if the input is large and, most importantly is evenly distributed. This means, that integers should frequently differ in their top k bits in order to ensure that the *table* is filled out as much as possible.

2 Part 2

2.1 Task 3

The three implementations for QuickSort can be found under the following path: part2/QuickSort1P, part2/QuickSort2P and part2/QuickSort3P respectively. The implementations are based on the pseudocode presented in the papers "Average Case Analysis of Java 7's Dual Pivot Quicksort" by Wild and Nebel [WN12](Algorithm 1 and 3), and "Multi-Pivot Quicksort: Theory and Experiments" from Kushagra et al. [K+13](Algorithm A.1.1). From a design perspective, each of the algorithms class extends the QuickSort.java class. In this superclass we implement the sort(int[] arr, int i, int n) method, which sorts the array using the Arrays.sort library.

2.2 Task 4

In order to test the correctness of our implementations for QuickSort1P, QuickSort2P, QuickSort3P we followed the below steps: 1) Created the ArrayGenerator class; with the help of this class we generate 5 different inputs for an input array: array with small values, array with large values,

array with elements in increasing order, elements in decreasing order and array containing n equal elements. 2) In the Task4.java class, for each of the input type, we compare the result of the Arrays.sort with the results given by sorting the array with the QuickSort1P, QuickSort2P and QuickSort3P respectively.

We run this test with an array of size 10999 elements. The maximum array size that we tried and works is 13999. A larger number would give an StackOverflow error. This value is for the experiment executed on a Lenovo Yoga with a i3-3217U CPU, with a nominal frequency of 1.80GHz.

Discussion on the bound check for Algorithm 3 and A.1.1 In Algorithm 3 from[WN12] if we remove the bound check in Line 11, the algorithm does not work properly anymore for all the cases. One example that shows that sometimes the sorting fails is: int[] arr = 500,3,8,5,9,9,209,200,14,13,7,4,1,12,6,103,92,50; when the penultimate element(arr[n-2]) < arr[left +1] the sorting is right; otherwise it is not ordering it right and it sorts only the sub-arrays of 2/3 elements. In other words, elements are sorted around k and g, always comparing them with p, respectively q(which represent the two pivots). So l and k move together but only until k < g.Then l is left behind and it is used to put smaller elements in front of k; otherwise the order is lost and some elements are skipped even. Also, for the elements arr[k] > q, the bound check is utilized in moving the elements for keeping the order and moving higher elements between g and q.

Similar, in the Algorithm A.1.1 from [K+13] if we take out the bound check (b<=c, b is the element on index 2, c is element on n-1(penultimate element) in Line 6, the algorithm is not working properly in all the cases. When the bound check is taken out, if the array size affords it (array has more than 4 elements), sub-arrays only are sorted, if the array is, it gives "Index out of bound exception" . Another note is that, if the element on index 2 is > penultimate element(n - 2 index) there is the index out of bounds exception (an example is:int[] arr = 5,1,10,6,3,1,4,11,2,0,5). The index out of bounds appears in the last recursive call, meaning where we have the numbers < r, and it is because it gets to a value which is higher that the maximum index value of the array, hence the out of bound exception.

3 Task 5

In order to measure the performance of each algorithm we use the five different input types mentioned in Task 4. The performance of each input type was measured for different sizes of n, from 311 up to 5097504 (in the plots x axis represent the input size and y axis represent the time that it took one an algorithm to sort a certain input size). The experiments were run on MacBook Pro with a i5-5257U CPU, with a nominal frequency of 2.70GHz.

The first one called 'bigNumbers' is an array of numbers of a maximum size 7000000. We can see that for this input type, java's Quicksort implementation performs the best when the input array is more or less bigger than 25000. Second best is 3 pivots with 2 pivots and one pivot on the third and fourth place respectively. The curves fluctuate, but it can be observed that for most of the sizes, Arrays.sort() method still performs better than our implementations.

Another figure represents performance measure for a small input size which means that the algorithms were given arrays of elements smaller or equal 1000. In this we can see that two pivot Quicksort outperforms other algorithms.

The third benchmark uses randomly generated array. In this benchmark we can see that 3 piviot Quicksort is the best solution especially for the highest values.

For the sorted array (increasing order) and decreasing order array, our implementations gets StackOverflow error when the input size is more than 13999(as above also mentioned).

