Comparison of Efficiency of Sorting Algorithms

Mateusz Tabaszewski 151945

Poznań, March 2022

Introduction and Methods of Research

This paper is divided into 2 parts. First analyses and compares the speeds of Bubble Sort, Heap Sort, Counting Sort and Shell Sort(implemented with step h'=(3h-1) for randomly generated data. All algorithms have been implemented from scratch using C++ language(source code included in the document). Time of sorting for each n(number of elements to be sorted in an array) has been measured using GetProcessTimes(), GetCurrentProcess() functions included via <processthreadapi.h>* header. This function has been chosen due to its high accuracy, being able to calculate with accuracy of up to 100 nanoseconds**. Time has been measured for each algorithm for each n and for 50 times the generated table of size n and for 10 runs of the same algorithm in order to minimize the outside interference on time of calculations(interruptions, etc.). After each time measurement the number was divided by number of repetitions of an array and number of runs of algorithm(in order to get the average time for a run of algorithm for a given n-sized array) and saved to a text file. When generating multiple n sized arrays the program used rand() function. When an array of size n is generated it is saved and reused for all runs of all algorithms for array of size n to make sure that the same set of data is given for each n for each algorithm.

The second program and subsequent analysis compares Quick Sort, Heap Sort and Merge Sort for different types of data: random data(generated using the same function as in previous analysis), constant value, increasing order, decreasing order, ascending-descending data and descending-ascending data. The Methods used in the second experiment were mostly the same as those used in the previous experiment with the only difference being the types of data being generated and types of algorithms being tested.

*<processthreadapi.h>- time measuring functions belonging to this library are not compatible with certain compilers because of that an executable program version has been provided to test the correctness of the programs.
**this Is theoretical accuracy as stated in documentation, real accuracy may differ.

Experiment 1

In the first experiment the program was run for n between 100 and 9000 with step equal to 100(90 measuring points) for Bubble Sort, Heap Sort, Counting Sort and Shell Sort. Points acquired after the analysis have been mapped on graphs in order to compare time of sorting to number of elements n in an array.

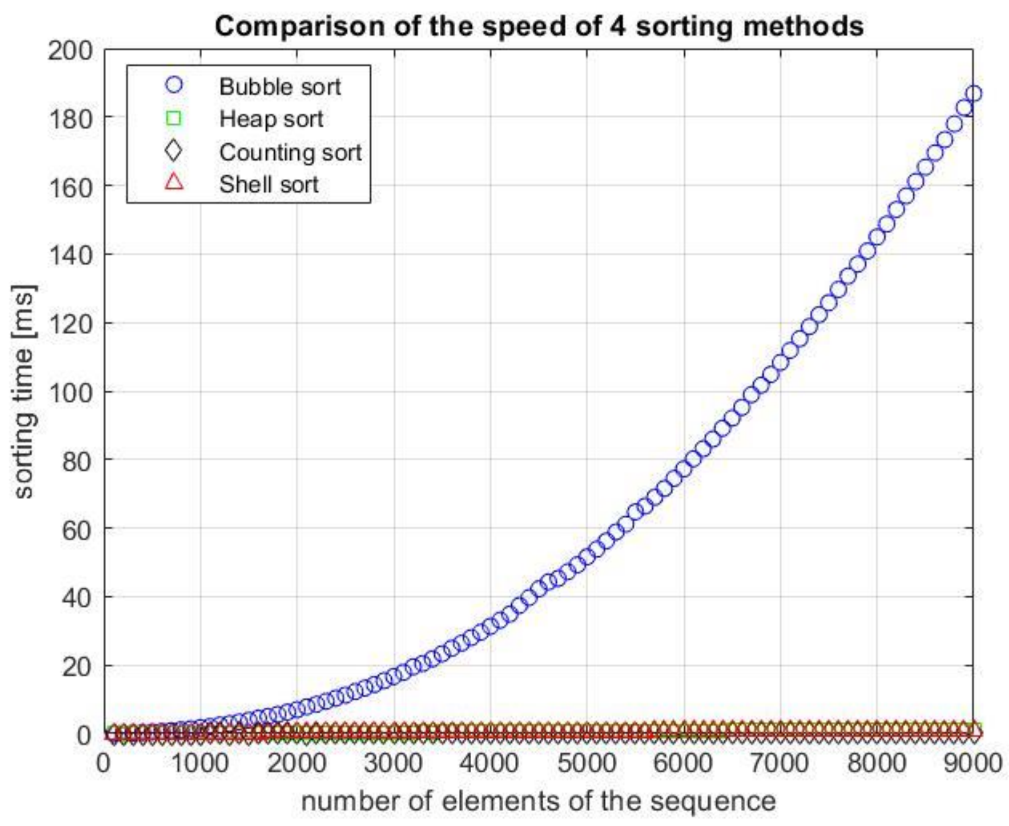

Fig. 1. Comparison of speed of 4 sorting algorithms.

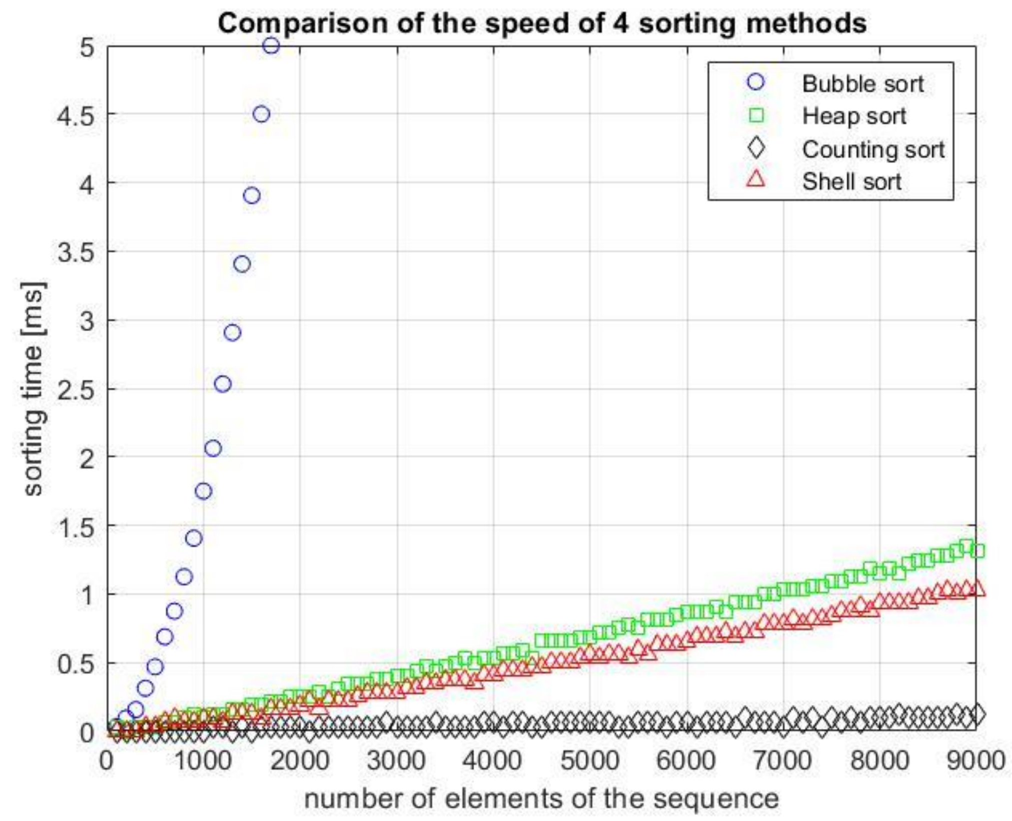

Fig. 2. Comparison of 4 sorting algorithms with Y-axis scale up to 5ms.

