Fatih Sultan Mehmet Vakif University
Engineering Faculty
Department of Computer Engineering

Analyses of Sorting Algorithm

Muhammet HALAK
2021221010

Berna KİRAZ
Zeliha Kaya

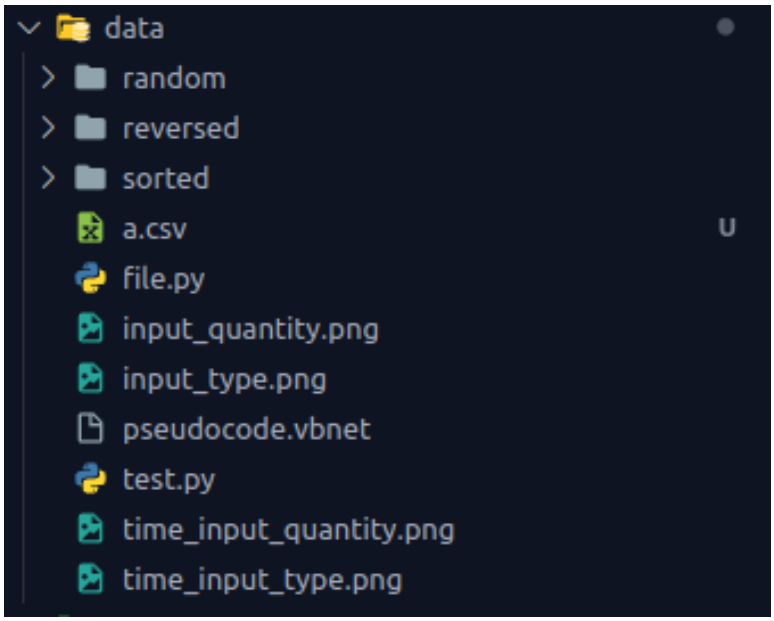
Algorithm Analysis & Design- Assignment 1

İçindekiler

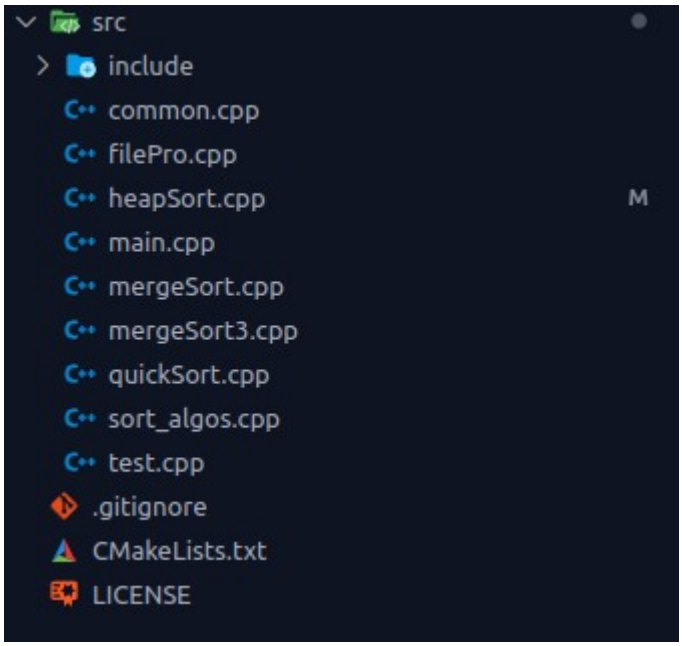
1 What is required of us:

1. **Algorithm Implementation:** Implement the following sorting algorithms in Java:
    - Selection Sort
    - Insertion Sort
    - Shellsort
    - Merge Sort
    - 3-way Merge Sort
    - Quick Sort with Lomuto & Hoare partitioning
    - Heapsort
2. **Pseudo-code and Time Complexity:** Provide pseudo-code for each algorithm and analyze their time complexities in best, worst, and average cases.
3. **Experimental Design:** Design a comprehensive experimental plan for comparing the algorithms:
    - Theoretical comparison: Evaluate their theoretical time complexities.
    - Empirical comparison: Conduct experiments with various input sizes and data distributions, measuring actual execution times.