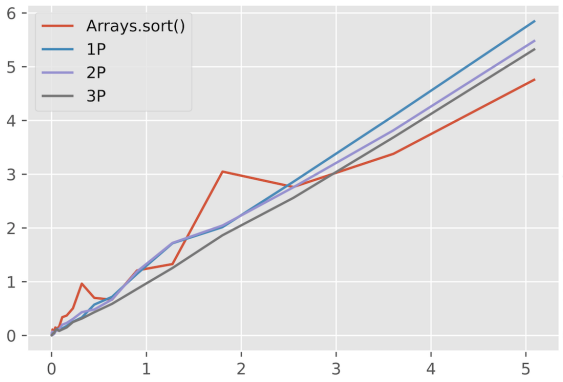

Figure 3: bigNumbers

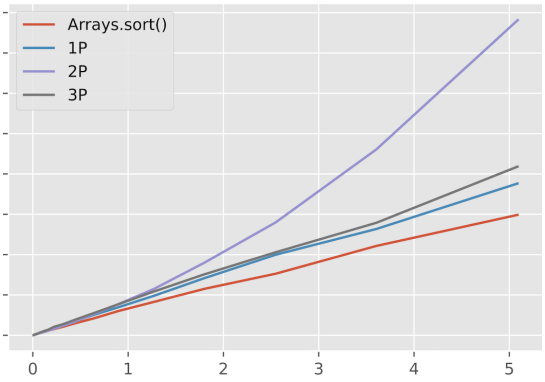

Figure 4: smallNumbers

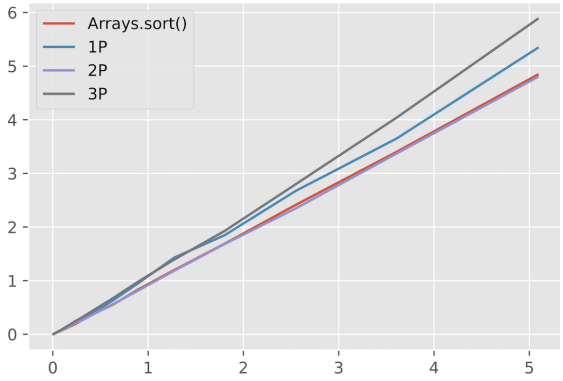

Figure 5: generateRandomArray

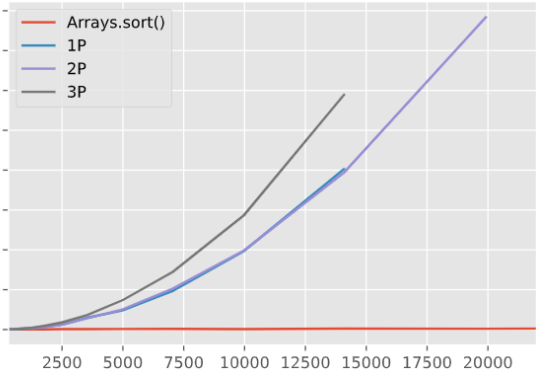

Figure 6: decreasingOrder

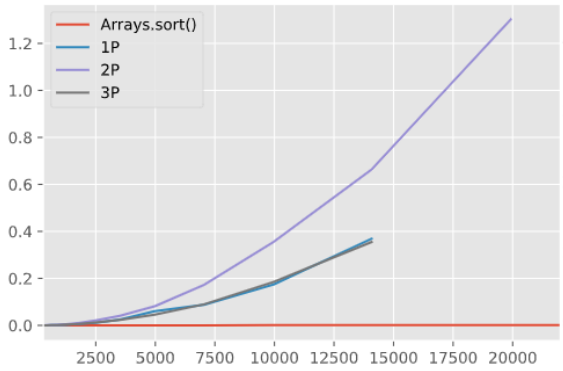

Figure 7: increasingOrder

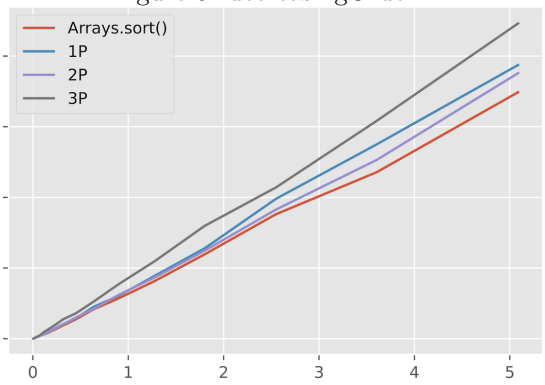

Figure 8: sameElements

Arrays.sort() remains low as java's implementations checks if the array is already sorted which saves time for calling the sort algorithm.

The last type of input is an array that consist of the same elements. The value of elements is generates in a random fashion. For this array type, the bigger the input size is the better three pivot Quicksort performance compering to the other algorithms. Second the best is one pivot implementation and two pivot and Arrays.sort() as the third and fourth place respectively.