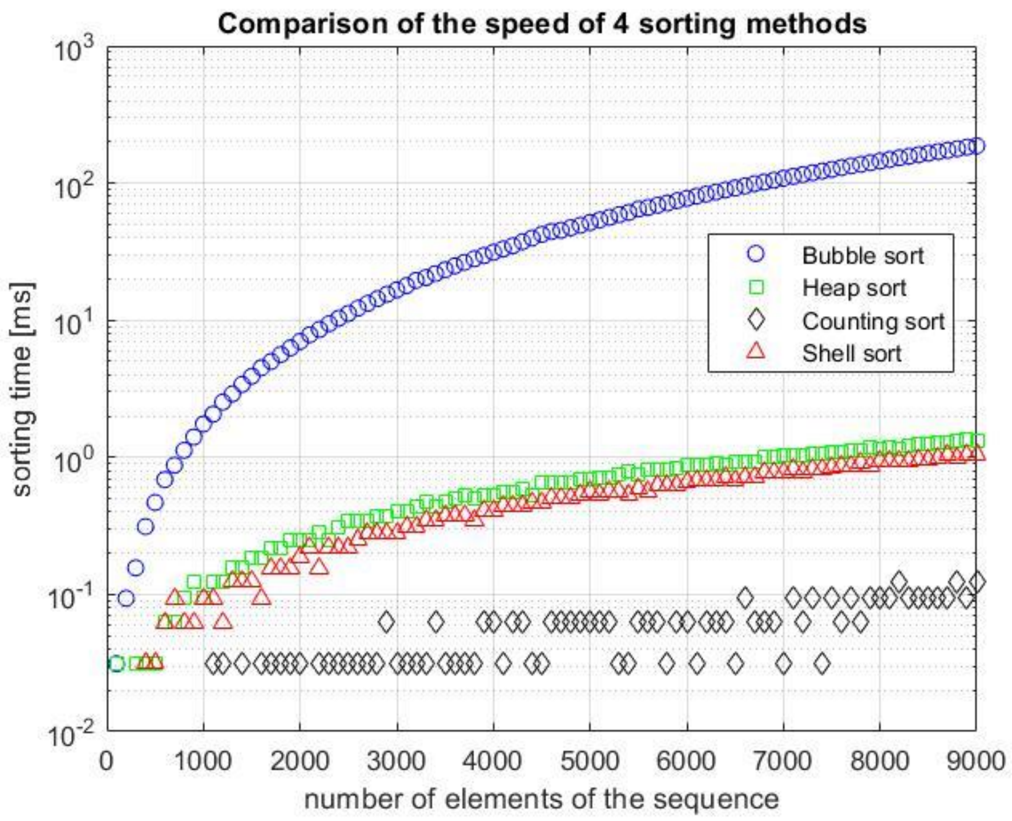

Fig. 3. Comparison of 4 sorting algorithms in semi-logarithmic scale.

Observations:

Based on presented figures it is quite obvious that the complexity(time necessary to sort an array) grows much quicker for Bubble Sort than it does for the remaining algorithms. Difference of complexity between Heap Sort and Shell Sort is the smallest between all tested algorithms. Counting Sort remains the most time-efficient algorithm out of the ones tested with increase being hard to notice on figures 1 and 2 and only being visible on figure 3.

Conclusions:

Based on Figures 1-3 the time complexity of Bubble Sort is $O(n^2)$ because the algorithm is forced to compare adjacent pairs. The number of swaps In the algorithm itself depends on number of inversion pairs, which is also more likely to be higher in an array with more elements. This sorting method requires only one additional memory space, required for temporary variable, meaning that sorting is “in place”. Heap Sort exhibits time complexity of O(nlog(n)) and sorting is of “in place” kind. Shell Sort outperforms Heap Sort for given data, however the time complexity of Shell Sort largely depends on the predetermined step, which may mean that the step tested in algorithms turned out to be especially beneficial for arrays of the given range*. Counting Sort achieves overall best results time-wise especially for big arrays. It is mostly caused by the fact, that counting sort has complexity O(n+k) as it runs through original array twice, and twice through array based on greatest number in the original array(one of those runs does not require comparisons, as the algorithm only adds values from based on previous elements of the array). In spite of the previous, Counting Sort is very memory-inefficient as it requires another array with the size of the greatest number in the original array. This means that for especially big numbers it may not be possible to run the algorithm on certain hardware. Counting Sort also requires integer representation of numbers in an array, which means that for sorting data that is not originally in an integer format, it would need to be assigned corresponding integer values.

In conclusion, Bubble Sort is not an optimal sorting algorithm and should not be used when sorting data. Both Shell Sort and Heap Sort have their uses, and they seem to be able to sort data in a similar time, however this depends heavily on the step of shell algorithm. Counting Sort should be used when prioritizing speed of the algorithm above all else, however doing so sacrifices the memory space of the computer, the algorithm also requires the elements of the array to be integers.

*Step of Shell Sort algorithm was set according to the recommendation found in “Struktury danych w języku C” by Adam Drozdek and Donald L. Simon. In the book authors argued that presented step was the best out of those tested in multiple publications as such the step h’=(h-1)/3 was used in the simulations.

Experiment 2

In the second experiment the goal is to assess the performance of 3 different sorting algorithms(Quick Sort, Heap Sort and Merge Sort) for 6 different types of data(random, constant, increasing, decreasing, A-shaped, V-shaped). The graphs have been created in a way to show 2 different types of data on the same figure. The algorithms and types of data have been mapped using different colors for different algorithms and different dots to visualize differences in types of data.

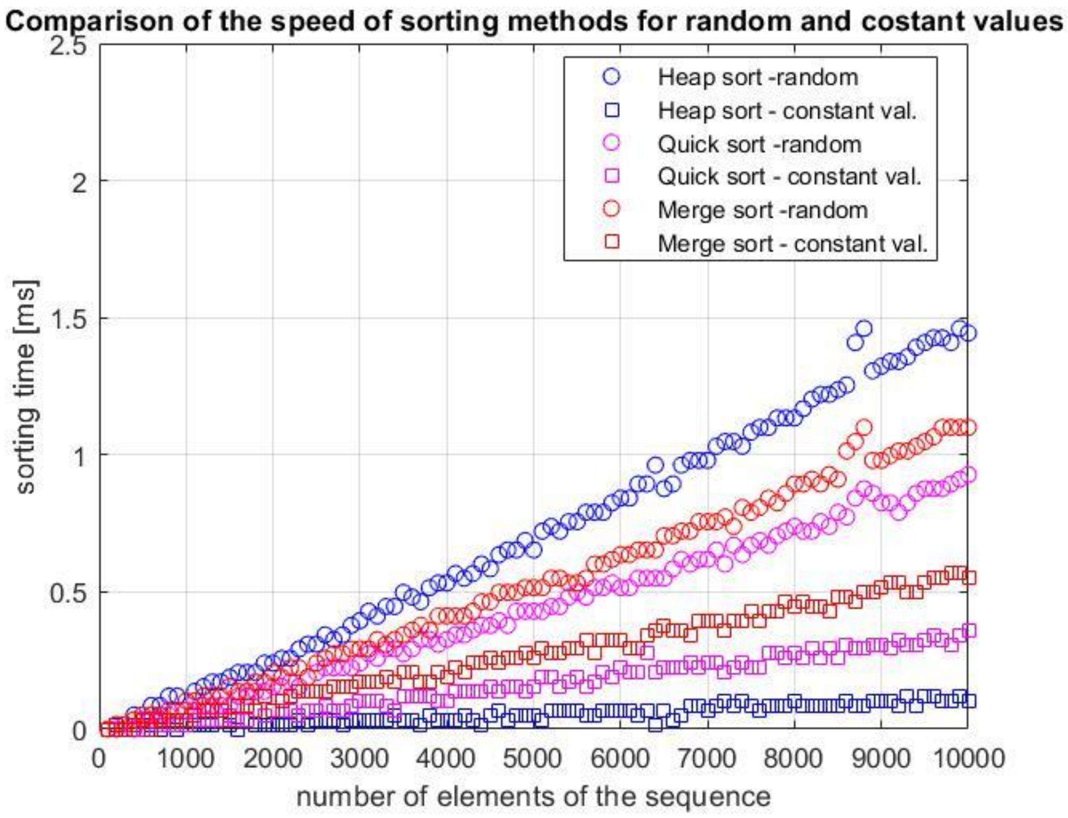

Fig. 4. Comparison of speed of sorting algorithms for random and constant values.