2 Project hierarchy



Şekil 2: Project hierarchy: data



Şekil 1: Project hierarchy: src and others

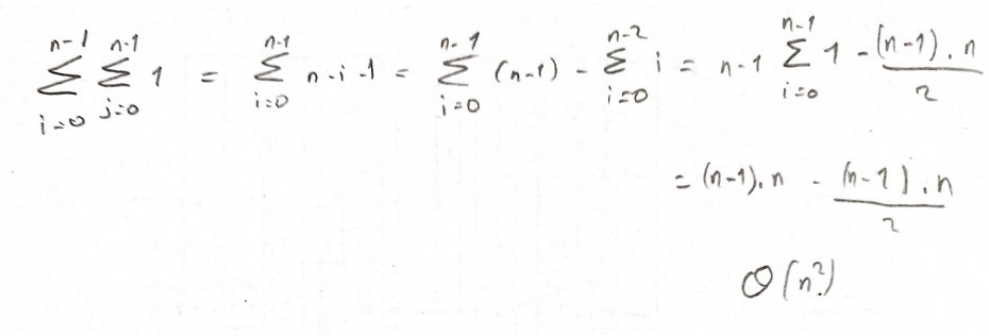
3 Analyses of Algorithms:

3.1 Selection Sort:

```
procedure selectionSort
  i_a   : array of char

  c     : integer
  count : integer
  i     : integer
  j     : integer
  min_i : integer

  c = 0
  count = size of i_a
  min_i = 0

  for i = 0 to count - 2
    min_i = i
    for j = i + 1 to count - 1
      if i_a[j] < i_a[min_i] then
        min_i = j
      end if
    end for
    c = c + 1
    swap i_a[min_i] and i_a[i]
  end for

  return c
end procedure
```

3.1.1 Mathematical analysis

W = Q(n^2) A = Q(n^2) B = Q(n^2) S = In-place B.O. = Comparession



Çizim 1: Selection Sort Time Complexity

3.1.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
1.638239	511.952.001,00	Selection Sort	random	32K_string_1
6.388741	2.047.904.001,00	Selection Sort	random	64K_string_1
25.560665	8.191.808.001,00	Selection Sort	random	128K_string_1
1.596770	511.984.000,00	Selection Sort	reversed	32K_string_1
6.394872	2.047.968.000,00	Selection Sort	reversed	64K_string_1
25.581491	8.191.936.000,00	Selection Sort	reversed	128K_string_1
1.593891	511.952.001,00	Selection Sort	sorted	32K_string_1
6.426521	2.047.904.001,00	Selection Sort	sorted	64K_string_1
25.534030	8.191.808.001,00	Selection Sort	sorted	128K_string_1

Tablo 1: Sellection Sort Analyse

3.1.3 Result

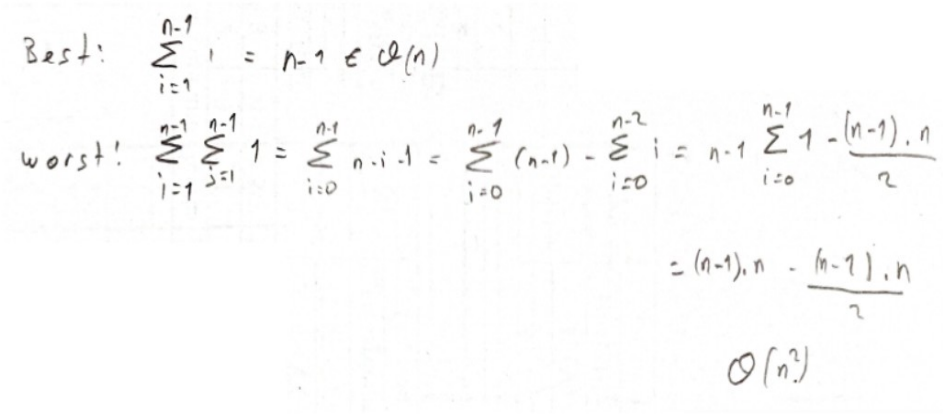
In the Selection Sort algorithm, the input type is not very important because when we look at the pseudo-code, element checking is done in all cases. Therefore, both mathematically and experimentally, the number of execution is the same.

3.2 Insertion Sort:

```
procedure insertionSort
  i_a   : array of char

  c     : integer
  count : integer
  i     : integer
  j     : integer

  c = 0
  count = size of i_a

  for i = 0 to count - 1
    for j = i to 1 step -1
      c = c + 1
      if i_a[j] > i_a[j - 1] then
        exit loop
      else
        swap i_a[j] and i_a[j - 1]
      end if
    end for
  end for

  return c
end procedure
```