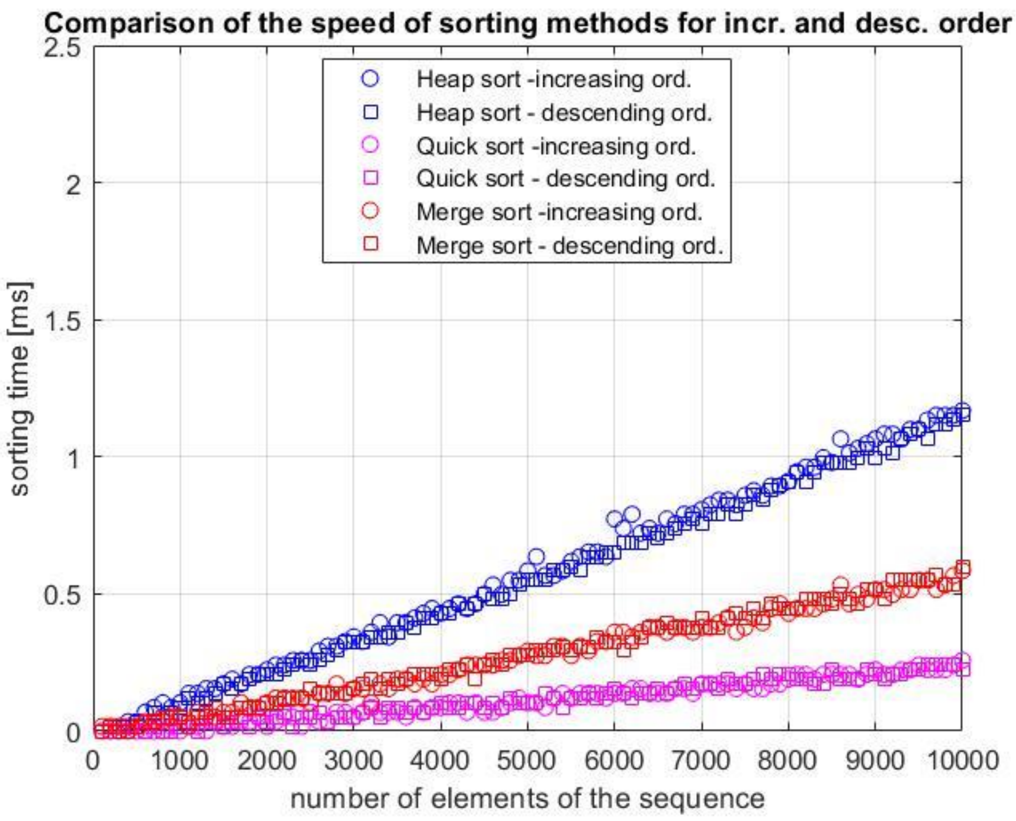

Fig. 5. Comparison of speed of sorting algorithms for strictly increasing and strictly decreasing data.

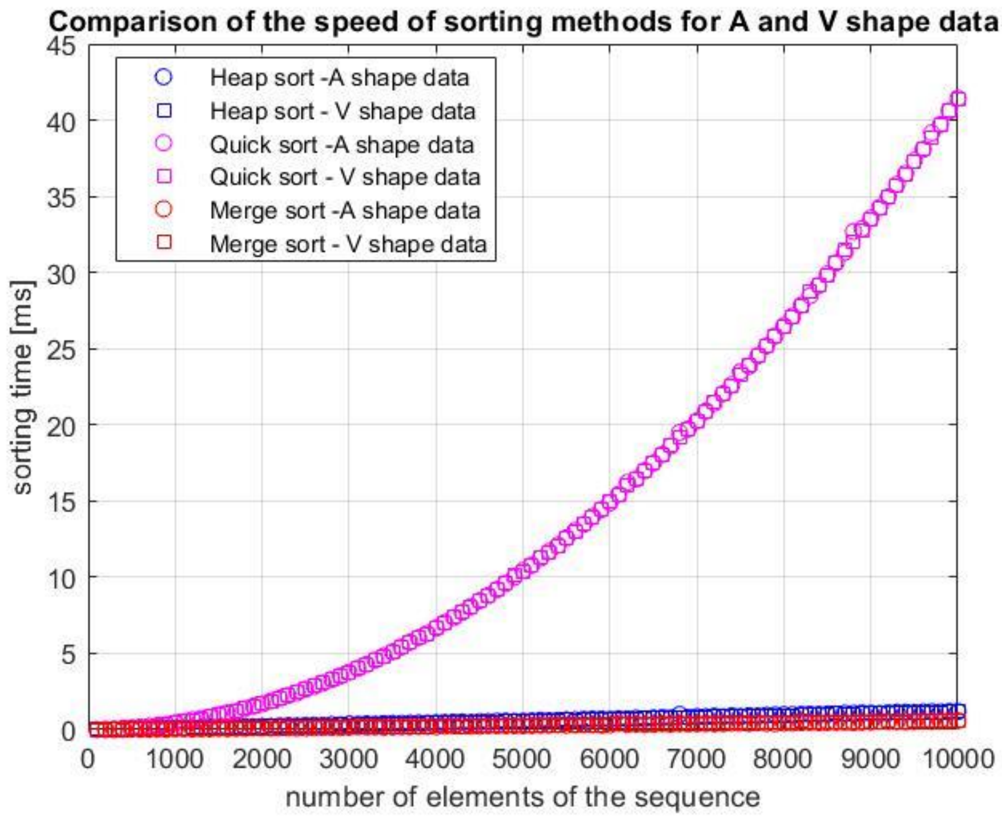

Fig. 6. Comparison of speed of sorting algorithms for A-shape and V-shape data types.

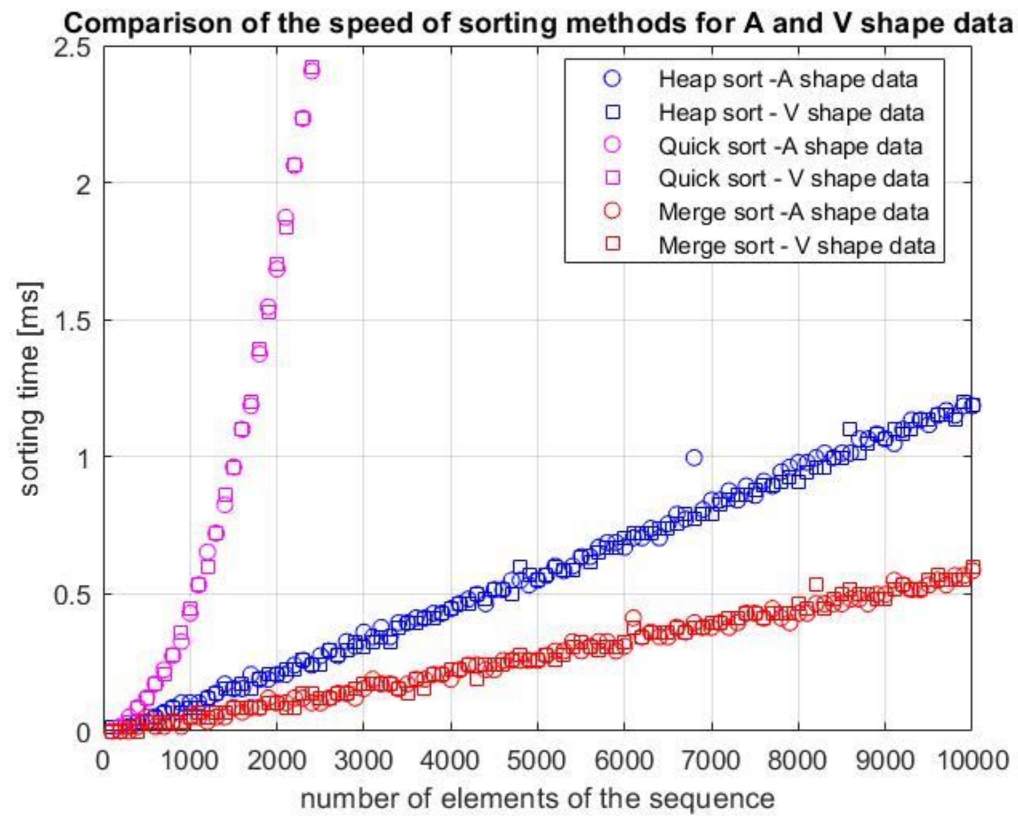

Fig. 7. Comparison of speed of sorting algorithms for A-shape and V-shape data types with Y-Axis cut off point equal to 2.5 ms.

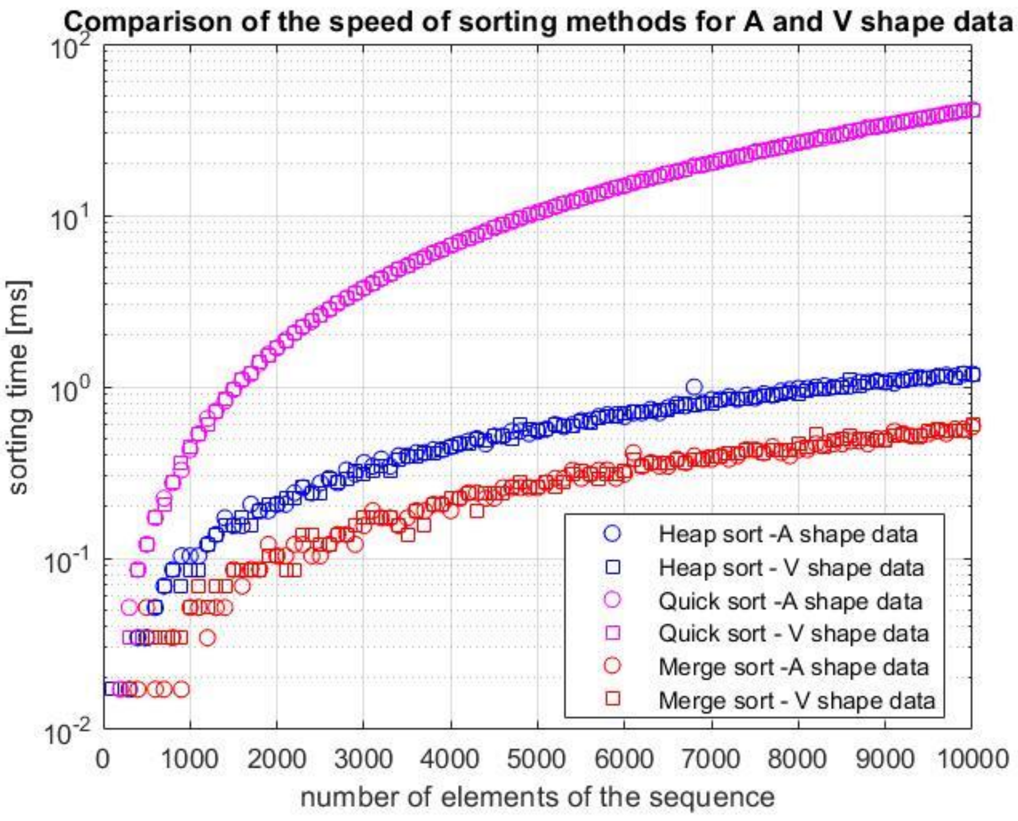

Fig. 8. Comparison of speed of sorting algorithms for A-shape and V-shape data types with semi-logarithmic scale.

Observations:

Based on Figure 4 it is easy to notice that all algorithms work faster for constant data than for random data. For random data Heap Sort is the slowest with Merge Sort being faster and Quick Sort being the fastest. However, for constant data Heap Sort is the fastest of tested algorithms, Quick Sort is the second fastest and Merge Sort is the slowest for constant values. Strictly increasing or strictly decreasing data(Figure 5) causes all of the algorithms to work faster than in case of random data. It seems to make little difference for algorithms if data is strictly increasing or strictly decreasing. Heap Sort is the slowest of presented algorithms for strictly increasing/decreasing data, Merge Sort is faster and Quick Sort is the fastest. Quick Sort works the slowest for A and V-shaped data. It's time complexity increases much faster than it does for remaining two algorithms. Merge Sort has the smallest time complexity for A and V shaped data.

Conclusions:

Quick Sort sorted the data faster than other tested sorting algorithms. Quick Sort also has the average complexity of O(nlog(n))(lie the remaining 2 tested algorithms) however it's fast sorting time may be caused by the algorithm's good cache locality(the same data is used over and over again, which means it's faster for the processor to access it for subsequent runs) on top of that Quick Sort does not require a temporary array. Although it uses recursion so stack overflow issues may occur.

For the constant values Quick Sort performed better than Merge Sort but worse than Heap Sort most likely out speeding Merge Sort thanks to the algorithm's good cache locality.

A shape and V data types turned out to be especially and equally problematic for Quick Sort algorithm as the complexity of the method rises to O(n^2). This is mostly caused by the fact that the initial pivot element will be the smallest/biggest of the numbers generated. Meaning that Quick Sort is not a suitable algorithm for A and V shaped data arrays, however certain implementations of the algorithm may help is circumventing that issue.

Choosing middle element as pivot means the possibility of worst-case scenario with complexity O(n^2), although such case is unlikely. This can be avoided by choosing median of 3 elements as pivot minimizing chances of the worst-case happening. This however slightly slows the algorithm down for all other cases other than the worst-case.

In conclusion, Quick Sort algorithm should be used when speed is of utmost importance to a project, however it is important to avoid this algorithm if there is a high risk of the worst-case scenario as that will greatly increase the time necessary to sort the array. Quick Sort should also be avoided if recursion is not possible/cannot be afforded. In such cases an alternate version of Quick Sort may be implemented or either of the two remaining algorithms may be used.

Additional Conclusions(Algorithm’s other than QS)

In the experiments for random data heap sort had took longer than remaining algorithms to sort the array of size n, despite having time complexity equal to O(nlog(n)) which is the same as time complexity for Merge Sort. This may be caused by Heap Sort’s algorithm not guarantying to leave elements with the same values unchanged, which may cause additional operations to take place. On top of that Heap Sort is also forced to perform more read and write operations while performing swaps while Merge Sort only moves data with one read and one write operation. This speed however comes at a price of memory space as Merge Sort uses an additional array.

For the constant values Merge Sort exhibits the greatest time complexity. It may be caused by the fact that the algorithm is still required to compare elements of the arrays in lists that are supposed to be merged. Although the algorithm does not require the same number of reads and writes as Heap Sort it still loses a lot of time for merging of lists. Heap Sort’s speed for constant values is mostly likely caused by the time saved by not having to exchange any elements to build the initial heap.

Per figure 7 and 8 it is noticeable that Merge Sort solves the problem faster than Heap Sort and Quick Sort. This is most likely caused by the fact that neither of the remaining algorithms suffers from the same O(n^2) problem as this implementation of Quick Sort does. On top of that Merge Sort sorts data faster than Heap Sort for reasons mentioned before, smaller number of reads and writes as well as no unnecessary swaps.

Merge Sort seems to generally out-speed Heap Sort however the sorting is not in-place which means that it should only be used when it is possible to afford the extra memory space for this algorithm. Heap Sort on the other hand can be used for all tasks which require minimal memory allocation and in cases where Quick-Sort’s worst-case cannot be afforded and is possible to happen.