3.2.1 Mathematical analysis

W = Q(n^2) A = Q(n^2) B = Q(n) S = In-place B.O. = Comparession



Çizim 2: Insertion Sort Time Complexity

3.2.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
2.726770	266.126.115,00	Insertion Sort	random	32K_string_1
10.905719	1.063.433.616,00	Insertion Sort	random	64K_string_1
43.688155	4.272.385.479,00	Insertion Sort	random	128K_string_1
5.210074	512.016.000,00	Insertion Sort	reversed	32K_string_1
20.936976	2.048.032.000,00	Insertion Sort	reversed	64K_string_1
83.518801	8.192.064.000,00	Insertion Sort	reversed	128K_string_1
0.202830	19.721.070,00	Insertion Sort	sorted	32K_string_1
0.810879	78.825.594,00	Insertion Sort	sorted	64K_string_1
3.238974	315.207.899,00	Insertion Sort	sorted	128K_string_1

Tablo 2: Insertion Sort Analyse

3.2.3 Result

Insertion Sort is generally effective for small data sets. As the data grows, the computational time and operating cycle increases. If we take an average situation, Q(n^2) becomes high. It performs the best situation in the series according to the entered data type, because comparison and replacement operations will be reduced as it compares with the previous element. In the worst case, it will occur in reverse order elements.

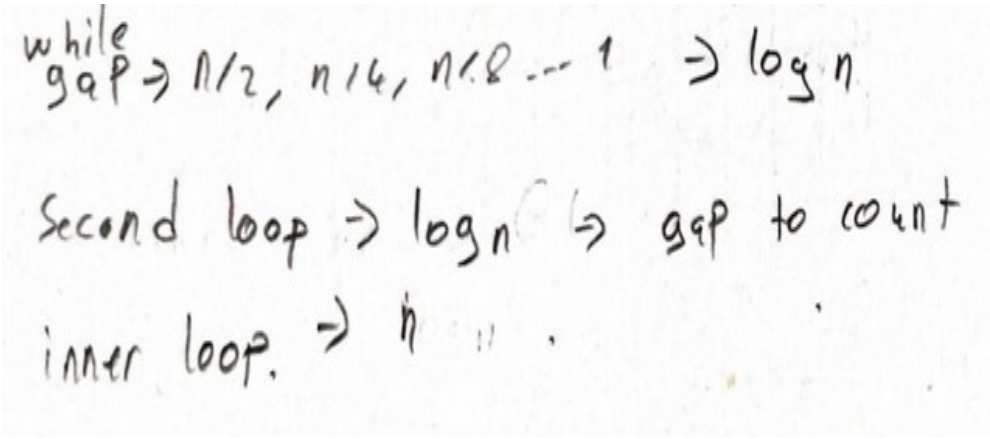
3.3 Shell Sort:

```
procedure shellSort
  i_a   : array of char

  c     : integer
  count : integer
  gap   : integer
  i     : integer
  k     : integer

  c = 0
  count = size of i_a
  gap = count / 2

  while gap > 0
    for i = gap to count - 1
      for k = i to gap step -gap
        if i_a[k - gap] > i_a[k] then
          c = c + 1
          swap i_a[k] and i_a[k - gap]
        end if
      end for
    end for
    gap = gap / 2
  end while

  return c
end procedure
```

3.3.1 Mathematical analysis

W = Q(n(log n)^2) A = Q(n(log n)^2) B = Q(n) S = In-place

B.O. = Comparession



Çizim 3: Shell Sort Time Complexity

3.3.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
0.007138	321.213,00	Shell Sort	random	32K_string_1
0.015556	755.454,00	Shell Sort	random	64K_string_1
0.035471	1.817.150,00	Shell Sort	random	128K_string_1
0.002353	89.298,00	Shell Sort	reversed	32K_string_1
0.004809	178.517,00	Shell Sort	reversed	64K_string_1
0.010147	360.229,00	Shell Sort	reversed	128K_string_1
0.001485	0,00	Shell Sort	sorted	32K_string_1
0.003184	0,00	Shell Sort	sorted	64K_string_1
0.006867	0,00	Shell Sort	sorted	128K_string_1

Tablo 3: Shell Sort Analyse

3.3.3 Result

Shell Sort runtime may vary depending on the data distribution type and data size. Working performance, especially in sequential arrays, is very good according to experimental results and calculations. But when the data became random, the time and amount of work increased. The 'sequential' 0 times operation values that appear in the table show that way because they do not enter the innermost 'for' loop, that is, the base operation part. Like many other algorithms, it grows exponentially in the worst case as the data set grows.