Source Code for programs used in experiments:

C++ Code:

Main Code for Experiment 1

```
#include <iostream>
# include <stdlib.h>
//#include <time.h>
#include <fstream>
#include <processthreadsapi.h>

using namespace std;

void swap(int &val1, int &val2);
void bubble_sort(int arr[],int arr_size);
void selection_sort(int arr[],int arr_size);
void insertion_sort(int arr[], int arr_size);
void heap_move(int arr[], int first, int last);
void heapsort(int arr[], int arr_size);
void quicksort(int arr[], int first, int last);
void shellsort(int arr[], int arr_size);
void merge(int arr[], int temp[], int left, int center, int right);
void mergesort(int arr[], int temp[], int left, int right);
void countingsort(int arr[], int arr_size, int out_arr[]);
void print_arr(int arr[], int arr_size);
void generate_data_random(int arr[], int arr_size, int range);
void copy_arr(int arr1[], int arr2[], int arr_size);
void generate_data_constant(int arr[], int arr_size, int val);
void generate_data_increasing(int arr[],int arr_size, int step=1);
void generate_data_decreasing(int arr[],int arr_size, int step=1);
void generate_big_A(int arr[], int arr_size, int step=1);
void generate_big_V(int arr[], int arr_size, int step=1);
void generate_small_V(int arr[], int arr_size, int step=1);
void generate_small_A(int arr[], int arr_size, int step=1);
void saveToFile(int** arr,int ntables,int num_of_elem,char* name);
void readFromFile(int** arr,int num_tabl,int num_of_elem, char* name);
void isSorted(int arr[],int count);
double get_cpu_time(void);

const int NUM_TABLES=50;
const int NUM_REPS=10; //number of repetiotions for beter time measusrement (interuptions ect.)
char nameOfFile[]="data_excersie1.dat";

int main()
{
   double start=0,stop=0,elapsed=0;
         ofstream file;
   file.open("results_ex1.txt");
   int **ptrToPtr=NULL;
```

```
ptrToPtr=new int*[NUM_TABLES];
for( int kk=0;kk<NUM_TABLES;kk++)
  ptrToPtr[kk]=NULL;

int *out_arr=NULL;
for(int i=1000;i<=10000;i+=500){
        try{

  for(int kk=0;kk<NUM_TABLES;kk++)
    {
    delete []ptrToPtr[kk];
    ptrToPtr[kk]=new int[i];
    }
   }
   catch(...)
   {
    cout<<"Problem"<<endl;
                return 1;
               }
  for(int ii=0;ii<NUM_TABLES;ii++){
   generate_data_random(ptrToPtr[ii],i,i);
  }
  saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

  cout<<"************************************"<<endl;
  cout<<"Size of table "<<i<<endl<<endl;
  //bubble sort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
            readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
    {
                        bubble_sort(ptrToPtr[ii],i);
    }
    stop = get_cpu_time();
    elapsed+=stop-start;
  }
  cout<<"bubble_sort"<<endl;
  file<<"bubble_sort"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  //heap sort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
            readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
    {

                        heapsort(ptrToPtr[ii],i);
                        //isSorted(ptrToPtr[ii],i);
        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"heap_sort"<<endl;
      file<<"heap_sort"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


      //countingsort
      delete [] out_arr;
      out_arr=new int [i];

      elapsed=0;
      for (int jj=0;jj<NUM_REPS;jj++)
      {
                      readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
      start = get_cpu_time();
      for (int ii=0;ii<NUM_TABLES;ii++)
        {

                                              countingsort(ptrToPtr[ii],i,out_arr);

        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"countingsort"<<endl;
      file<<"countingsort"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;

      //Shell sort
      elapsed=0;
      for (int jj=0;jj<NUM_REPS;jj++)
      {
                      readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
      start = get_cpu_time();
      for (int ii=0;ii<NUM_TABLES;ii++)
        {
          // isSorted(ptrToPtr[ii],i);
                                              shellsort(ptrToPtr[ii],i);
                                  //  isSorted(ptrToPtr[ii],i);
        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"Shell sort"<<endl;
      file<<"Shell sort"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;

    }
    file.close();
    return 0;
}
```

```
void bubble_sort(int arr[],int arr_size){
   for(int i=0;i<arr_size;i++){
      for(int ii=arr_size-1;ii>i;ii--){
         if (arr[ii]<arr[ii-1]){
            swap(arr[ii],arr[ii-1]);
         }
      }
   }
}

void selection_sort(int arr[],int arr_size){
   int index=0;
   for(int i=0;i<arr_size;i++){
      index=i;
      for(int ii=i;ii<arr_size;ii++){
         if(arr[ii]<arr[index]){
            index=ii;
         }
      }
      swap(arr[i],arr[index]);
   }
}

void insertion_sort(int arr[], int arr_size){
   int tmp,j;
   for(int i=0;i<=arr_size;i++){
      tmp=arr[i]; j=i;
        while(j>0 && tmp<arr[j-1])
        {
            arr[j]=arr[j-1];
            j--;
         }
     arr[j]=tmp;
   }
}

void heap_move(int arr[], int first, int last){
   int largest=2*first+1;
   while(largest<=last){
      if(largest<last && arr[largest]<arr[largest+1]){
            largest++;
      }
      if(arr[first]<arr[largest]){
         swap(arr[first],arr[largest]);
         first=largest;
         largest=2*first+1;
      }
      else{
         largest=last+1;
      }
   }
}
```

```
void swap(int &val1, int &val2){
    int temp_val=val1;
    val1=val2;
    val2=temp_val;
}

void heapsort(int arr[], int arr_size){
    for(int i=arr_size/2-1;i>=0;i--){
        heap_move(arr,i,arr_size-1);
    }
    for(int j=arr_size-1;j>1;j--){
        swap(arr[0],arr[j]);
        heap_move(arr,0,j-1);
    }
    if(arr[0]>arr[1]){
        swap(arr[0],arr[1]);
    }
}

void quicksort(int arr[], int first, int last){
    int lower=first, upper=last, pivot=arr[(first+last)/2];
    while (lower<=upper){
        while(arr[lower]<pivot)
            lower++;
        while(pivot<arr[upper])
            upper--;
        if (lower<upper)
            swap(arr[lower++],arr[upper--]);
        else lower++;
    }
    if (first<upper)
        quicksort(arr,first,upper);
    if (upper+1<last)
        quicksort(arr,upper+1,last);
}

void countingsort(int arr[], int arr_size, int out_arr[]){
    int max_num=arr[0];
    int temp_val=0;
    for(int i=0;i<arr_size;i++){
        if (max_num<arr[i]){
            max_num=arr[i];
        }
    }
    int *count_arr=new int[max_num+1];
    for(int i=0;i<max_num+1;i++){
        count_arr[i]=0;
    }
    for(int i=0;i<arr_size;i++){
        count_arr[arr[i]]++;
    }
    for(int i=0;i<max_num+1;i++){
        temp_val=temp_val+count_arr[i];
        count_arr[i]=temp_val;
    }
    for(int i=0;i<arr_size;i++){
        out_arr[count_arr[arr[i]]-1]=arr[i];
        count_arr[arr[i]]--;
    }
    delete[] count_arr;
}

void shellsort(int arr[], int arr_size){
    register int i,j,h,temp,h_cnt;
    int sort_steps[30];
    int k;
    for(h=1, i=0;h<arr_size;i++){
        sort_steps[i]=h;
        h=3*h+1;
    }
    for(i--;i>=0;i--){
        h=sort_steps[i];
        for(int h_cnt=h;h_cnt<2*h;h_cnt++){
            for(int j=h_cnt;j<arr_size; ){
                temp=arr[j];
                k=j;
                while(k-h>=0 && temp<arr[k-h]){
                    arr[k]=arr[k-h];
                    k-=h;
                }
                arr[k]=temp;
                j+=h;
            }
        }
    }
}

void merge(int arr[], int temp[], int left, int center, int right){
    for(int i=left;i<=right;i++){
        temp[i]=arr[i];
    }
    int i=left, j=center+1;
    for(int k=left; k<=right; k++){
        if(i<=center){
            if(j<=right){
                if(temp[j]<temp[i])
                    arr[k]=temp[j++];
                else
                    arr[k]=temp[i++];
            }
            else
                arr[k]=temp[i++];
        }
    else
      arr[k]=temp[j++];
    }
}

void mergesort(int arr[], int temp[], int left, int right){
    if(right<=left)
      return;
    int center=(left+right)/2;
    mergesort(arr,temp,left,center);
    mergesort(arr,temp,center+1,right);

    merge(arr, temp, left, center, right);
}

void print_arr(int arr[], int arr_size){
    for(int i=0;i<arr_size;i++){
      cout<<arr[i]<<" ";
    }
    cout<<"\n";
}

void generate_data_random(int arr[], int arr_size, int range){
    for(int i=0;i<arr_size;i++){
      arr[i]=rand()%(range+1);
    }
}

void copy_arr(int arr1[], int arr2[], int arr_size){
    for(int i=0; i<arr_size; i++){
      arr2[i]=arr1[i];
    }
}

void generate_data_constant(int arr[], int arr_size, int val){
    for(int i=0;i<arr_size;i++){
      arr[i]=val;
    }
}

void generate_data_increasing(int arr[],int arr_size, int step){
    for(int i=0;i<arr_size;i++){
      if(i==0)
        arr[i]=0;
      else
        arr[i]=arr[i-1]+step;
    }
}

void generate_data_decreasing(int arr[],int arr_size, int step){
    for(int i=0;i<arr_size;i++){
      if(i==0)
```

```
            arr[i]=arr_size*step-1;
        else
            arr[i]=arr[i-1]-step-1;
    }
}

void generate_big_A(int arr[], int arr_size, int step){
    for(int i=0;i<=arr_size/2;i++){
        arr[i]=2*i*step+1;
    }
    for(int i=arr_size/2;i<arr_size;i++){
        if(i==arr_size/2)
            arr[i]=arr[i-1]-1;
        else
            arr[i]=arr[i-1]-2*step;
    }
}

void generate_big_V(int arr[], int arr_size, int step){
    int minimal;
    for(int i=0; i<=arr_size/2;i++){
        if(i==0)
            arr[i]=2*arr_size*step-1;
        else
            arr[i]=arr[i-1]-2*step;
    }
    for(int i=arr_size/2;i<arr_size;i++){
        if(i==arr_size/2)
            arr[i]=arr[i-1]+1;
        else
            arr[i]=arr[i-1]+2*step;
    }
    for(int i=0;i<arr_size;i++){
        if (i==0)
            minimal=arr[i];
        else
            if (minimal>arr[i])
                minimal=arr[i];
    }
    for(int i=0;i<arr_size;i++){
        arr[i]=arr[i]-minimal+1;
    }
}

void generate_small_V(int arr[], int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i%2==0)
            arr[i]=i*2*step;
        else
            arr[i]=(arr_size*2-1)-i*2*step;
    }
}
```

```
void generate_small_A(int arr[], int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i%2==0)
            arr[i]=(arr_size*2)-i*2*step;
        else
            arr[i]=i*2*step+1;
    }
}

void saveToFile(int** arr,int num_tabl,int num_of_elem,char* name){
    fstream file;
    file.open(name,ios::binary |ios::out);
    for (int i=0;i<num_tabl;i++)
     file.write((char*)(arr[i]),num_of_elem*sizeof(int));
    file.close();
}

void readFromFile(int** arr,int num_tabl,int num_of_elem, char* name){
   ifstream file;
    file.open(name,ios::binary );
    for (int i=0;i<num_tabl;i++)
     file.read((char*)(arr[i]),num_of_elem*sizeof(int));
    file.close();
}

void isSorted(int arr[],int count)
{
            for (int i=1;i<count;i++)
                        if(arr[i-1]>arr[i])
                         {
                          cout<<"Not sorted"<<endl;
                          return;
                 }
   cout<<"Sorted"<<endl;
}


double get_cpu_time(){
            //in ms
    FILETIME a,b,c,d;
    if (GetProcessTimes(GetCurrentProcess(),&a,&b,&c,&d) != 0){

        return
            (double)(d.dwLowDateTime |
            (((unsigned long long)d.dwHighDateTime) << 32)) * 0.0001; //ms
    }else{

        return 0;
    }}
```

```
#include <iostream>
# include <stdlib.h>
//#include <time.h>
#include <fstream>
#include <processthreadsapi.h>

using namespace std;

void swap(int &val1, int &val2);
void bubble_sort(int arr[],int arr_size);
void selection_sort(int arr[],int arr_size);
void insertion_sort(int arr[], int arr_size);
void heap_move(int arr[], int first, int last);
void heapsort(int arr[], int arr_size);
void quicksort(int arr[], int first, int last);
void shellsort(int arr[], int arr_size);
void merge(int arr[], int temp[], int left, int center, int right);
void mergesort(int arr[], int temp[], int left, int right);
void countingsort(int arr[], int arr_size, int out_arr[]);
void print_arr(int arr[], int arr_size);
void generate_data_random(int arr[], int arr_size, int range);
void copy_arr(int arr1[], int arr2[], int arr_size);
void generate_data_constant(int arr[], int arr_size, int val);
void generate_data_increasing(int arr[],int arr_size, int step=1);
void generate_data_decreasing(int arr[],int arr_size, int step=1);
void generate_big_A(int arr[], int arr_size, int step=1);
void generate_big_V(int arr[], int arr_size, int step=1);
void generate_small_V(int arr[], int arr_size, int step=1);
void generate_small_A(int arr[], int arr_size, int step=1);
void saveToFile(int** arr,int ntables,int num_of_elem,char* name);
void readFromFile(int** arr,int num_tabl,int num_of_elem, char* name);
void isSorted(int arr[],int count);
double get_cpu_time(void);

const int NUM_TABLES=50;
const int NUM_REPS=10; //number of repetitions for beter time measusrement (interuptions ect.)
char nameOfFile[]="data_excersie1.dat";

int main()
{
   double start=0,stop=0,elapsed=0;
            ofstream file;
   file.open("results_ex2.txt");
   int **ptrToPtr=NULL;
   ptrToPtr=new int*[NUM_TABLES];
   for( int kk=0;kk<NUM_TABLES;kk++)
      ptrToPtr[kk]=NULL;

   int *out_arr=NULL;
```

```
for(int i=100;i<=10000;i+=100){
        try{

  for(int kk=0;kk<NUM_TABLES;kk++)
     {
     delete []ptrToPtr[kk];
     ptrToPtr[kk]=new int[i];
     }
   }
   catch(...)
   {
    cout<<"Problem"<<endl;
                  return 1;
                  }
  for(int ii=0;ii<NUM_TABLES;ii++){
   generate_data_random(ptrToPtr[ii],i,i);
  }
  saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);



  cout<<"**********************************"<<endl;
  cout<<"Size of table "<<i<<endl<<endl;
  //quicksort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
              readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
     {
                        quicksort(ptrToPtr[ii],0,i-1);
     }
     stop = get_cpu_time();
     elapsed+=stop-start;
  }
  cout<<"quick_sort"<<endl;
  file<<"quick_sort"<<"\t"<<"random_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  //heap sort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
              readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
     {

                        heapsort(ptrToPtr[ii],i);
                        //isSorted(ptrToPtr[ii],i);
     }
     stop = get_cpu_time();
     elapsed+=stop-start;
   }
   cout<<"heap_sort"<<endl;
   file<<"heap_sort"<<"\t"<<"random_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


   //mergesort
   delete [] out_arr;
   out_arr=new int [i];

   elapsed=0;
   for (int jj=0;jj<NUM_REPS;jj++)
   {
                   readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
   start = get_cpu_time();
   for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                          mergesort(ptrToPtr[ii],out_arr,0,i-1);

      }
      stop = get_cpu_time();
      elapsed+=stop-start;
   }
   cout<<"merge_sort"<<endl;
   file<<"merge_sort"<<"\t"<<"random_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
}
//------------------------------------------------------------------------------------------------------
for(int i=100;i<=10000;i+=100){
         try{

   for(int kk=0;kk<NUM_TABLES;kk++)
      {
      delete []ptrToPtr[kk];
      ptrToPtr[kk]=new int[i];
      }
    }
    catch(...)
    {
     cout<<"Problem"<<endl;
                    return 1;
                   }
   for(int ii=0;ii<NUM_TABLES;ii++){
    generate_data_constant(ptrToPtr[ii],i,i-100);
   }
   saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

   cout<<"**********************************"<<endl;
   cout<<"Size of table "<<i<<endl<<endl;
   //quicksort
   elapsed=0;
   for (int jj=0;jj<NUM_REPS;jj++)
   {
```

```