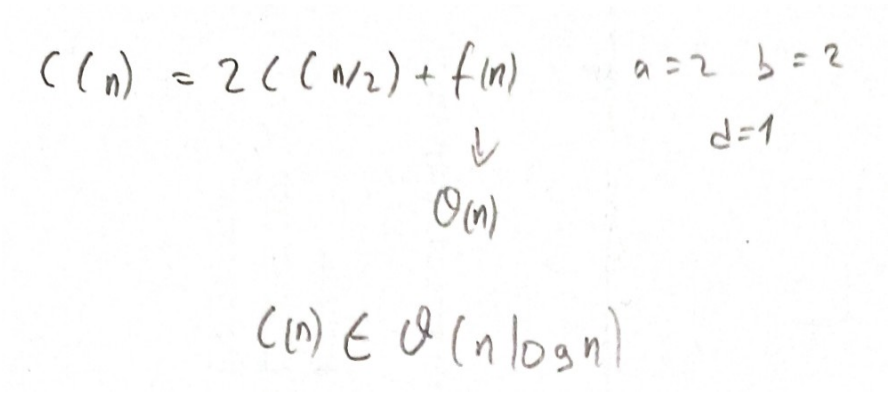
3.4 Merge Sort:

```
procedure mergeSort
   whole  : array of char

   if size of whole is 1
      return whole
   else
      left = slice of whole from 0 to size of whole / 2
      right = slice of whole from size of whole / 2 to end

      left = mergeSort(left)
      right = mergeSort(right)

      merge(left, right, whole)

      return whole
   end if
end procedure

procedure merge
   left   : array of char
   right  : array of char
   result : array of char

   x     : integer,  y     : integer,  k     : integer

   x = 0,   y = 0,   k = 0

   while x < size of left and y < size of right
      if left[x] < right[y] then
         result[k] = left[x]
         x = x + 1
      else
         result[k] = right[y]
         y = y + 1
      end if
      k = k + 1
   end while

   rest = empty array of char
   restIndex = 0

   if x >= size of left then
      rest = right
      restIndex = y
   else
      rest = left
      restIndex = x
   end if

   for i = restIndex to size of rest - 1
      result[k] = rest[i]
      k = k + 1
   end for
end procedure
```

3.4.1 Mathematical analysis

W = Q(n log n) A = Q(n log n) B = Q(n)

S = not In-place B.O. = Comparession



Çizim 4: Merge Sort Time Complexity

3.4.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
0.027570	433.948,00	Merge Sort	random	32K_string_1
0.055403	930.127,00	Merge Sort	random	64K_string_1
0.112098	1.986.681,00	Merge Sort	random	128K_string_1
0.026160	241.430,00	Merge Sort	reversed	32K_string_1
0.052705	514.840,00	Merge Sort	reversed	64K_string_1
0.106385	1.093.658,00	Merge Sort	reversed	128K_string_1
0.025859	273.513,00	Merge Sort	sorted	32K_string_1
0.052183	572.955,00	Merge Sort	sorted	64K_string_1
0.105094	1.215.778,00	Merge Sort	sorted	128K_string_1

Tablo 4: Merge Sort Analyse

3.4.3 Result

Depending on the distribution type of the data, the running time of Merge Sort algorithm differs from previous algorithms. When we look at the pseudo-code: In the merge sort algorithm, it divides it into two and then sorts it. Therefore, we obtain an approximately similar result for ordered or reverse-ordered arrays. But in a random series, as the number of comparisons increases, the difference becomes larger.

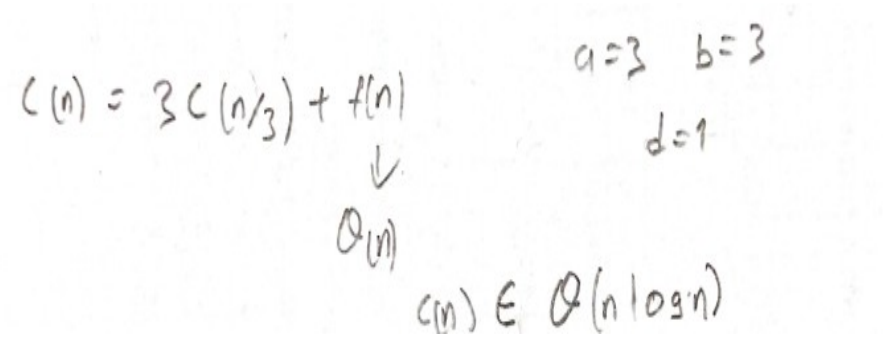
3.4.1 3 Way Merge Sort:

```
procedure mergeSort
   liste : array of char

   return sort(liste, 1, size of liste)
end procedure

procedure sort
   liste : array of char, start : integer,  end   : integer

   if end - start < 1
      return liste
   else
      mid1 = start + ((end - start) / 3)
      mid2 = start + 2 * ((end - start) / 3)

      sort(liste, start, mid1)
      sort(liste, mid1 + 1, mid2)
      sort(liste, mid2 + 1, end)
      merge(liste, start, mid1, mid2, end)

      return liste
   end if
end procedure

procedure merge
   liste : array of char
   s     : integer
   mid1, mid2,  end   : integer

   left_array, mid_array,  right_array : array of char
   ind_left,  ind_mid,  ind_right   : integer

   left_array = slice of liste from s to mid1 + 1
   mid_array = slice of liste from mid1 + 1 to mid2 + 1
   right_array = slice of liste from mid2 + 1 to end + 1

   left_array.push_back(CHAR_MAX)
   mid_array.push_back(CHAR_MAX)
   right_array.push_back(CHAR_MAX)

   ind_left = 0,  ind_mid = 0,  ind_right = 0

   for i = s to end
      c = c + 1
      minimum = min(min(left_array[ind_left],
            mid_array[ind_mid]), right_array[ind_right])
      if minimum == left_array[ind_left]
         liste[i] = left_array[ind_left]
         ind_left = ind_left + 1
      else if minimum == mid_array[ind_mid]
         liste[i] = mid_array[ind_mid]
         ind_mid = ind_mid + 1
      else
         liste[i] = right_array[ind_right]
         ind_right = ind_right + 1
      end if
   end for
end procedure
```

3.4.1.1 Mathematical analysis

W = Q(n log n) A = Q(n log n) B = Q(n)

S = not In-place B.O. = Comparession



Çizim 5: 3 Way Merge Sort Time Complexity

3.4.1.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
0.027570	433.948,00	Merge Sort	random	32K_string_1
0.055403	930.127,00	Merge Sort	random	64K_string_1
0.112098	1.986.681,00	Merge Sort	random	128K_string_1
0.026160	241.430,00	Merge Sort	reversed	32K_string_1
0.052705	514.840,00	Merge Sort	reversed	64K_string_1
0.106385	1.093.658,00	Merge Sort	reversed	128K_string_1
0.025859	273.513,00	Merge Sort	sorted	32K_string_1
0.052183	572.955,00	Merge Sort	sorted	64K_string_1
0.105094	1.215.778,00	Merge Sort	sorted	128K_string_1

Tablo 5: Merge Sort Analyse

time	times_worked	type	input_type	input_quantity
0.061803	313.872,00	Way 3 Merge Sort	random	32K_string_1
0.198495	674.476,00	Way 3 Merge Sort	random	64K_string_1
0.718163	1.437.615,00	Way 3 Merge Sort	random	128K_string_1
0.060746	313.883,00	Way 3 Merge Sort	reversed	32K_string_1
0.196034	674.486,00	Way 3 Merge Sort	reversed	64K_string_1
0.712980	1.437.628,00	Way 3 Merge Sort	reversed	128K_string_1
0.060598	313.872,00	Way 3 Merge Sort	sorted	32K_string_1
0.200082	674.476,00	Way 3 Merge Sort	sorted	64K_string_1
0.707165	1.437.615,00	Way 3 Merge Sort	sorted	128K_string_1

Tablo 6: 3 Way Merge Sort Analyse

3.4.1.3 Result

The 3 Way Merge Sort algorithm requires less processing than the normal merge sort algorithm, but dividing 3 times and then merging has a negative impact on time. But when the data grows, on the contrary, the number of transactions increases and the elapsed time decreases. In terms of real time complexity, the logarithm for normal is base 2, while for 3 Way Merge Sort the logarithm is base 3.

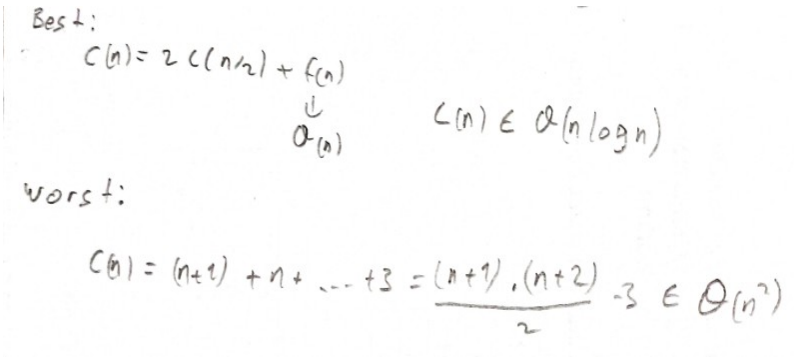
3.5 Quick Sort:

```
procedure quickSort
  liste   : array of char
  isHoare : boolean

  c = 0
  if isHoare
    quickSortHoare(liste, 0, size of liste - 1)
  else
    quickSortLomuto(liste, 0, size of liste - 1)
end procedure

procedure quickSortHoare
  liste : array of char
  low   : integer
  high  : integer

  if low < high
    pi = partitionHoare(liste, low, high)
    quickSortHoare(liste, low, pi)
    quickSortHoare(liste, pi + 1, high)
  end if
end procedure

function partitionHoare
  liste : array of char
  low   : integer
  high  : integer

  pivot = liste[low]
  i = low - 1
  j = high + 1

  while true
    repeat
      c = c + 1
      i = i + 1
    until liste[i] >= pivot

    repeat
      j = j - 1
    until liste[j] <= pivot

    if i >= j
      return j

    swap liste[i] and liste[j]
  end while
end function
```