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {
                                    quicksort(ptrToPtr[ii],0,i-1);
      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"quick_sort"<<endl;
    file<<"quick_sort"<<"\t"<<"const_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
    //heap sort
    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                    heapsort(ptrToPtr[ii],i);
                                    //isSorted(ptrToPtr[ii],i);
      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"heap_sort"<<endl;
    file<<"heap_sort"<<"\t"<<"const_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


    //mergesort
    delete [] out_arr;
    out_arr=new int [i];

    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                          mergesort(ptrToPtr[ii],out_arr,0,i-1);

      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"merge_sort"<<endl;
    file<<"merge_sort"<<"\t"<<"const_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  }
  //--------------------------------------------------------------------------------------------------------------------------------------
```

```
for(int i=100;i<=10000;i+=100){
        try{

  for(int kk=0;kk<NUM_TABLES;kk++)
     {
     delete []ptrToPtr[kk];
     ptrToPtr[kk]=new int[i];
     }
   }
   catch(...)
   {
    cout<<"Problem"<<endl;
                  return 1;
                  }
  for(int ii=0;ii<NUM_TABLES;ii++){
   generate_data_increasing(ptrToPtr[ii],i);
  }
  saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

  cout<<"*********************************"<<endl;
  cout<<"Size of table "<<i<<endl<<endl;
  //quicksort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
     {
                                quicksort(ptrToPtr[ii],0,i-1);
     }
     stop = get_cpu_time();
     elapsed+=stop-start;
  }
  cout<<"quick_sort"<<endl;
  file<<"quick_sort"<<"\t"<<"increasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  //heap sort
  elapsed=0;
  for (int jj=0;jj<NUM_REPS;jj++)
  {
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
  start = get_cpu_time();
  for (int ii=0;ii<NUM_TABLES;ii++)
     {

                                heapsort(ptrToPtr[ii],i);
                                //isSorted(ptrToPtr[ii],i);
     }
     stop = get_cpu_time();
     elapsed+=stop-start;
  }
  cout<<"heap_sort"<<endl;
```

```
    file<<"heap_sort"<<"\t"<<"increasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


    //mergesort
    delete [] out_arr;
    out_arr=new int [i];

    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                    mergesort(ptrToPtr[ii],out_arr,0,i-1);

      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"merge_sort"<<endl;
    file<<"merge_sort"<<"\t"<<"increasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  }
  //--------------------------------------------------------------------------
  for(int i=100;i<=10000;i+=100){
          try{

    for(int kk=0;kk<NUM_TABLES;kk++)
      {
      delete []ptrToPtr[kk];
      ptrToPtr[kk]=new int[i];
      }
     }
     catch(...)
     {
      cout<<"Problem"<<endl;
                    return 1;
                    }
    for(int ii=0;ii<NUM_TABLES;ii++){
     generate_data_decreasing(ptrToPtr[ii],i);
    }
    saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

    cout<<"**********************************"<<endl;
    cout<<"Size of table "<<i<<endl<<endl;
    //quicksort
    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {
                              quicksort(ptrToPtr[ii],0,i-1);
      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"quick_sort"<<endl;
    file<<"quick_sort"<<"\t"<<"decreasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
    //heap sort
    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                 readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                              heapsort(ptrToPtr[ii],i);
                              //isSorted(ptrToPtr[ii],i);
      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"heap_sort"<<endl;
    file<<"heap_sort"<<"\t"<<"decreasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


    //mergesort
    delete [] out_arr;
    out_arr=new int [i];

    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                 readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                     mergesort(ptrToPtr[ii],out_arr,0,i-1);

      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"merge_sort"<<endl;
    file<<"merge_sort"<<"\t"<<"decreasing_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  }
  //----------------------------------------------------------------------------------------------------------
  for(int i=100;i<=10000;i+=100){
        try{
```

```
for(int kk=0;kk<NUM_TABLES;kk++)
   {
   delete []ptrToPtr[kk];
   ptrToPtr[kk]=new int[i];
   }
 }
 catch(...)
 {
  cout<<"Problem"<<endl;
                    return 1;
                   }
for(int ii=0;ii<NUM_TABLES;ii++){
 generate_big_A(ptrToPtr[ii],i);
}
saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

cout<<"**********************************"<<endl;
cout<<"Size of table "<<i<<endl<<endl;
//quicksort
elapsed=0;
for (int jj=0;jj<NUM_REPS;jj++)
{
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
start = get_cpu_time();
for (int ii=0;ii<NUM_TABLES;ii++)
   {
                                quicksort(ptrToPtr[ii],0,i-1);
   }
   stop = get_cpu_time();
   elapsed+=stop-start;
}
cout<<"quick_sort"<<endl;
file<<"quick_sort"<<"\t"<<"A_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
//heap sort
elapsed=0;
for (int jj=0;jj<NUM_REPS;jj++)
{
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
start = get_cpu_time();
for (int ii=0;ii<NUM_TABLES;ii++)
   {

                                heapsort(ptrToPtr[ii],i);
                                //isSorted(ptrToPtr[ii],i);
   }
   stop = get_cpu_time();
   elapsed+=stop-start;
}
cout<<"heap_sort"<<endl;
file<<"heap_sort"<<"\t"<<"A_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
```

```
    //mergesort
    delete [] out_arr;
    out_arr=new int [i];

    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
      {

                                    mergesort(ptrToPtr[ii],out_arr,0,i-1);

      }
      stop = get_cpu_time();
      elapsed+=stop-start;
    }
    cout<<"merge_sort"<<endl;
    file<<"merge_sort"<<"\t"<<"A_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
  }
  //-----------------------------------------------------------------------------------------------------------
  for(int i=100;i<=10000;i+=100){
        try{

    for(int kk=0;kk<NUM_TABLES;kk++)
      {
      delete []ptrToPtr[kk];
      ptrToPtr[kk]=new int[i];
      }
     }
     catch(...)
     {
      cout<<"Problem"<<endl;
                    return 1;
                    }
    for(int ii=0;ii<NUM_TABLES;ii++){
     generate_big_V(ptrToPtr[ii],i);
    }

    saveToFile(ptrToPtr,NUM_TABLES,i,nameOfFile);

    cout<<"************************************"<<endl;
    cout<<"Size of table "<<i<<endl<<endl;
    //quicksort
    elapsed=0;
    for (int jj=0;jj<NUM_REPS;jj++)
    {
                    readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
    start = get_cpu_time();
    for (int ii=0;ii<NUM_TABLES;ii++)
```

```
        {
                                   quicksort(ptrToPtr[ii],0,i-1);
                                  // isSorted(ptrToPtr[ii],i);
        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"quick_sort"<<endl;
      file<<"quick_sort"<<"\t"<<"V_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
      //heap sort
      elapsed=0;
      for (int jj=0;jj<NUM_REPS;jj++)
      {
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
      start = get_cpu_time();
      for (int ii=0;ii<NUM_TABLES;ii++)
        {

                                   heapsort(ptrToPtr[ii],i);
                                   //isSorted(ptrToPtr[ii],i);
        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"heap_sort"<<endl;