3.5.1 Mathematical analysis

W = Q(n^2) A = Q(n log n) B = Q(n log n)

S = In-place B.O. = Comparession



Çizim 6: Qoick Sort Hoare and Lomuto Time Complexity

3.5.2 Experimental analysis

time	times_worked	type	input_type	input_quantity
0.003664	290.895,00	Quick Sort Hoare	random	32K_string_1
0.007464	600.212,00	Quick Sort Hoare	random	64K_string_1
0.015697	1.339.904,00	Quick Sort Hoare	random	128K_string_1
0.006054	742.107,00	Quick Sort Hoare	reversed	32K_string_1
0.013761	2.053.996,00	Quick Sort Hoare	reversed	64K_string_1
0.027355	2.595.505,00	Quick Sort Hoare	reversed	128K_string_1
0.009535	190.465,00	Quick Sort Hoare	sorted	32K_string_1
0.020747	412.857,00	Quick Sort Hoare	sorted	64K_string_1
0.045475	889.471,00	Quick Sort Hoare	sorted	128K_string_1

Tablo 7: Quick Sort Hoare Analyse

time	times_worked	type	input_type	input_quantity
0.168219	19.859.346,00	Quick Sort Lomuto	random	32K_string_1
0.670502	79.117.970,00	Quick Sort Lomuto	random	64K_string_1
2.665851	315.773.255,00	Quick Sort Lomuto	random	128K_string_1
0.843411	99.958.692,00	Quick Sort Lomuto	reversed	32K_string_1
6.947305	822.862.843,00	Quick Sort Lomuto	reversed	64K_string_1
7.594092	899.488.732,00	Quick Sort Lomuto	reversed	128K_string_1
4.325110	511.984.000,00	Quick Sort Lomuto	sorted	32K_string_1
17.315804	2.047.968.000,00	Quick Sort Lomuto	sorted	64K_string_1
69.310656	8.191.936.000,00	Quick Sort Lomuto	sorted	128K_string_1

Tablo 8: Quick Sort Loumuto Analyse

3.5.3 Result

The important thing in the Quicksort algorithm is to select the pivot element. If the pivot element is in the right place, the elements will be distributed more regularly to the right or left. If the pivot is in the worst place every time, the entire distribution will be one-way, which will cause problems when sorting. Of the Lomuto and Hoare techniques developed for this purpose, Hoare is more effective in sorted arrays due

to its pivot selection process. but Lomuto is more effective, especially in random situations and, conversely, in sequential situations.

```
procedure quickSort
  liste   : array of char
  isHoare : boolean

  c = 0
  if isHoare
    quickSortHoare(liste, 0, size of liste - 1)
  else
    quickSortLomuto(liste, 0, size of liste - 1)
end procedure

procedure quickSortLomuto
  liste : array of char
  low   : integer
  high  : integer

  if low < high
    pi = partitionLomuto(liste, low, high)
    quickSortLomuto(liste, low, pi - 1)
    quickSortLomuto(liste, pi + 1, high)
  end if
end procedure
```

```
function partitionLomuto
  liste : array of char
  low   : integer
  high  : integer

  pivot = liste[high]
  i = low - 1

  for j = low to high - 1
    c = c + 1
    if liste[j] <= pivot
      i = i + 1
      swap liste[i] and liste[j]
    end if
  end for

  swap liste[i + 1] and liste[high]
  return i + 1
end function
```