      file<<"heap_sort"<<"\t"<<"V_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;


      //mergesort
      delete [] out_arr;
      out_arr=new int [i];

      elapsed=0;
      for (int jj=0;jj<NUM_REPS;jj++)
      {
                  readFromFile(ptrToPtr,NUM_TABLES,i,nameOfFile);
      start = get_cpu_time();
      for (int ii=0;ii<NUM_TABLES;ii++)
        {

                                           mergesort(ptrToPtr[ii],out_arr,0,i-1);

        }
        stop = get_cpu_time();
        elapsed+=stop-start;
      }
      cout<<"merge_sort"<<endl;
      file<<"merge_sort"<<"\t"<<"V_data"<<"\t"<<i<<"\t"<<elapsed/NUM_TABLES/NUM_REPS<<endl;
    }
    file.close();
    return 0;
}
```

```
void bubble_sort(int arr[],int arr_size){
    for(int i=0;i<arr_size;i++){
        for(int ii=arr_size-1;ii>i;ii--){
            if (arr[ii]<arr[ii-1]){
                swap(arr[ii],arr[ii-1]);
            }
        }
    }
}

void selection_sort(int arr[],int arr_size){
    int index=0;
    for(int i=0;i<arr_size;i++){
        index=i;
        for(int ii=i;ii<arr_size;ii++){
            if(arr[ii]<arr[index]){
                index=ii;
            }
        }
        swap(arr[i],arr[index]);
    }
}

void insertion_sort(int arr[], int arr_size){
    int tmp,j;
    for(int i=0;i<=arr_size;i++){
        tmp=arr[i]; j=i;
          while(j>0 && tmp<arr[j-1])
          {
                arr[j]=arr[j-1];
                j--;
           }
       arr[j]=tmp;
    }
}

void heap_move(int arr[], int first, int last){
    int largest=2*first+1;
    while(largest<=last){
        if(largest<last && arr[largest]<arr[largest+1]){
                largest++;
        }
        if(arr[first]<arr[largest]){
            swap(arr[first],arr[largest]);
            first=largest;
            largest=2*first+1;
        }
        else{
            largest=last+1;
        }
    }
```

```
}

void swap(int &val1, int &val2){
    int temp_val=val1;
    val1=val2;
    val2=temp_val;
}

void heapsort(int arr[], int arr_size){
    for(int i=arr_size/2-1;i>=0;i--){
        heap_move(arr,i,arr_size-1);
    }
    for(int j=arr_size-1;j>1;j--){
        swap(arr[0],arr[j]);
        heap_move(arr,0,j-1);
    }
    if(arr[0]>arr[1]){
        swap(arr[0],arr[1]);
    }
}

void quicksort(int arr[], int first, int last){
    int lower=first, upper=last, pivot=arr[(first+last)/2];
    while (lower<=upper){
        while(arr[lower]<pivot)
            lower++;
        while(pivot<arr[upper])
            upper--;
        if (lower<upper)
            swap(arr[lower++],arr[upper--]);
        else lower++;
    }
    if (first<upper)
        quicksort(arr,first,upper);
    if (upper+1<last)
        quicksort(arr,upper+1,last);
}

void countingsort(int arr[], int arr_size, int out_arr[]){
    int max_num=arr[0];
    int temp_val=0;
    for(int i=0;i<arr_size;i++){
        if (max_num<arr[i]){
            max_num=arr[i];
        }
    }
    int *count_arr=new int[max_num+1];
    for(int i=0;i<max_num+1;i++){
        count_arr[i]=0;
    }
    for(int i=0;i<arr_size;i++){
        count_arr[arr[i]]++;
    }
    for(int i=0;i<max_num+1;i++){
        temp_val=temp_val+count_arr[i];
        count_arr[i]=temp_val;
    }
    for(int i=0;i<arr_size;i++){
        out_arr[count_arr[arr[i]]-1]=arr[i];
        count_arr[arr[i]]--;
    }
    delete[] count_arr;
}

void shellsort(int arr[], int arr_size){
    register int i,j,h,temp,h_cnt;
    int sort_steps[30];
    int k;
    for(h=1, i=0;h<arr_size;i++){
        sort_steps[i]=h;
        h=3*h+1;
    }
    for(i--;i>=0;i--){
        h=sort_steps[i];
        for(int h_cnt=h;h_cnt<2*h;h_cnt++){
            for(int j=h_cnt;j<arr_size; ){
                temp=arr[j];
                k=j;
                while(k-h>=0 && temp<arr[k-h]){
                    arr[k]=arr[k-h];
                    k-=h;
                }
                arr[k]=temp;
                j+=h;
            }
        }
    }
}

void merge(int arr[], int temp[], int left, int center, int right){
    for(int i=left;i<=right;i++){
        temp[i]=arr[i];
    }
    int i=left, j=center+1;
    for(int k=left; k<=right; k++){
        if(i<=center){
            if(j<=right){
                if(temp[j]<temp[i])
                    arr[k]=temp[j++];
                else
                    arr[k]=temp[i++];
            }
            else
                arr[k]=temp[i++];
        }
    else
        arr[k]=temp[j++];
    }
}

void mergesort(int arr[], int temp[], int left, int right){
    if(right<=left)
        return;
    int center=(left+right)/2;
    mergesort(arr,temp,left,center);
    mergesort(arr,temp,center+1,right);

    merge(arr, temp, left, center, right);
}

void print_arr(int arr[], int arr_size){
    for(int i=0;i<arr_size;i++){
        cout<<arr[i]<<" ";
    }
    cout<<"\n";
}

void generate_data_random(int arr[], int arr_size, int range){
    for(int i=0;i<arr_size;i++){
        arr[i]=rand()%(range+1);
    }
}

void copy_arr(int arr1[], int arr2[], int arr_size){
    for(int i=0; i<arr_size; i++){
        arr2[i]=arr1[i];
    }
}

void generate_data_constant(int arr[], int arr_size, int val){
    for(int i=0;i<arr_size;i++){
        arr[i]=val;
    }
}

void generate_data_increasing(int arr[],int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i==0)
            arr[i]=0;
        else
            arr[i]=arr[i-1]+step;
    }
}

void generate_data_decreasing(int arr[],int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i==0)
            arr[i]=arr_size*step-1;
        else
            arr[i]=arr[i-1]-step-1;
    }
}

void generate_big_A(int arr[], int arr_size, int step){
    for(int i=0;i<=arr_size/2;i++){
        arr[i]=2*i*step+1;
    }
    for(int i=arr_size/2;i<arr_size;i++){
        if(i==arr_size/2)
            arr[i]=arr[i-1]-1;
        else
            arr[i]=arr[i-1]-2*step;
    }
}

void generate_big_V(int arr[], int arr_size, int step){
    int minimal;
    for(int i=0; i<=arr_size/2;i++){
        if(i==0)
            arr[i]=2*arr_size*step-1;
        else
            arr[i]=arr[i-1]-2*step;
    }
    for(int i=arr_size/2;i<arr_size;i++){
        if(i==arr_size/2)
            arr[i]=arr[i-1]+1;
        else
            arr[i]=arr[i-1]+2*step;
    }
    for(int i=0;i<arr_size;i++){
        if (i==0)
            minimal=arr[i];
        else
            if (minimal>arr[i])
                minimal=arr[i];
    }
    for(int i=0;i<arr_size;i++){
        arr[i]=arr[i]-minimal+1;
    }
}

void generate_small_V(int arr[], int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i%2==0)
            arr[i]=i*2*step;
        else
            arr[i]=(arr_size*2-1)-i*2*step;
    }
```

```
}

void generate_small_A(int arr[], int arr_size, int step){
    for(int i=0;i<arr_size;i++){
        if(i%2==0)
            arr[i]=(arr_size*2)-i*2*step;
        else
            arr[i]=i*2*step+1;
    }
}

void saveToFile(int** arr,int num_tabl,int num_of_elem,char* name){
    fstream file;
    file.open(name,ios::binary |ios::out);
    for (int i=0;i<num_tabl;i++)
     file.write((char*)(arr[i]),num_of_elem*sizeof(int));
    file.close();
}

void readFromFile(int** arr,int num_tabl,int num_of_elem, char* name){
   ifstream file;
    file.open(name,ios::binary );
    for (int i=0;i<num_tabl;i++)
     file.read((char*)(arr[i]),num_of_elem*sizeof(int));
    file.close();
}

void isSorted(int arr[],int count)
{
                for (int i=1;i<count;i++)
                                if(arr[i-1]>arr[i])
                                 {
                                  cout<<"Not sorted"<<endl;
                                  return;
                     }
   cout<<"Sorted"<<endl;
}


double get_cpu_time(){
                //in ms
    FILETIME a,b,c,d;
    if (GetProcessTimes(GetCurrentProcess(),&a,&b,&c,&d) != 0){

        return
            (double)(d.dwLowDateTime |
            (((unsigned long long)d.dwHighDateTime) << 32)) * 0.0001; //ms
    }else{

        return 0;
    }
}
```