3.6 Heap Sort:

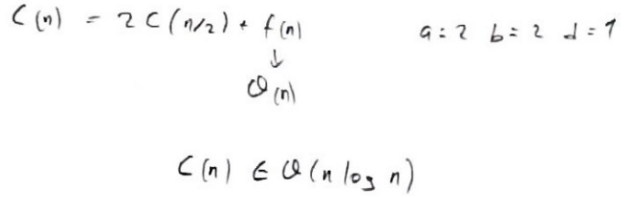
```
procedure heapify
  liste : array of char
  N     : integer
  i     : integer

  c = c + 1
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2

  if l < N and liste[l] > liste[largest]
    largest = l

  if r < N and liste[r] > liste[largest]
    largest = r

  if largest is not equal to i
    swap liste[i] and liste[largest]
    heapify(liste, N, largest)
  end if
end procedure

procedure heapSort
  liste : array of char

  s = size of liste

  for i = s / 2 - 1 down to 0
    heapify(liste, s, i)

  for i = s - 1 down to 1
    swap liste[0] and liste[i]
    heapify(liste, i, 0)
  end for
end procedure
```

3.6.1 Mathematical analysis

W = Q(n log n) A = Q(n log n) B = Q(n log n)

S = In-place B.O. = Comparession

3.6.2 Experimental analysis



1Çizim 7: Heap Sort Time Complexity

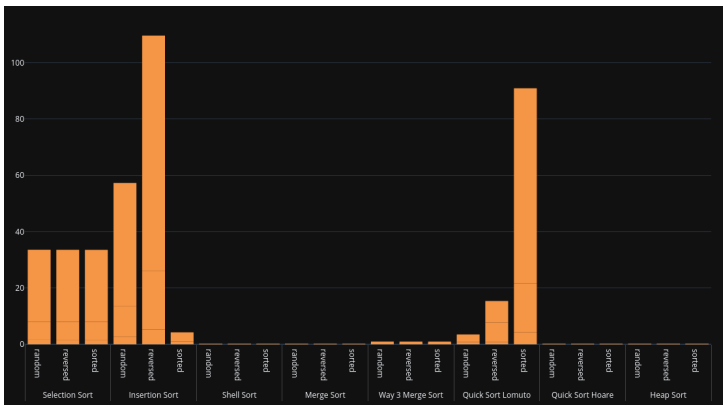
time	times_worked	type	input_type	input_quantity
0.008971	450.948,00	Heap Sort	random	32K_string_1
0.018220	962.731,00	Heap Sort	random	64K_string_1
0.038559	2.047.085,00	Heap Sort	random	128K_string_1
0.007587	421.410,00	Heap Sort	reversed	32K_string_1
0.016231	905.717,00	Heap Sort	reversed	64K_string_1
0.034552	1.930.506,00	Heap Sort	reversed	128K_string_1
0.007285	412.271,00	Heap Sort	sorted	32K_string_1
0.015488	877.908,00	Heap Sort	sorted	64K_string_1
0.033693	1.863.974,00	Heap Sort	sorted	128K_string_1

Tablo 10: Merge Sort Analyse

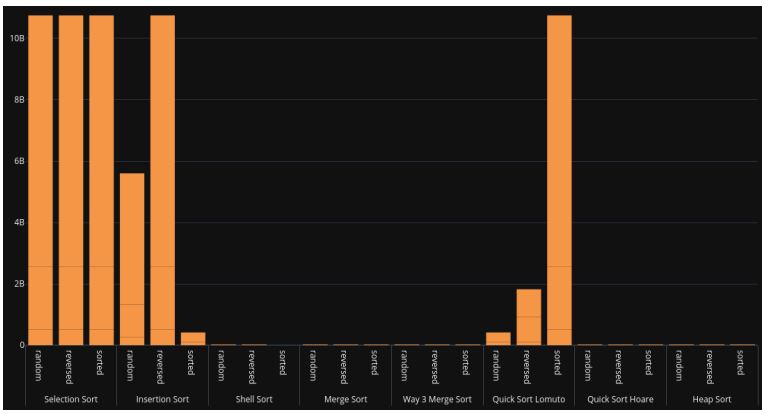
3.6.3 Result

What remains from last year about the Heap Sort algorithm is that it is an algorithm used in so many operating systems. Now, after seeing the comparison and logarithmic time complexity, I understand it better. It was also effective that the ranking was stable. When we compare it with many other algorithms, it is generally faster in terms of speed. This is due to the fact that he keeps the series in a certain order every time.

4 General Comparison

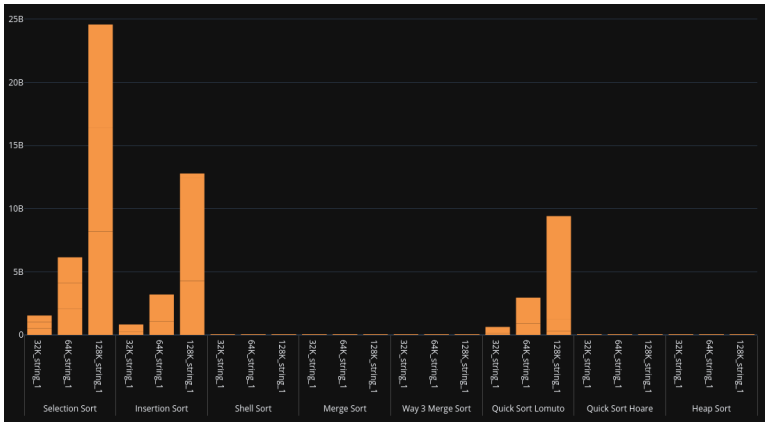


Şekil 3: Input Type - Execution Time

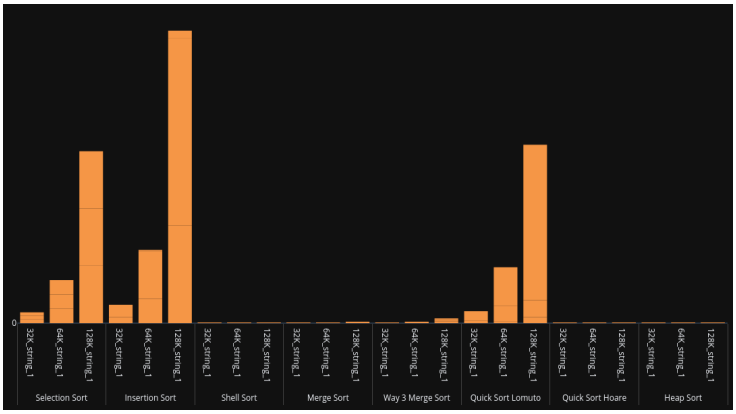


Şekil 4: Input Type - Basic Operation Count

As can be seen in the graphs above, in some sorting algorithms, the initial sorting of the data affects both the runtime and the running time. Some algorithms that we expect good performance in a sorted array (Quick Sort Lomuto) may, on the contrary, lose even more performance.



Şekil 5: Input Size - Execution Time



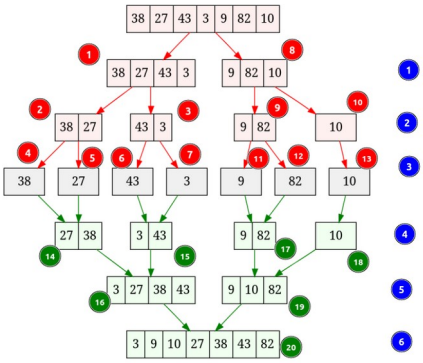
Şekil 6: Input Size - Basic Operation Count

We can see that the amount of data entered increases exponentially depending on the time complexity of the algorithm, and how this happens depending on the type of algorithm.

5 Additionally, I would like to add

- It was taking a very long time to run the algorithms. While I was thinking about how I could solve this in a better way, parallelization came to my mind.
- When we compile the program, if we set the 'omp_set_num_threads(n)' value according to the core value of the computer, we can save time by sending each algorithm to another core instead of running each algorithm on a single core.
- In terms of gain, I can say this: While it takes approximately 10 minutes and 30 seconds in the normal process, when we apply parallel processing, the time decreases to 4 minutes and 8 seconds.

- Normally, this method (using multi core) and adding threads can provide a huge performance gain, especially in 'divide and conquer' algorithms.
- Because although theoretically adding threads to a single core does not seem to provide any gain, it may provide gain as it will increase the clock speed of the processor and increase the resources used.
- For example, if you add this partitioning technique to the Merge Sort algorithm and perform partitioning and merging, it will be as if you were merging and merging at the same time.
- If I had time, I would like to try this in an algorithm.



Şekil 7: Merge Sort(i added colorful number)

- Since the algorithm was developed in C++, not Java, you can compile and test it with CMake, which is specific to the operating system, instead of Java's slowness in the virtual environment.

- I used some external resources for developing for this project. You can find it in resources.

6 Resources:

https://bilgisayarkavramlari.com/2008/08/09/siralama-algoritmalari-sorting-algorithms/

https://bilgisayarkavramlari.com/2008/12/20/kabuk-siralama-shell-sort/

https://www.geeksforgeeks.org/heap-sort/

https://gist.github.com/Akohrr/4dfd2cd4df43489269a30abce9044120

https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/

https://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg