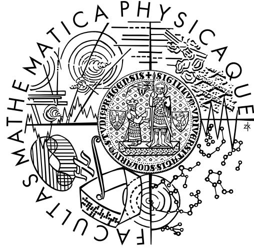

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Josef Matějka

Efficient sorting algorithms for memory hierarchies

Computer Science Institute of Charles University

Supervisor of the master thesis: prof. Mgr. Michal Koucký, Ph.D.
Study programme: Computer Science
Study branch: Theoretical Computer Science

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In . . . . date . . . .

Author's signature

I want to thank my supervisor Michal Koucký for his help and guidance through this journey, without his wisdom and insight I would be lost.

Title: Efficient sorting algorithms for memory hierarchies

Author: Josef Matějka

Institute: Computer Science Institute of Charles University

Supervisor: prof. Mgr. Michal Koucký, Ph.D., Computer Science Institute of Charles University

Abstract: In this thesis, we introduce a novel random cache-oblivious sorting algorithm loosely based on another sorting algorithm called ColumnSort shown in [1]. Our algorithm achieves asymptotic optimality in expected case. As it is cache-oblivious, no further finetuning is necessary. We also demonstrate that its implementation is straightforward and can fit in approximately 100 lines of code, therefore we believe it is accessible and can be easily integrated into existing systems. After we show the implementation in detail, we prove its expected and worst running times and then we compare our implementation with the existing implementation of FunnelSort[2] and std::sort provided by standard C++ library.

Keywords: sorting cache-oblivious, memory hierarchy

Contents

Introduction **2**

1 Introduction to sorting **3**
- 1.1 Problem statement . . . 3
- 1.2 Brief overview of sorting algorithms . . . 3

2 Description of the computation model **8**
- 2.1 RAM model . . . 8
- 2.2 RAM model with external memory . . . 8
- 2.3 Cache-oblivious sorting . . . 9

3 Overview of ColumnSort **10**
- 3.1 Algorithm description . . . 10
- 3.2 Proof of correctness . . . 11
  - 3.2.1 After step four . . . 11
  - 3.2.2 After completion . . . 12

4 SquareSort algorithm **13**
- 4.1 Algorithm overview . . . 13
- 4.2 Detailed description . . . 13
  - 4.2.1 Pivot selection . . . 14
  - 4.2.2 SkewTranspose . . . 15
  - 4.2.3 SkewTranspose complexity analysis . . . 17
  - 4.2.4 Analysis expected bucket sizes . . . 18
  - 4.2.5 Cost of SquareSort recursive call . . . 19
  - 4.2.6 Best case complexity of SquareSort . . . 20

5 Complexity of SquareSort in detail **22**
- 5.1 About distribution of bucket sizes . . . 22
- 5.2 Worst case analysis . . . 24
- 5.3 Expected case analysis . . . 25

6 Experiments **27**
  - 6.0.1 Results . . . 27
  - 6.0.2 Cutoff . . . 28
  - 6.0.3 External sorting . . . 29

Conclusion **31**

Bibliography **32**

List of Figures **34**

List of Abbreviations **35**

A Attachments **36**
- A.1 Square sort implementation . . . 36

Introduction

In this thesis, we will introduce a novel cache-oblivious sorting algorithm. Although existing cache-oblivious algorithms already achieve the asymptotical optimum, both in the speed and number of IO operations. The two most popular are FunnelSort and cache-oblivious distribution sort. FunnelSort may be challenging to implement, while distribution sort is a significantly simpler algorithm using linear median find as a sub-routine. The verb implement means to us describing the algorithm in a C-like pseudocode.

We thought we could develop a cache-oblivious version of the quicksort algorithm. Our intuition was that random pivot selection could speed up the practical sorting on modern computers as it is the case in Quicksort. Which has quadratic time complexity in the worst case but is optimal on average, and it is usually the fastest algorithm for practical sorting. In addition, random algorithms seem easier to implement, therefore we could have an algorithm that performs well in practice and can be described easily.

There exists a randomized cache-oblivious algorithm [3] which is mainly aimed at multi-core parallel model. This algorithm utilizes the same approach for pivot selection, the pivots are then used to split the array into smaller parts called buckets that need to be sorted separately, yet their approach for dividing the elements into the buckets differs significantly.

The main idea for all these algorithms is to recursively split n elements into parts of size $\sqrt[x]{n}$, where usually $2 \leq x \leq 4$. In each recursive call the number of elements to be sorted drops exponentially therefore in a logarithmic number of steps we reach a size of tasks that fit into the cache. FunnelSort differs the most from other algorithms as it uses k merger to merge k sorted sequences together. While other mentioned algorithms separate the tasks to buckets in such a manner that any element from bucket i is less than or equal to any elements from bucket $i + 1$, therefore there is no need to merge buckets after sorting we can just put one bucket after another.

The main challenge is to select an appropriate pivot for every bucket, while the cache-oblivious distribution sort utilizes the median-select algorithm, the square sort and algorithm [3] utilizes randomness for pivot selection.

1. Introduction to sorting

Sorting is perhaps the most fundamental algorithmic problem [4]. As we will see the problem of taking permutation of elements and ordering them in the correct order was in some sense solved even before we were using computers. We do not have any single algorithm that would solve all our sorting problems and therefore there is a diverse set of algorithms solving it. We can split the algorithms into several categories considering their time or memory complexity, stability, and whether they are comparison-based or use other properties of the given elements. To name a few, we have quick sort, merge sort, heap sort, bubble sort, and bucket sort.

Each algorithm has its benefits and drawbacks, therefore it is necessary to select the correct one for our problem since using the right sorting algorithm can significantly reduce the time or memory complexity. For example, consider Kruskal's algorithm, its complexity depends on how fast we can sort the edges. In some cases, we can use an effective algorithm for integer sorting.

All but one of the sorting algorithms we will mention here are comparison-based sorting algorithms, which means that they need to compare elements in order to sort them. We know, that any comparison-based algorithm must make $\Omega(n \log n)$ comparisons, but for integer sorting, we can do better. As was shown[5] we can reduce the running time to $\mathcal{O}(n\sqrt{\log \log n})$ in expected case.

1.1 Problem statement

As we hinted in the previous paragraphs the main problem will be sorting. We are given elements in any order and we want to produce the sequence of the same elements in non-decreasing order[4]. Additionally, we will require that comparison of two elements can be done in constant time.
Input: Any permutation of n elements, $a_1, a_2, \ldots, a_n$.
Output: A permutation of the elements, such that $a_1 \leq a_2 \leq \ldots \leq a_n$.

1.2 Brief overview of sorting algorithms

As the problem is stated we offer a short overview of current sorting algorithms. As each of them is of theoretic interest, offers an interesting perspective to the solution. As a bonus, all are in some way used in practice.

Radix sort

Probably the oldest sorting algorithm is Radix sort, it dates back to 1887 to the work of Herman Hollerith on tabulating machines [6]. It represents one of the non-comparative algorithms, which means we do not compare directly elements in order to sort them. Instead, we assign each element to an appropriate bucket, the buckets are already sorted and the order of the buckets imposes an order on the elements. Suppose we have n elements consisting of at most w digits/characters. Then the algorithm will run in w rounds, in each we create a bucket for every

possible digit/character and assign the element to the bucket equal to their i-th digit.

For example suppose we have elements $542, 132, 23, 742, 11, 543, 732, 142, 131$. We will start with the least significant digit. We will create 10 buckets 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and put each number into the bucket that matches the last digit. We obtain these buckets:

- $1 - [11, 131]$,
- $2 - [542, 132, 742, 732, 142]$,
- $3 - [23, 543]$,
- and the rest of the buckets are empty.

We will repeat this process, we will take the numbers in order yielded by the buckets and put them into the new set of buckets, and match them against their second digit.

- $1 - [11]$,
- $2 - [23]$,
- $3 - [131, 132, 732]$,
- $4 - [542, 742, 142, 543]$,
- and the rest of the buckets are empty.

Since each number has at most three digits, we will repeat the whole process for the last time but against the last digit. This yields the sequence: $11, 23, 131, 132, 142, 542, 543, 732, 742$. As we can see this algorithm has time complexity $\mathcal{O}(nw)$ where n is the number of elements and w is the maximum size of elements (in terms of digits).

Although the algorithm was known it was not utilized on early computers as it was believed that the size of each bucket must be $\mathcal{O}(n)$ to hold all the elements, therefore the algorithm would need $\mathcal{O}(nw)$ memory. In 1954 H. H. Seward in his master thesis pointed out that we only need $\mathcal{O}(n + m)$ memory, where m is the number of buckets.

Algorithm 1 RadixSort(I, w)

Input: Array I of size n.
Output: Prints elements of I in order.

```
1  B = makeBuckets() ;   // Creates buckets, we will assume 10 buckets,
    one for each digit from 0-9.
2  for i = 1,...,n do          // Add all elements into the first bucket.
3  |  append(B[1], I[i]) ;
4  end
5  for i = 1,...,w do                                  // For every digit.
6  |  B' = makeBuckets() ;
7  |  for j = 1,...,10 do                       // For every every bucket.
8  |  |  for k = 1,..., size(B[j]) do   // For every element in the bucket.
9  |  |  |  digit = B[j][k][i] ;                     // Gets the i-th digit.
10 |  |  |  append(B'[digit] = B[j][k]) ;
11 |  |  end
12 |  end
13 |  B = B'
14 end
15 for j = 1,...,10 do                          // For every every bucket.
16 |  for k = 1,..., size(B[j]) do      // For every element in the bucket.
17 |  |  print B[j][k] ;                       // Prints elements in order.
18 |  end
19 end
```

Bubble sort

The earliest description of bubble sort (also called sinking sort) can be found in a paper named Sorting on Electronic Computer Systems[7] where it is called the "Sorting by exchange" algorithm. Interestingly radix sort is there also mentioned. This algorithm iterates n times over the sequence of n elements. In each pass, it compares two neighboring elements in the sequence and switches them whenever they are out of order. We can imagine this process as bubbles of different sizes going up, or alternatively as objects of different densities sinking down (hence the second name).

For example, consider sequence 542, 132, 23, 742, 11, 543, 732, 142, 131. We compare 542 with 132 since they are out of order we swap them. Then compare 542 with 23, and swap them again. Continuing these exchanges until we arrive at the end of the sequence obtaining 132, 23, 542, 11, 543, 732, 142, 131. Then we go back to the first element and repeat this process, yielding 23, 132, 11, 542, 543, 142, 131, 732, 742. In i-th such pass the i-th largest element will be at the correct position, therefore we need to iterate through the sequence n times. The time complexity is $\mathcal{O}(n^2)$ and the memory complexity is $\mathcal{O}(n)$.

Algorithm 2 BubbleSort(I)

Input: Array I of size n.

Output: Array I with elements sorted.

```
20 for i = 1, ..., n − 1 do
21 │ for j = 1, ..., n − i do
22 │ │ if I[j] > I[j + 1] then      // If I[j] is greater than I[j + 1] swap
   │ │   them.
23 │ │ │ swap I[j], I[j + 1] ;
24 │ │ end
25 │ end
26 end
27 return I ;
```

Insertion sort

We will introduce another sorting algorithm that has the same time complexity, quadratic, as bubble sort, but in practice is faster[8]. Its name is insertion sort. It is a simple algorithm, which can be implemented in three lines of C++ code. Although we have asymptotically faster algorithms than this, it is still used for sorting arrays smaller than a given constant and it is still used in hybrid sorting algorithms like introsort.

Suppose we are given input of n elements in array I. We start with a sorted array O which will be empty at the start. In each step, we take the first element from the array I and we will put it in the right place in the array O. If we take the sequence 542, 132, 23, 742, 11, 543, 732, 142, 131 again. In the first step, we take 542 from I and put it into O, then we take 132, which in O belongs to the first index, therefore we need to move 542 to the second one. We continue this until there is no element left in the array I. As every insert can potentially necessitate moving all elements from O every insert takes $\mathcal{O}(n)$, therefore together it takes $\mathcal{O}(n^2)$.

```
Algorithm 3 InsertionSort(I)
Input: An array I of size n.
Output: Array O with elements sorted.
28 O = copy(I) ;                              // Allocate output array O.
29 for i = 1, ..., n do                        // For every element in I.
30 |  element = I[i] ;
31 |  for j = 1, ..., i − 1 do
32 |  |  if element < O[j] then  // Find the position for the element and
   |  |   move all the items after it.
33 |  |  |  swap O[j], element ;
34 |  |  end
35 |  end
36 |  O[i] = element ;
37 end
38 return O
```

Heapsort

Sorting can be simplified if we use the right data structure for our data. This approach was discovered by Williams[9] in 1964 who also discovered heap data structure. Heapsort is also part of introsort, where it is used whenever the quicksort degenerates and requires too many recursive calls. In that case, we can fall back to this algorithm which has time complexity $\mathcal{O}(n \log n)$.

The goal is to take the input, put it into a heap, and then by n times calling extract-min we can obtain all elements in the correct order. We can build the heap over the input array needing only constant additional memory. Of course, we must check and fix the heap property, meaning we want both children of any node to be less than or equal to their parent. After that we can work with the array as we would with a regular heap.

Quicksort

Quicksort is probably the go-to algorithm for sorting. It was discovered by Hoare[10] already in 1959 but he published it in 1962. It is a representation of the divide-and-conquer algorithm, it uses a theory of probability for reasoning about its efficiency. Therefore we talk about expected time complexity, which is $\mathcal{O}(n \log n)$, but in worst case the complexity is $\mathcal{O}(n^2)$. In practice, it is usually faster than heapsort, especially on random data.

In each step, we want to split the input array into two parts, which will be in the best-case scenario of similar lengths. For that, we select one element called pivot. We put all elements less than pivot into the first part, while the rest will be in the second. After that we have two smaller problems, and we recurse on them. We do this until the size of the array is not constant.

2. Description of the computation model

To be able to speak about complexity we need to define the model we will be working with as we cannot reason about memory or time complexity if we do not know how much each operation takes or how the memory is organized. We use a random access machine[11][12] - which is also sometimes called register machine - we will abbreviate it as RAM. This model approaches real computers sufficiently well and it is commonly used for algorithm analysis.

2.1 RAM model

The model consists of registers - this is our memory. There is an infinite number of them, but we will measure how many the algorithm needs to use. Each register can hold any integer of size $\mathcal{O}(\log n)$-bits. If the size of integers would be unbounded we could encode all the information the algorithm needs into a single register which is unrealistic.

We can divide the instruction set of the model into four categories:

- Instructions for integer arithmetic.
- Instructions for reading/writing from/into registers.
- Instruction for control of program flow.
- Instructions for reading/printing the input/output.

We will model our instruction set as a subset of C language. For the arithmetics we will have $+, -, *, /, \%$, we will also include integer comparison like $==, <, >, <=, =>$. For reading and writing, we use assignment $=$. For control flow, we will have *for*, *while* loops, and *if / else* statements. We will also work with functions therefore we need to be able to call them and return from them. As we will also need random bits, we will add to the model instruction for obtaining a single random bit. This random instruction will return either 0 or 1 with uniform probability. Using this instruction we can also generate random integers.

The algorithm will be a sequence of instructions in read-only memory, therefore our theoretical machine will have Harvard architecture. The algorithm will run until it reaches the last instruction in the sequence then it stops. As there are instructions that control the flow, it can happen that a program will run forever. We will not consider such programs correct, the same applies if there is division by zero. Each instruction will take a single unit of time, therefore the time complexity is a sum of executed instructions by the program.

2.2 RAM model with external memory

We have stated that the RAM approximates real computers sufficiently well. This is true in terms of the instruction set and how the memory can be accessed.

Unfortunately, it does not account for the memory hierarchy. As reading from memory takes in modern computers non-trivial time, we have equipped CPUs with cache. If data is needed and it is not available in the cache the processor must first load it into the cache and then work with them.

As real CPUs are complicated, they have several layers of cache, they can differ in cache placement and replacement policies, we will utilize a theoretical model called (M, B)-ideal cache model defined in [13], and [14]. This model has only two layers of memory (cache and external memory), optimal cache replacement strategy, and full associativity of the cache.

In our model, the registers can be thought of as the external memory. In order to work with them we need to load them into the cache. As the size of the cache is M - meaning we can fit M integers of size $\mathcal{O}(\log n)$ inside, it can easily happen that our cache is full. In that case, the cache will evict some data, our cache model knows what data to evict in order to minimize the cache misses. A cache miss event occurs when we cannot find data in the cache and we need to load them. On the other hand, when the data is found in the cache, we call this cache hit.

The other parameter B tells us how many integers fit into a single cache line. To simulate reality we cannot load a single piece of data, but the whole cache line only. This means that also our external memory needs to be split into these lines. We have also one more assumption on the relation between M and B, which is called tall cache. We want $M = \Omega(B^2)$. It can be shown that this assumption is necessary to build an optimal cache-oblivious sorting algorithm [15].

The goal is to minimize the number of cache misses the algorithm does while it cannot use any knowledge about M and B. We call such algorithm cache-oblivious. It means that no matter our cache hierarchy in the computer the algorithm without any fine-tuning utilizes the cache in an asymptotically optimal way. If we adjust the algorithm using parameters M and B we obtain a so-called cache-aware algorithm.

Our model may seem to have too strong assumptions and maybe such a model can be seen as impractical since we cannot create a cache that would fulfill all the assumptions. It has been shown that any algorithm running on the ideal (M, B) cache model can be ported to LRU (least recently used) $(2M, B)$ cache model while incurring at most double the number of cache misses[13]. Furthermore, empirical results show that using this model is justified.

2.3 Cache-oblivious sorting

The IO complexity of classical sorting algorithms was studied and tested in [16]. We know that utilizing the cache can speed up the algorithm significantly despite the fact we end with longer and more complex code. Another example of a significant increase in speed by lowering the number of cache misses is Yaroslavskiy's Quicksort, which uses two pivots[17][18].

The next natural question is whether we can improve the speed even further by creating an algorithm that is asymptotically optimal in IO operations. It was shown in [19] that the optimal lower bound is $\mathcal{O}\left(n \log_M n\right)$ along with two well-known cache-oblivious sorting algorithms, FunnelSort and distribution sort.

3. Overview of ColumnSort

Before we dive into the SquareSort we would like to spend a few words about another sorting algorithm, which served as an inspiration. It is called ColumnSort and it was discovered by Leighton[1] in his paper giving tight bounds on the parallel sorting. While his computation model was different, as the algorithm was developed for parallel computation with $\mathcal{O}(n)$ processors which will sort n numbers in $\mathcal{O}(\log n)$ steps, the ideas can be converted for sequential RAM and create a cache-oblivious algorithm.

3.1 Algorithm description

Let the input I be a $r \times s$ matrix of numbers where $rs = n$, $s \mid r$ and $r \geq 2(s-1)^2$. The output O of the algorithm will be again a matrix where each column will be sorted and any element from i-th column will be less than or equal to any column from j-th column where $1 \leq i < j \leq s$. Rank p for element on position i, j is $p = i + jr$.

The algorithm itself consists from 8 steps. Steps $1, 3, 5, 7$ sort all elements in each distinct column, while steps $2, 4, 6, 8$ permute the elements of the matrix. The first kind of permutation done in step 2 is called "transposition" - while it is not transposition in the traditional sense it is similar. The entries of the matrix will be picked column by column and we will fill rows with them. As the rows are smaller than the columns, one column can fill multiple rows. The operation in step 4 is inverse to the "transposition" we will pick each row and save it into columns.

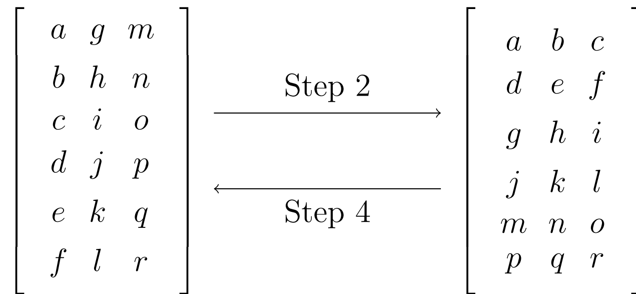

Figure 3.1: The transpose and untranspose permutations in Steps 2 and 4, respectively.

The last kind of permutation is called shift. The entries of the matrix need to be shifted by $\lceil s/2 \rceil$ position in each column, while the elements that do not fit into their column anymore need to be moved to the next column. This operation will add one column to the matrix. We fill the first half of the first column by $-\infty$ and the last half of the last column by ∞. We do this in step 6 and in step 8 we do the inverse operation to shift.

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \underset{\text{Step 8}}{\overset{\text{Step 6}}{\rightleftarrows}} \begin{bmatrix} -\infty & d & j & p \\ -\infty & e & k & q \\ -\infty & f & l & r \\ a & g & m & \infty \\ b & h & n & \infty \\ c & i & o & \infty \end{bmatrix}$$

Figure 3.2: The shift and unshift permutations in Steps 6 and 8, respectively.

3.2 Proof of correctness

The proof of correctness of the algorithm can be split into two parts. First, we will analyze the rank of any element after step four is done, we will show that in that case, the position of said element is within $(s-1)^2$ places of its correct position. Assuming this we then will show that the rest of steps is sufficient to sort the matrix.

3.2.1 After step four

We will define $rank_s(x)$ which is an index of element x in the sorted sequence, this is zero-based. We also define $rank_i(x)$ which is an index in the current matrix after the step i - if x lies in the matrix at position i, j, then its index is $ri + j$, again the positions are zero-based.

Let Q_i be the matrix after the i-th step of the main program. This means that Q_0 is the original matrix we will use for sorting, Q_1 is the matrix after the first sorting of the columns and Q_8 is the resulting matrix. We will also establish $rank_i(x)$ and we will compare it to $rank_s(x)$ rank of x.

Take the element x in Q_3, it lies on the position i, j. Now we will calculate how the position changes when we do the transpose operation, therefore what is its position in Q_4. As untransposition puts the rows into columns, we can see that the new position is $rank_4(x) = si + j$.

Now we wish to argue about some properties of x in Q_3. Elements in the columns are sorted therefore each item before x in the same column must be lower or equal to x. This means there are i elements and if we check which column these elements occupy in Q_1 we can see that $i = \sum_{k=1}^{s} a_k$. Where a_k is the number of elements in k-th column in Q_1 that are currently in Q_3 lower or equal to x.

We wish to compute $|rank_3(x) - rank_s(x)|$. For each a_k but last we know that there were s lower elements in Q_1 since it still holds that after step 3 the rows are non-decreasing. Therefore we can get lower and upper bound for the $rank_s(x)$.

$$rank_s(x) \geq \left(\sum_{k=1}^{s} (a_k - 1)s + j + 1 \right) - 1$$

$$rank_s(x) \geq si - s^2 + sj + s - 1 \geq s_i + s_j - (s-1)^2$$

For the upper bound, we will proceed in similar fashion. The number of elements that are bigger than x in j-th column is $r - i = \sum_{k=1}^{s} a_k$, this leads to

$$rank_s(x) \leq \sum_{k=1}^{s} (a_k - 1)s + s - j$$

$$rank_s(x) \leq rs - si - s^2 + s^2 - sj = si + sj$$

Now we only need to compare our ranks with $rank_4(x)$:

$$rank_4(x) - si - sj + (s-1)^2 = (s-j)^2 - j(s-1) \leq (s-1)^2$$

$$si + sj - rank_4(x) = j(s-1) \leq (s-1)^2$$

Therefore we know that the $rank_4(x)$ differs from $rank_s(x)$ at most by $(s-1)^2$.

3.2.2 After completion

After step 5 we can observe that if x is in the top half of a column it either will end there or it could be possibly somewhere in the bottom half of the column preceding x. Since the current position differs from the rank at most by $(s-1)^2 < r$. This is the reason we need to utilize shift, to compare the element x with elements in the preceding column. Sort the columns again, and then proceed to unshift. After the algorithm completes we will obtain a sorted matrix, where the sorted sequence is given by the columns.

4. SquareSort algorithm

In this chapter, we will introduce our cache-oblivious sorting algorithm, which is asymptotically optimal, yet uses a different approach than FunnelSort and Distribution sort. It could be an easier-to-implement alternative, especially to the FunnelSort. Our experiments also hint that SquareSort may be faster in practice. Unfortunately, both algorithms are still slower than introsort algorithm implemented in GCC and Microsoft standard C++ libraries.

4.1 Algorithm overview

The algorithm is inspired by the ColumnSort algorithm of Leighton[1]. It has some resemblance to the distribution sort[14] algorithm for various cache models and it also has some similarities with the cache-oblivious version of distribution sort. The input is seen as $m \times m$ square matrix, where $m = \sqrt{n}$. It is a recursive algorithm. In each call, it recursively sorts the columns of the matrix, and then it performs skew transposition. Afterward, it sorts the columns given by skew transposition again.

The main subprocedure of the algorithm called skew transposition should in the ideal case split the matrix into m columns of size m, the transposition also imposes order on the columns. Meaning that any element from one column is less than or equal to any element from the next column. Therefore when the elements inside every column are sorted the whole matrix is sorted. For the transposition, we need $m - 1$ pivots that split the elements into sets of roughly equal sizes, in this step we defer to randomness and select the pivots by chance. This approach is similar to the quicksort pivot selection.

The key insight is that we can perform skew transposition while keeping the number of IO operations in $\mathcal{O}(m + n/B)$. We utilize the divide-and-conquer strategy that can be found in many cache-oblivious algorithms and mainly in the cache-oblivious matrix transposition algorithm.

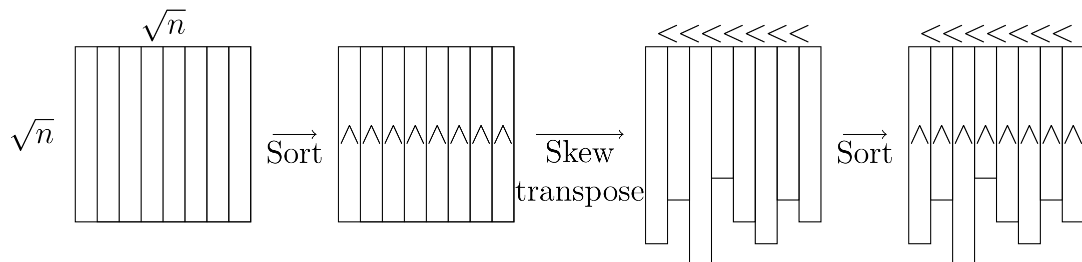

Figure 4.1: An illustration of the SquareSort algorithm.

4.2 Detailed description

To simplify the description of the algorithm we will assume that the elements in the input array I are distinct. We take the elements from I and return them ordered in output array O. The elements in the input array may be permuted after the procedure.

The procedure as the first step checks the size of I, if it is less than some constant we will sort the array using any other sorting algorithm, this is our base case. After that, the input needs to be partitioned into m columns, where each column has size at most $m = \sqrt{n}$. We will store the beginning of each column in the array $C_s[1 \ldots m]$, also for each column we will have an index pointing to the first element after the given column in the array $C_e[1 \ldots m]$. Once we have the columns determined we will sort them using a recursive call.

In the next stage we need to select $m - 1$ distinct pivots and put them in the array $P[1 \ldots m-1]$, we will also define $P[0] = -\infty$ and $P[m] = \infty$. Each pivot determines a bucket of elements. The quality of the pivots determines the size of the buckets. As we have the pivots we can compute the size of each bucket and save this information to array $B_s[1 \ldots m]$. Using this we can also calculate the last index of each bucket in the output array O, we will save these indices in the auxiliary array $B_i[1 \ldots m]$. Now we have sufficient information to start the SkewTranspose sub-procedure, which transposes the data from array I and puts them to the output array O. After that, we need to sort recursively each bucket.

Algorithm 4 SquareSort(I, O, n)

Input: Arrays I and O of size n.
Output: Sorts items from I into O.

```
39 if n ≤ 16 then
40 │ simple_sort(I, O, n); ;                       // sort small arrays directly
41 end
42 m = ⌈√n⌉;
43 Allocate arrays Cs[1...m], Ce[1...m], P[1...m], Bs[0...m], and Bi[0...m];
44 for i = 1,...,m do                          // calculate span of each column
45 │ Cs[i] = 1 + min((i − 1) ∗ m, n); Ce[i] = 1 + min(i ∗ m, n);
46 end
47 for i = 1,...,m do                           // sort each column of I into O
48 │ SquareSort(I[Cs[i], Ce[i] − 1], O[Cs[i], Ce[i] − 1], Ce[i] − Cs[i]);
49 end
50 Sample uniformly at random set S of m − 1 distinct elements from I \ {min(I)}
     and sort S ∪ {−∞, ∞} into P[0...m] ;                      // select pivots
51 Calculate Bi[1...m], where Bs[j] = 1 + |{t ∈ {1,...,n}; O[t] < P[j]}|
52 SkewTranspose(I, O, m, Cs, Ce, m, P, Bi); ;     // skew transpose I into O
53 Set Bi[0] = 1 ;        // SkewTranspose shifted Bi[1...m] by one position
54 for j = 1,...,m do                         // sort each bucket from I into O
55 │ SquareSort(I[Bi[j − 1], Bi[j]], D[Bi[j − 1], Bi[j]], Bi[j] − Bi[j − 1]);
56 end
```

4.2.1 Pivot selection

The selection of the pivots can be done by sampling $m - 1$ elements from D uniformly at random. Whenever any element is picked multiple times we can restart the sampling. Analyzing this is akin to analyzing the famous Birthday paradox, but in our case, we do not have 365 days, but n. Nevertheless, the

probability that we pick any element multiple times is bounded by a constant. Therefore in expectation, there will be also a constant number of restarts. We also can add further requirements to the pivots, we wish that the least pivot is distinct from the least element in I. This can be verified by checking the least element in each sorted column.

To show this we start by bounding the probability that no element was picked multiple times denoted as $p(n)$ where n is the number of pivots we wish to choose. In our algorithm the $n > 16$, therefore $\frac{m}{n} < \frac{1}{2}$. This can be expressed as

$$p(m-1) = \prod_{k=1}^{m-2} \left(1 - \frac{k}{n}\right) \geq \prod_{k=1}^{m-2} e^{-2\frac{k}{n}}.$$

We used the well known bound $1-x > e^{-2x}$ which holds for $x \in [0, \frac{1}{2}]$[20]. We can sum the exponents utilizing another well known expression $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

$$p(m-1) \geq e^{-\frac{(m-2)(m-1)}{n}} = e^{-1+\frac{3}{m}+\frac{2}{n}} > \frac{1}{e} > 0.3$$

4.2.2 SkewTranspose

A skew transposition takes pivots P, bucket indices B_i and sizes B_s, input I, and output O array and transforms the data from I columns to buckets in O. As each bucket is defined by its pivot, it holds that any element from i-th bucket is less than any element from j-th bucket, where $i < j$, therefore as the buckets itself become sorted the whole output array O will be sorted.

To better illustrate how SkewTranspose works we start with a naive version. The idea is to iterate through each sorted column $C_s[i]$ for $i \in [1 \dots m]$. For every element e in the column, we want to determine to which bucket belongs meaning we are looking for i such that $P[i-1] < e \leq P[i]$. Then we move e to $A[B_i[i]]$ and increment $B_i[i]$ by one.

Algorithm 5 NaiveSkewTranspose($I, O, \ell, C_s, C_e, k, P, B_i$)

Input: A source array I, a destination array O, starting positions $C_s[1, \ell]$ of sorted sub-columns in I, $C_e[1, \ell]$ upper-bound positions of sub-columns in I, k pivots $P[1, k]$ and positions $B_i[1, k]$ of free slots in corresponding buckets in O.

Output: Moves items of sub-columns $C_s[1, \ell]$ that are less than $P[k]$ into their respective buckets in O. Updates $C_s[1, \ell]$ and $B_i[1, k]$ which are passed by reference.

```
57 for i = 1, ..., k do
58     for j = 1, ..., ℓ do
59         while C_s[j] < C_e[j] and O[C_s[j]] ≤ P[i] do
60             O[B_i[i]] = I[C_s[j]];
61             B_i[i] = B_i[i] + 1;
62             C_s[j] = C_s[j] + 1;
63         end
64     end
65 end
```

The problem with this approach is that we can incur a cache miss for each element of a column if each element belongs to a different bucket. This means that the whole transposition can take $\mathcal{O}(n)$ IO operations, which is as bad as it can get. As we have a constant number of IOs for every element. We want to lower this to $\mathcal{O}(n/B)$, which utilizes the caches in an asymptotically optimal manner.

As is common in cache-oblivious algorithms we will employ the divide and conquer strategy. There is no need to transform the whole column before we move to the next one. In fact, at the start, we can divide the work into four parts. One part will solve the first half of columns $[1 \dots m/2]$ using the first half of buckets, the second part will solve the second half of buckets in the same columns, and analogously third and fourth parts will solve columns $[m/2 \dots m]$. Notice that we solve every part by a recursive call to SkewTranspose. We continue this division of work until the size of the column to be transposed is trivial, now we can employ the naive approach.

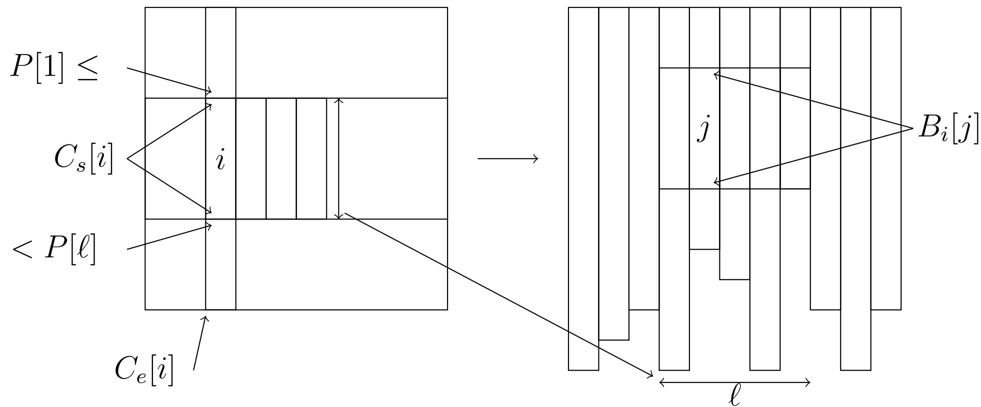

Figure 4.2: Illustration of a call to SkewTranspose. Pointers $C_s[i]$ and $B_i[j]$ will advance during the procedure.

Another important detail is that we need to pass the arrays C_s and B_i as a reference (memory pointer) since we modify them in each NaiveSkewTranspose call. The C_s serves as a mark where in each column we have ended and what element needs to be taken of next and in a similar matter B_i tells us at which index we put the next element to the appropriate bucket.

Algorithm 6 SkewTranspose$(I, O, \ell, C_s, C_e, k, P, B_i)$

Input: A source array I, a destination array O, starting positions $C_s[1,\ell]$ of sorted sub-columns in I, $C_e[1,\ell]$ upper-bound positions of sub-columns in I, k pivots $P[1,k]$ and positions $B_i[1,k]$ of free slots in corresponding buckets in O.

Output: Moves items of sub-columns $C_s[1,\ell]$ that are less than $P[k]$ into their respective buckets in O. Updates $C_s[1,\ell]$ and $B_i[1,k]$ which are passed by reference.

```
66 if ℓ < 4 or k < 4 then
67 │ NaiveSkewTranspose(I, O, ℓ, C_s, C_e, k, P, B_i).
68 end
69 ℓ' = ⌊ℓ/2⌋;
70 k' = ⌊k/2⌋;
71 SkewTranspose(I, O, ℓ', C_s[1, ℓ'], C_e[1, ℓ'], k', P[1, k'], B_i[1, k']);
72 SkewTranspose(I, O, ℓ − ℓ', C_s[ℓ' + 1, ℓ], C_e[ℓ' + 1, ℓ], k', P[1, k'], B_i[1, k']);
73 SkewTranspose(I, O, ℓ', C_s[1, ℓ'], C_e[1, ℓ'], k − k', P[k' + 1, k], B_i[k' + 1, k]);
74 SkewTranspose(I, O, ℓ−ℓ', C_s[ℓ'+1, ℓ], C_e[ℓ'+1, ℓ], k−k', P[k'+1, k], B_i[k'+1, k]);
```

4.2.3 SkewTranspose complexity analysis

Procedure SkewTranspose is an integral part of the SquareSort algorithm and it can lead to cache-oblivious sorting only if it is asymptotically optimal in the number of possible IO operations. We will analyze its cache complexity in this section. Assume that we have array I with its columns sorted, the length of I is $n \geq 4$. Let $m = \sqrt{n}$, we have arrays $C_s[1,\dots,m]$, and $C_e[1,\dots m]$ which store starting and ending indices for every column in I. We have also selected $m+1$ pivots and put them in the array $P[0,\dots,m]$, the first and last elements are $P[0] = -\infty$, $P[m] = \infty$, while the rest is randomly selected from elements of I. We also have array $B_i[1,\dots,m]$ which stores the starting index of every bucket in the array O.

Lemma 4.2.1. *There exists $c_{ST} > 0$ such that for any $n \geq 4$, $m = \lceil\sqrt{n}\rceil$, $B \geq c_{ST}$, $M \geq B^2$,* SkewTranspose$(I, O, m, C_s, C_e, m, P, B_i)$ *causes at most $\mathcal{O}(1 + n/B)$ IO's.*

Proof. We start with an observation about the relation between arguments ℓ and k during the recursive calls. For the first call to SkewTranspose we have $\ell = k = m$, we claim that in each call $|l - k| \leq 1$. We analyze two cases:

- $k = l$ then $|l' - k'| = 0$, $|l' - k + k'| \leq 1$, $|l - l' - k'| \leq 1$, $|l - l' - k + k'| \leq 1$,
- $|l - k| = 1$ then $|l' - k'| = 1$, $|l' - k + k'| \leq 1$, $|l - l' - k'| \leq 1$, $|l - l' - k + k'| = 1$.

As the depth of the recursion is given by ℓ or k and they differ by at most one, we can observe that it is bounded by $\log_2 m + 1$. The arguments of each call consist of references and integers only, we pass arrays as references and span inside them as two indices, therefore we can bound each call by a constant. This means that the memory needed to remember all recursive calls is bounded by $\mathcal{O}(\log m)$.

The next portion of our proof consists of determining the size of k and ℓ such that we can load the whole computation of SkewTranspose into memory and determine the number of IO operations during the computation. Consider $k < B/4$ which implies $l \leq k+1 \leq B/4$. The subsequent depth of recursion is also bounded by $\min(\ell, k) < B/4$, the local variables and parameters of recursion of each SkewTranspose call is bounded by a constant (we will assume that it fits into a single cache line). To hold each relevant block of data from arrays C_s, C_e, P, and B_i we need a constant number of cache lines. Finally, we need a cache line for each column/bucket we are transforming the data from/to, therefore we need $\leq B/2$ lines. Therefore as $k < B/4$ we can fit the whole computation into cache.

The number of cache misses depends on the number of elements we move from the columns to the buckets. Consider sets I and J, which consist of sets of indices from $[m]$. Each set $i \in I$ corresponds to columns, and each set $j \in J$ corresponds to buckets we are working with in each SkewTranspose call where $k < B/4$. Therefore, $|i|, |j| \leq B/4$. In each such call we move $n_{i,j}$ elements, where $\sum_{(i,j)\in I\times J} n_{i,j} = n$. From that, for each (i,j) pair as we can fit all necessary information into the cache, we have $B + \frac{2n_{i,j}}{B}$ IO's.

The outermost call to SkewTranspose with $k = m$ will need to visit every $(i,j) \in I \times J$, this is also the number of calls to SkewTranspose with $k < B/4$, which is bounded by

$$\left(\left\lceil \frac{8m}{B} \right\rceil\right)^2 \leq \frac{128n}{B^2}$$

Each internal node of the recursive call tree we have presented will incur one IO operation on a call stack as the parameters and local variable fit into one cache line. Therefore the number of IO operations given by the outermost SkewTranspose is given by:

$$\frac{128n}{B^2} + \sum_{(i,j)\in I\times J} \left(B + \frac{n_{i,j}}{B}\right) \leq \frac{128n}{B^2} + \frac{128nB}{B^2} + \frac{n}{B} \leq \frac{131n}{B}$$

If $n \leq B^2/8 \leq M/8$ we can store the call stack and arrays I and O in the cache, therefore utilizing only a constant number of operations, which gives the complexity $\mathcal{O}(1 + n/B)$ of IO operations. □

4.2.4 Analysis expected bucket sizes

The complexity of the whole SquareSort algorithm depends on the size of buckets which is given by the quality of sampled pivots from our array I. In each call of SquareSort we are tasked with selecting $m = \sqrt{n}$ pivots $P[0 \dots m]$ where $P[0] = -\infty$, $P[1] \neq \min(I)$, and $P[m] = \infty$. We will denote the distribution of the bucket sizes by μ_n^m, we will also denote size of each bucket by numbers $n_1, \dots, n_m$.

Lemma 4.2.2. *Let P be array of m selected pivots at random without repetition from the $(n-1)$ elements of array I without the least element* $\min(I)$*. Then the expected size of bucket given by P is*

$$\frac{n}{m}.$$

Proof. Consider the array P with the elements sorted, they have been selected uniformly at random from I without the $\min(I)$ element. Consider it sorted, then each element has a single index it lies on, consider this index to be the rank of the element. We will focus on the expected distance between ranks of two neighboring pivots $P[i]$ and $P[i+1]$. Take the first pair $P[1]$ and $P[2]$ then

$$\mathbb{E}\left[P[1]|P[2]=r\right]=\sum_{i=1}^{r-1}\frac{i}{r}=\frac{r}{2}$$

as $P[1]$ can be any number with rank between 1 and $r-1$ with equal probability.

This observation can be extended for every $P[j] : j \in \{2, \dots, m-2\}$. Similarly, we fix values $P[j-1] = r$ and $P[j+1] = s$, then we have an equal probability to select any number from span $r+1, \dots, s-1$ which gives

$$\mathbb{E}\left[P[j]|P[j-1]=r \wedge P[j+1]=s\right]=\sum_{i=1}^{s-r+1}\frac{i}{s-r}=\frac{s-r}{2}.$$

Lastly the same rationale goes for the last $P[m-1]$, where we consider the distance between rank of $P[m-2]$ and the maximum rank $n-1$:

$$\mathbb{E}\left[P[m-1]|P[m-2]=r\right]=\sum_{i=1}^{n-r}\frac{i}{n-1-r}=\frac{n-1-r}{2}.$$

From our analysis, we can conclude that whenever we pick three neighboring pivots, the expected size of two given buckets is the same. We say that rank of $P[0] = 0$ and rank of $P[m+1] = n$ then we can claim $\mathbb{E}\left[P[1]\right] = \mathbb{E}\left[P[2]-P[1]\right] = \cdots = \mathbb{E}\left[P[j]-P[j-1]\right] = \cdots = \mathbb{E}\left[P[m]\right]$. Therefore the expected size of each bucket is

$$\frac{n}{m}.$$

□

4.2.5 Cost of SquareSort recursive call

We want to observe how much memory each call to SquareSort needs, as we mentioned before since we only need pointers to the necessary arrays and start/end indices, each call therefore cost a constant memory to store the arguments. Then we allocate five arrays (C_s, C_e, P, B_s, B_i) of size $m \leq \lceil\sqrt{n}\rceil$, in each call we shrink the array I by at least $m-1$ elements. Summing these memory requirements together we observe, that each call needs at most $c_a m$ memory and the whole stack needs $\mathcal{O}(n)$ memory since in each call the array shrink by $m-1$ elements.

In SquareSort we call SkewTranspose, its memory requirements can be also bound by $c_1 n$ where c_1 is a constant. We also call merge sort for sorting the pivots, again we can bound the memory requirements by $c_2 n > c_2 m \log m$. Let's bound both these constants by c_{st}. This means that as n shrinks to a size below αM, where $\alpha = \frac{1}{2(c_{\text{st}}+2)}$ we can fit the whole computation into cache and we add only $\mathcal{O}(1+\frac{n}{B})$ IO operations.

For the $n \geq \alpha M$, the memory requirements for argument calls are still constant. We need to allocate arrays of size m that can be done in $\mathcal{O}(1+\frac{n}{B})$ IO's. Selecting pivots repeats constant times in expectation, therefore for this part,

we obtain $\mathcal{O}(1 + \frac{n}{B})$ IO's. Sorting done for $n < 16$ can be bound by a constant number of IO's. Therefore the amount of cache misses excluding the calls to SquareSort are in $\mathcal{O}(1 + \frac{n}{B})$, this means that the complexity for $n \geq \alpha M$ can be expressed as $T(n) = mT(m) + \sum_{i=0}^{m-1} T(n_i) + \mathcal{O}(1 + \frac{n}{B})$, where n_i represent a size of i-th bucket obtained from the pivot selection.

4.2.6 Best case complexity of SquareSort

In this section we will be focused on complexity SquareSort in the best case scenario. The amount of work can be described by recursive formula $W(n) = 2\sqrt{n}W(\sqrt{n}) + \mathcal{O}(n)$ as in the procedure we make two recursive calls to the SquareSort, one call SkewTranspose and prepare arrays C_s, C_e, B_i and P. Since the size of each bucket will be $m = \sqrt{n}$ the second call to SquareSort will take the same amount of work as the first one.

We can represent the formula as a recursive tree with a root. The root has $\sqrt{n}$ children, each operating on a column of size $\sqrt{n}$, and each such node has $\sqrt[4]{n}$ children with column size also $\sqrt[4]{n}$, we continue this on each level we apply square root on number of children and the size of the column. This means that the depth of the tree can be found by solving

$$n^{\frac{1}{2^d}} = 2,$$

$$\frac{1}{2^d} \log_2 n = 1,$$

$$\log_2 \log_2 n = d.$$

On each i-th level of the tree, we need to perform $2^i \cdot \mathcal{O}(n)$ work, therefore the total amount of work is given by

$$\sum_{i=0}^{\log_2 \log_2 n} 2^i \cdot \mathcal{O}(n) \leq 2 \log_2 n \cdot \mathcal{O}(n) = \mathcal{O}(n \log_2 n).$$

Which can be solved using well known formula $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$. Therefore from the point of time complexity is our algorithm optimal in expectation.

The number of IO operations follows a similar recursive formula, but now we need to consider two cases. One is when we can bound the used memory by a universal constant α times the size of cache M, while the second case applies whenever the used memory is over αM.

$$T(n) \leq \begin{cases} O\left(1 + \frac{n}{B}\right), & \text{if } n \leq \alpha M \\ 2\sqrt{n}T(\sqrt{n}) + O\left(1 + \frac{n}{B}\right), & n > \alpha M. \end{cases}$$

Again we can solve this recurrence using the same logic as before. The depth of recursion changes slightly as whenever the size of work is less than αM, since from that point we can solve the sorting using only a constant number of IO operations. Therefore

$$n^{\frac{1}{2^d}} = \alpha M,$$

$$\frac{1}{2^d} \log_{\alpha M} n = 1,$$

$$\log_2 \log_{\alpha M} n = d.$$

We combine this result with our previous analysis of the tree yielding the IO complexity of $2 \log_{\alpha M} n \cdot \mathcal{O}(1 + \frac{n}{B})$ therefore we obtain

$$\mathcal{O}\left(1 + \frac{n}{B}\left(1 + \log_{\alpha M} n\right)\right).$$

5. Complexity of SquareSort in detail

In the previous chapter we have introduced the SquareSort algorithm along with its complexity analysis in the best case, but we have to yet discuss the distribution μ_n^m which is a distribution of bucket sizes given by randomly selected pivot. We would like to give bounds on the probabilities that the size of a bucket is in the desired span and ultimately show that the expected complexity is not far from the best case. Also, we cannot forget to talk about the worst-case complexity, when we are unlucky and we cannot hit the appropriate pivots in our selection.

5.1 About distribution of bucket sizes

We start by bounding the probability that i-th bucket size denoted n_i will be greater than $t\sqrt{n} \leq n_i$ where is $t > 1$. This bound allows us to investigate μ_n^m distribution and therefore the behavior of the algorithm better. Let us just reiterate, that the buckets are derived from the pivots, which are sampled at random without repetition from the input array I.

Lemma 5.1.1. *For any $n \geq 100$, $m = \lceil\sqrt{n}\rceil$, $i \in \{1, \dots, m\}$, $t > 1$ the* $\Pr_{(n_1,\dots,n_m)\sim\mu_n^m}[n_i \geq t\sqrt{n}] \leq e^{-0.9t+0.1}$.

Proof. As the probabilities on distribution for any n_i are the same, we will fix $i = 1$. This means that the bucket size is given by how many elements in I are $\leq P[1]$, where $P[1]$ denotes the first sampled pivot, or equally we can say that we are interested in the rank of this pivot. The pivots are unique, we selected $m - 1$ elements from I universally at random, but whenever we sample any element multiple times, we restart this process. In the previous chapter, we have concluded that this sampling can take constant restarts in expectation.

We start by bounding the probability that the i-th selected element does have rank in $\{2, \dots, \lceil t\sqrt{n}\rceil\}$, then we take the complement of this probability and we are interested in the intersection of this event over all possible i. This gives

$$\Pr_{n_1\sim\mu_n^m}[n_1 \geq t\sqrt{n}] \leq \left(1 - \frac{t\sqrt{n}-1}{n}\right)^{\sqrt{n}-1} \leq e^{-\frac{(t\sqrt{n}-1)\cdot(\sqrt{n}-1)}{n}} \leq e^{-t\cdot\frac{n-\sqrt{n}}{n}+\frac{\sqrt{n}-1}{n}},$$

which can be further simplified. As we are interested only in minimum possible value for fraction $\frac{n-\sqrt{n}}{n}$ and maximum for $\frac{\sqrt{n}-1}{n}$. Since solving these gives the upper bound on the probability. Therefore we can conclude that the probability is

$$\leq e^{-0.9t+0.1}.$$

□

We are not only interested in the probability of buckets being greater than desired but also in buckets that are much smaller than needed. For this, we will bound the probability of bucket i having size $n_i \leq s$ where $s \geq 2$. In contrast to the first bound, this bound tells us about buckets that have at most ”constant” size, where the constant is given by the parameter s.

Lemma 5.1.2. *For any $n \geq 100$, $m = \lceil\sqrt{n}\rceil$, $i \in \{1, \ldots, m\}$, $s \geq 1$ the* $\Pr_{(n_1,\ldots,n_m)\sim\mu_n^m}[n_i \leq s] \leq \frac{2s}{\sqrt{n}}$.

Proof. As in the last proof we fix i to be 1, therefore we are only interested in n_1. This means at least one selected pivot must have a rank from the set $\{2, \ldots, s\}$, therefore the probability of a pivot having such rank can be upper bound by $\frac{s}{n}$.Therefore we are interested in probability, that one from $m - 1$ pivots is in the desired range. We will use the union bound which gives this upper bound:

$$\Pr_{(n_1,\ldots,n_m)\sim\mu_n^m}[n_i \leq s] \leq (m-1)\frac{s}{n} \leq \sqrt{n}\frac{s}{n} \leq \frac{s}{\sqrt{n}}$$

□

In order to show the expected complexity we also need to be able to bound the following expected value $\mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^m n_i \log n_i\right]$. For this, we will use the following lemma.

Lemma 5.1.3. *For any $n \geq 16$ and $m = \lceil\sqrt{n}\rceil$:*

$$\mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^m n_i \log n_i\right] \leq \frac{1}{2} n \log n + 4en.$$

Proof. We start by analysing the bucket n_1 for this we use the previous lemma 5.1.1 from which we derive: $\Pr_{(n_1,\ldots,n_m)\sim\mu_n^m}[t\sqrt{n} \leq n_i < (t+1)\sqrt{n}] \leq e^{-0.9t+0.1}$. We group the possible sizes of n_1 as follows:

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^m n_1 \log\left\lceil\frac{n_1}{\sqrt{n}}\right\rceil\right] &= \sum_{l=1}^{n-1} \Pr[n_1 = l] l \log\left\lceil\frac{l}{\sqrt{n}}\right\rceil \\
&\leq \sum_{t\geq 0} \Pr[t\sqrt{n} \leq n_1 < (t+1)\sqrt{n}](t+1)\sqrt{n}\log(t+1) \\
&\leq \sqrt{n}\sum_{t\geq 0} e^{0.9t+0.1}(t+1)^2 \\
&\leq 3en
\end{aligned}$$

From linearity of expectation we obtain:

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^m n_i \log n_i\right] &= \mathbb{E}\left[\sum_{i=1}^m n_i \log\left(n_i\frac{\sqrt{n}}{\sqrt{n}}\right)\right] \\
&\leq \mathbb{E}\left[\sum_{i=1}^m n_i \log\sqrt{n} + n_i \log\left(\frac{n_i}{\sqrt{n}}\right)\right] \\
&\leq n\log\sqrt{n} + \sum_{i=1}^m 3en \\
&\leq n\log\sqrt{n} + 4en
\end{aligned}$$

□

5.2 Worst case analysis

The main problem with the worst-case analysis is that we cannot say much about the recursive function when the size of the buckets is an arbitrary sequence of $n_1, \ldots, n_m$ such that $\sum_{i=1}^{m} n_i = n$. Since the complexity stems directly from the tree of recursion, we cannot easily show which of the possible sequences leads to the worst case.

Therefore we offer this upper bound, suppose that for our call the i-th bucket has size $n_i = m + c_i > m$, where $0 < c_i < n - m$. Our goal is to keep unpacking the recursive part in the formula until it can be bounded by $W(m)$. Therefore at first, we obtain this:

$$W(m + c_i) = \sqrt{m + c_i} W(\sqrt{m + c_i}) + \sum_{j=1}^{\sqrt{m+c_i}} W(n_j) + c_T(m + c_i).$$

As $m + c_i < n$ then also $\sqrt{m + c_i} < \sqrt{n} = m$. Then we again have the sum of $W(n_j)$ where possibly some n_j can be greater than m, but each n_j is strictly smaller than $m + c_i$. This will help us to create the bound. As each call with $m + c_i$ can have at most c_i recursive calls with a value greater than m we can bound the expression like this:

$$W(m + c_i) \leq c_i m W(m) + c_i m W(m) + c_T c_i (m + c_i).$$

Since we now know how to bound each call such that we end up with $W(m)$ only. We can continue with the worst-case analysis. At the start, we have:

$$W(n) = mW(m) + \sum_{i=1}^{m} W(n_i) + c_T n.$$

We take each $n_i < m$ and bound it by m. For each $n_i > m$ we use our bound, we know that for each such n_i we add c_i levels and on each level, we add $c_i m W(m)$ where $m + c_i = n_i$. As the sum of all c_i must be less than $n - m$ we obtain:

$$W(n) \leq mW(m) + 2(n - m)mW(m) + c_T(n - m)n = 2nmW(m) + c_T n^2.$$

We already know what the depth of our bound will be, $\log \log n$. In level $d+1$ of recursion we will obtain

$$c_T n^{2^{-d+1}} 2^d \prod_{i=1}^{d} n^{\frac{3}{2^i}}$$

of work done. The resulting complexity is the sum of levels 1 through $\log \log n$. We can simplify the expression by utilizing the partial sum for the geometric series for the exponent. This gives an exponent in level $d + 1$:

$$\frac{1}{2^{d-1}} + \sum_{i=1}^{d} \frac{3}{2^i} = \frac{1}{2^{d-1}} + \frac{3}{2} \left(\frac{1 - \frac{1}{2^d}}{1 - \frac{1}{2}} \right) = 3 - \frac{3}{2^{d-1}} + \frac{1}{2^{d+1}} < 3.$$

Since on each level, we can bound the work by $\mathcal{O}(n^3 \log n)$ the whole procedure takes $\mathcal{O}(n^3 \log n \log \log n)$ steps. Utilizing our analysis for IO complexity we can directly translate our result also for the worst case IO operations, in this case, the depth of the recursion $\log \log_M n$ which results in $\mathcal{O}\left(\frac{n^3}{B} \log_2 n \log_2 \log_{\alpha M} n\right)$.

5.3 Expected case analysis

As we showed the best and worst-case scenario, the next step is to show what is the expected running time. We want to show this by using induction. For some trivial n, in our case $n < 16$ we sort the data in constant time incurring constant number of IO operations. This will serve as our base case.

Lemma 5.3.1. *There exist constants c_I, c_M such that expected time $W(n)$ and IO complexity $T(n)$ can be bound by:*

$$\mathbb{E}\left[W(n)\right] \leq c_I n \log n,$$

$$\mathbb{E}\left[T(n)\right] \leq \frac{c_M n}{B} \log_{\alpha M} n.$$

Proof. While this holds for $n-1$, we want to show that it holds also for n. Therefore in the induction step we obtain:

$$\mathbb{E}\left[W(n)\right] = \mathbb{E}\left[mW(m)\right] + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} W(n_i)\right] + c_T n,$$

$$\mathbb{E}\left[T(n)\right] = \mathbb{E}\left[mT(m)\right] + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} T(n_i)\right] + c_T \frac{n}{B}.$$

As $\sqrt{n} = m \leq n-1$ we can bound the first part of both expression utilizing our claim.

$$\mathbb{E}\left[W(n)\right] \leq \frac{c_I}{2} n \log n + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} W(n_i)\right] + c_T n,$$

$$\mathbb{E}\left[T(n)\right] \leq \frac{c_M}{2B} n \log_{\alpha M} n + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} T(n_i)\right] + c_T \frac{n}{B}.$$

It also holds, that each $n_i < n-1$ as we have m buckets and each bucket has a size of at least one. Therefore we can again use our claim to obtain:

$$\mathbb{E}\left[W(n)\right] \leq \frac{c_I}{2} n \log n + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} c_I n_i \log n_i\right] + c_T n,$$

$$\mathbb{E}\left[T(n)\right] \leq \frac{c_M}{2B} n \log_{\alpha M} n + \mathbb{E}_{(n_1,\ldots,n_m)\sim\mu_n^m}\left[\sum_{i=1}^{m} \frac{c_M}{B} n_i \log_{\alpha M} n_i\right] + c_T \frac{n}{B}.$$

We can plug our result from the previous chapter 5.1.3 which will result in:

$$\mathbb{E}\left[W(n)\right] \leq \frac{c_I}{2} n \log n + \frac{1}{2} n \log n + 4en + c_T n,$$

$$\mathbb{E}\left[T(n)\right] \leq \frac{c_M}{2B} n \log_{\alpha M} n + \frac{1}{2B} n \log_{\alpha M} n + \frac{4en}{B} + c_T \frac{n}{B}.$$

Now we can bound $4en + c_T n$ by $(4e + c_T) n \log n$ for time complexity. For IO complexity it is a tad bit more complicated. As long as $n \geq \alpha M$ which means $\log_{\alpha M} n \geq 1$ we can bound $4en + C_T n$ by $(4e + c_T) n \log_{\alpha M} n$, but otherwise, we must bound it by itself, so by $4en + C_T n$. From this we can derive the constants C_I and C_M as both must be $C_I, C_M \geq 1 + 8e + 2c_T$ for $n \geq \alpha M$, but for the other

case, we know that we can bound the number of IOs by a constant. This proves the claim:

$$\mathbb{E}\left[W(n)\right] \leq c_I n \log n,$$

$$\mathbb{E}\left[T(n)\right] \leq \frac{c_M}{B} n \log_{\alpha M} n.$$

□

6. Experiments

To provide a comparison among SquareSort and other sorting algorithms, we compare SquareSort with std::sort and FunnelSort. The first algorithm is a part of the C++ standard library defined in header "algorithm.h" on g++ and implemented as an Introsort algorithm. The Introsort algorithm is a hybrid sort algorithm that combines Quicksort and Heapsort. FunnelSort is another cache-oblivious algorithm; we use its implementation by Rønn [2] which is also written in C++.

We will compare the time each algorithm takes to sort an array of integers. In each step, we want to sort arrays of the total size of one-third of the memory. The arrays will consist of 32-bit signed integers. Since both std::sort and SquareSort are Las Vegas algorithms, the running time is a random variable. We repeat each test on multiple instances and take the average running time. All tests are run on the Linux operating system, the algorithms are written in C++ and compiled by the g++ compiler.

We start with the size of 1000 elements and in each round, we proportionally increase the size of the arrays. We will compare totally four distinct distributions of input elements: a random permutation of numbers in $\{1, \ldots, n\}$, a random sequence of binary values, a random sequence of integers from $\{1, \ldots, n\}$ selected uniformly at random, and a sequence of integers selected uniformly at random from the range $\{1, \ldots, \sqrt{n}\}$. We tested the algorithms on an AMD Ryzen 7 1800X Eight-Core Processor with three levels of caches with sizes of 96K (L1 per core), 512K (L2 per core) and 16MB (L3 shared) respectively, and 32 GB of main memory. (Measurements on other systems gave similar-looking results.) In the implementation of SquareSort whenever the size of an array is less than 1000 elements we sort it directly using std::sort, also in procedure SkewTranspose, we transpose elements directly whenever that given region has less than 10 columns or the number of buckets is less than 10.

6.0.1 Results

For each size, we measure the average time in nanoseconds. As all three algorithms have the same asymptotic time complexity, we normalize the measured average time t as $t/n \log n$, where n is the size of the sorted array. We plot this normalized time per item as it depends on the number of elements n.

For each type of array, std::sort was the fastest, then SquareSort, and last came the FunnelSort. As in the SquareSort, we split the problem into approximately $\sqrt{n}$ problems of size $\sqrt{n}$, this is the reason why we can observe a sudden increase around 10^6, since here we add another recursive call in expectation.

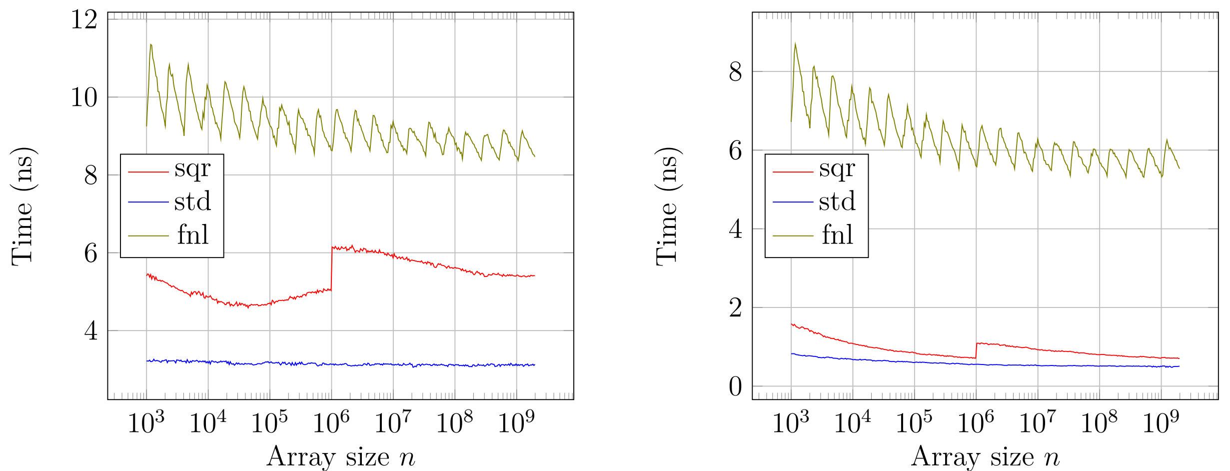

Figure 6.1: Time per item to sort a random permutation (left) and a random binary sequence (right).

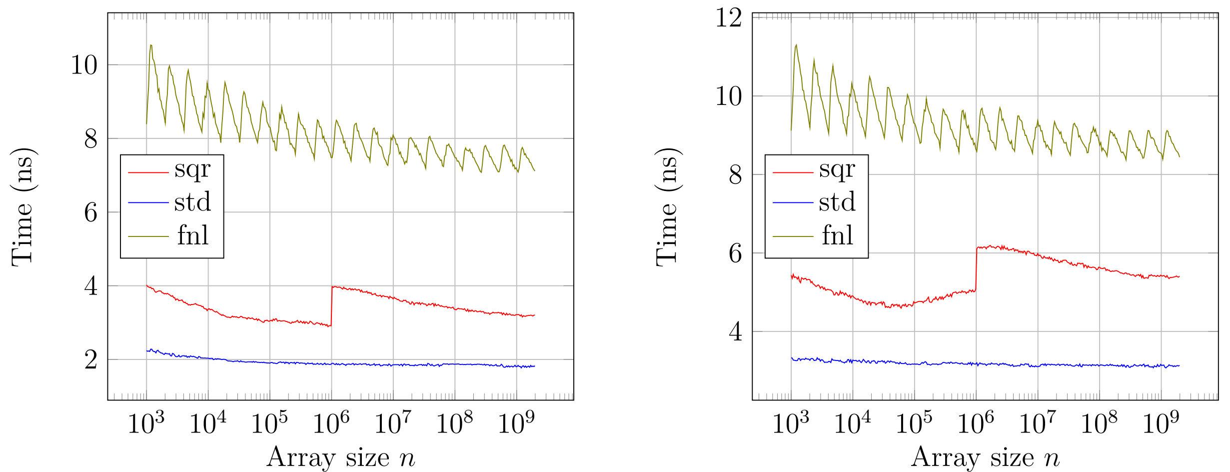

Figure 6.2: Time per item to sort a random sequence of elements from the universe of size n (left) and of size $\sqrt{n}$ (right).

6.0.2 Cutoff

One of the parameters in the SquareSort algorithm is the size of an array that we sort directly by std::sort at the bottom of the recursion; we will call this parameter *cutoff*. We have tested the previous experiment on multiple different cutoffs ranging from 100 to 958. We were interested in how this parameter affects the running time. We present one graph with four cutoffs: 100, 256, 493, and 958. Again we normalize the running time for each size.

The cutoff parameter mainly determines at what size we add an additional recursive call to SquareSort. At cutoff 100 the additional call happens around 10^4 items and then next at 10^8 items. As we increase the cutoff the additional call is added later and for 958 the call is added around one million.

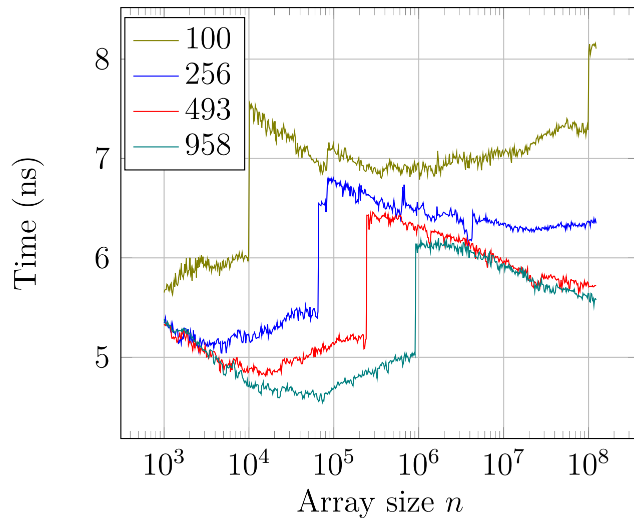

Figure 6.3: Time per item to sort a random permutation with different cutoffs.

6.0.3 External sorting

In order to test our algorithm in more diverse hierarchy of memories we decided to compare the three algorithms running external sort. The goal was to sort files whose sizes exceed the available RAM. As such program needs to access and load data from its external storage and each such accesss can significantly slower the sorting process.

We were testing in on computer with AMD Ryzen 5 7600 6-Core Processor with 8 GB of memory and as an external memory we used ssd disk Samsung SSD 970 EVO Plus 1TB. Our experiments start with small arrays of unsigned 64bit integers we increase the size of arrays until we cannot fit more integers into 64 GB of memory. The graphs are again normalized in the same manner, the time represents time spent on sorting on element of the array, the x axis corresponds to array size.

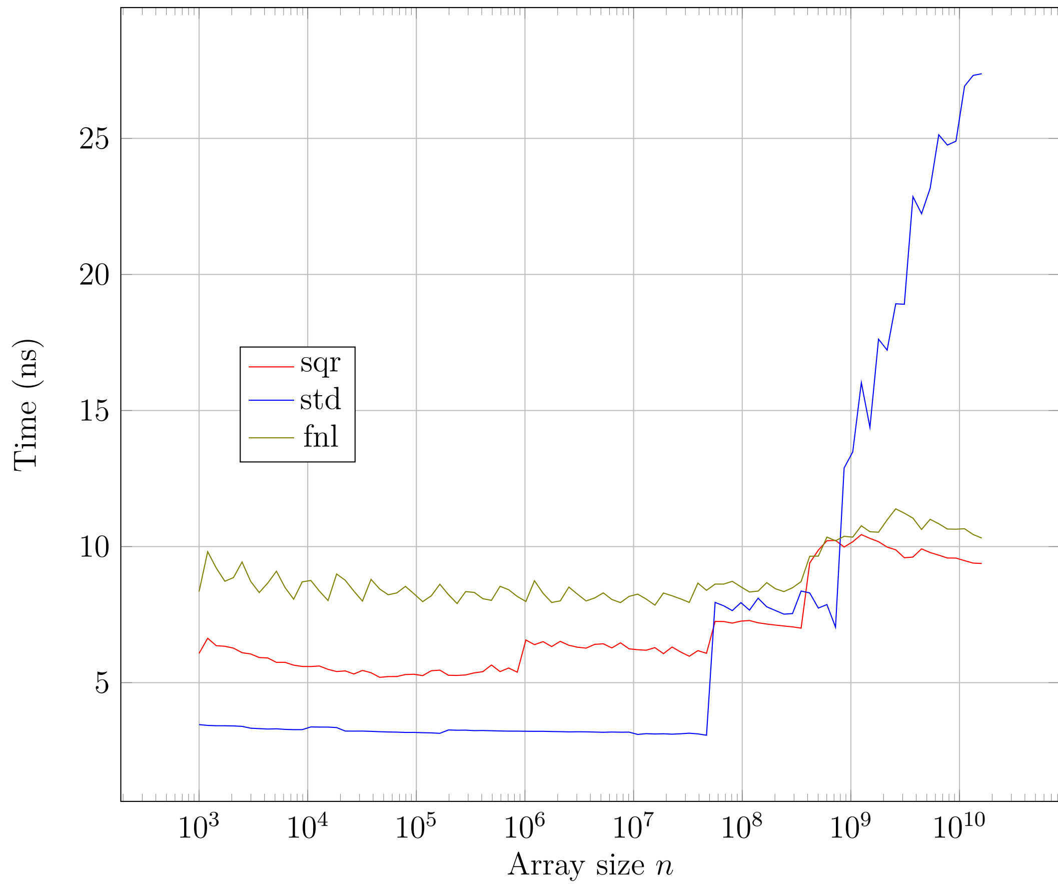

Figure 6.4: Comparison of all three algorithms in external sorting experiment.

Since there was no change in the relative speed of algorithms, we have been testing the arrays with random elements from $[n]$. While we can fit the whole array in the memory the relative order of the algorithms agrees with our previous result, the std::sort is fastest, followed by SquareSort and then by FunnelSort. We can see sharp rise in time taken around size of 860 millions elements, this is due to inability to fit the whole array into memory. From that size further we can notice sharp rise in time spend for the std::sort.

Conclusion

After a short introduction to the problem of sorting and introducing the external memory model in the first two chapters, we have shown the ColumnSort algorithm that served as an inspiration for the SquareSort. In the fourth chapter, we introduced SquareSort, and showed the basic idea and intuition behind the algorithm, in the next chapter we showed the correctness and the complexity. In order to test our theoretical result we compared our algorithm to the implementation of FunnelSort and std::sort which implements introsort.

We showed that SquareSort is optimal both in speed and in IO operations. Furthermore, our tests also suggest that this algorithm may be in practice faster than FunnelSort, but still does not beat the introsort algorithm implemented in the standard C++ library. On the other hand, when the sequence does not fit into the memory anymore our algorithm together with FunnelSort was faster than std::sort. This also shows that our algorithm behaves towards CPU cache and memory better than std::sort. We also believe that it is easier to implement SquareSort than FunnelSort. Our implementation fits under 125 lines of C++ code.

As it is a random algorithm the question for potential future work is whether we can derandomize it without increasing the IO or time complexity, another course of research is whether we can implement this algorithm in place, or at least reduce significantly the amount of temporary memory necessary, this could improve the speed significantly as non-trivial time is spend on copying the element from input array to the auxiliary arrays.

Bibliography

[1] Tom Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 71–80, 1984.

[2] Frederik Rønn. Cache-oblivious searching and sorting. *Diss. Diplomarbeit, Department of Computer Science (University of Copenhagen)*, 2003.

[3] Neeraj Sharma and Sandeep Sen. Efficient cache oblivious algorithms for randomized divide-and-conquer on the multicore model. *arXiv preprint arXiv:1204.6508*, 2012.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[5] Yijie Han and Mikkel Thorup. Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In *Proc. 43rd Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[6] E Knuth Donald et al. The art of computer programming. *Sorting and searching*, 3(426-458):4, 1999.

[7] Edward H Friend. Sorting on electronic computer systems. *Journal of the ACM (JACM)*, 3(3):134–168, 1956.

[8] Oluwakemi Sade Ayodele and Bamidele Oluwade. A comparative analysis of quick, merge and insertion sort algorithms using three programming languages i: Execution time analysis. *African Journal of Management Information System*, 1(1):1–18, 2019.

[9] J. W. J. Williams. Heapsort. *Communications of the ACM*, 27(6):347–348, 1964.

[10] Charles AR Hoare. Quicksort. *The computer journal*, 5(1):10–16, 1962.

[11] Peter van Emde Boas. Machine models and simulations. 1987.

[12] Stephen A Cook and Robert A Reckhow. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80, 1972.

[13] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 285–297. IEEE, 1999.

[14] Harald Prokop. Cache-oblivious algorithms. *Master's thesis, MIT*, 1999.

[15] Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[16] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[17] Sebastian Wild and Markus E Nebel. Average case analysis of java 7's dual pivot quicksort. In *European Symposium on Algorithms*, pages 825–836. Springer, 2012.

[18] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J Ian Munro. Multi-pivot quicksort: Theory and experiments. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 47–60. SIAM, 2014.

[19] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988.

[20] Jiří Matoušek and Jan Vondrák. The probabilistic method. *Lecture Notes, Department of Applied Mathematics, Charles University, Prague*, 2001.

List of Figures

3.1 The transpose and untranspose permutations in Steps 2 and 4, respectively. 10
3.2 The shift and unshift permutations in Steps 6 and 8, respectively. 11

4.1 An illustration of the SquareSort algorithm. 13
4.2 Illustration of a call to SkewTranspose. Pointers $C_s[i]$ and $B_i[j]$ will advance during the procedure. 16

6.1 Time per item to sort a random permutation (left) and a random binary sequence (right). 28
6.2 Time per item to sort a random sequence of elements from the universe of size n (left) and of size $\sqrt{n}$ (right). 28
6.3 Time per item to sort a random permutation with different cutoffs. 29
6.4 Comparison of all three algorithms in external sorting experiment. 30

List of Abbreviations

- RAM - Random Access Machine, a computational model used in this thesis.
- CPU - Central Processing Unit, the main processing unit in a computer.
- IO - Input Output, referring to the operation that loads data from/to external memory.

A. Attachments

A.1 Square sort implementation

```
#include "stdio.h"
#include <time.h>
#include <bits/stdc++.h>
using namespace std;

typedef int64_t T;

void add_bucket_sizes(T A[], T size, T pivots[], T pivot_cnt,
  T bucket_size[]) {
  T i=0;
  T j=0;
  pivot_cnt--;
  for(;i<pivot_cnt;i++) {
    while((j < size) && (A[j] <= pivots[i])) {
      bucket_size[i]++; j++;
    }
  }
  bucket_size[pivot_cnt] += size-j;
}

void sample_pivots(T A[], T size, T sample[], T sample_cnt) {
  T i;
  for(i = 0; i < sample_cnt; i++) {
    sample[i] = (random64() & (0x7FFFFFFFFFFFFFFFLL)) % size;
  }
  sort(sample, sample + sample_cnt);
  for(i = 0; i < sample_cnt; i++) {
    sample[i] = A[sample[i]];
  }
  sort(sample, sample+sample_cnt);
  sample[sample_cnt-1] = T_MAX;
  for(i = 0; i < sample_cnt - 1; i++) {
    if(sample[i] == sample[i+1]) {
      sample[i]--;
    }
  }
}

void naive_skew_transpoze(T A[], T B[],
  T col_start[], T col_end[], T col_cnt, T bucket_start[],
  T bucket_cnt, T pivots[]) {
  T i=0;
  T j=0;
```

```
  for(i=0; i<bucket_cnt; i++) {
    for(j=0; j<col_cnt; j++) {
      while(( col_start[j]  < col_end[j] )
        && (A[ col_start[j] ] <= pivots[i])) {
        B[ bucket_start[i] ] = A[ col_start[j] ];
        bucket_start[i]++;
        col_start[j]++;
      }
    }
  }
}

void skew_transpoze(T A[], T B[],
  T col_start[], T col_end[], T col_cnt, T bucket_start[],
  T bucket_cnt, T pivots[]) {
  if((col_cnt < 10)||(bucket_cnt < 10)){
    naive_skew_transpoze(A, B, col_start, col_end, col_cnt,
      bucket_start, bucket_cnt, pivots);
    return;
  }
  T half_col_cnt = col_cnt /2;
  T half_bucket_cnt = bucket_cnt /2;
  skew_transpoze(A, B, col_start, col_end,
    half_col_cnt, bucket_start, half_bucket_cnt,
    pivots);
  skew_transpoze(A, B, col_start+half_col_cnt,
    col_end+half_col_cnt, col_cnt - half_col_cnt,
    bucket_start, half_bucket_cnt, pivots);
  skew_transpoze(A, B, col_start, col_end,
    half_col_cnt, bucket_start + half_bucket_cnt ,
    bucket_cnt - half_bucket_cnt, pivots + half_bucket_cnt );
  skew_transpoze(A, B, col_start+half_col_cnt,
    col_end+half_col_cnt, col_cnt - half_col_cnt,
   bucket_start + half_bucket_cnt, bucket_cnt - half_bucket_cnt,
   pivots + half_bucket_cnt );
}

void square_sort(T A[], T B[], T size,
  T col_start[], T col_end[], T bucket_start[],
  T pivots[], T buf_size) {
  T i,j;
  if(size < 1000){
    for(i=0;i<size;i++) B[i]=A[i];
    sort(B,B+size);
    return;
  }
   int new_size = sqrt(size);
   sample_pivots(A, size, pivots, new_size);
```

```
  for(i=0;i<new_size;i++){
    col_start[i]=new_size*i;
    col_end[i]=new_size * (i+1);
    bucket_start[i]=0;
  }
  bucket_start[new_size]=bucket_start[new_size+1]=0;
  col_end[new_size-1] = size;
  if(buf_size <= new_size+2)
    throw std::logic_error("Not enough memory for recursion");
  for(i=0;i<new_size;i++){
    square_sort(A+col_start[i],B+col_start[i],
      col_end[i] - col_start[i], col_start+new_size,
      col_end+new_size, bucket_start+new_size+2,
      pivots+new_size, buf_size - new_size-2);
    add_bucket_sizes( B+col_start[i], col_end[i] - col_start[i],
      pivots, new_size, bucket_start+2);
  }
  for(i=2;i<new_size+1;i++) {
    bucket_start[i] += bucket_start[i-1];
  }
  skew_transpoze(B, A, col_start, col_end,
    new_size, bucket_start+1, new_size, pivots);

  for(i=0;i<new_size;i++){
    if((i>0)&&(pivots[i]==pivots[i-1]+1)){
      for(j=bucket_start[i];j<bucket_start[i+1];j++)B[j]=A[j];
    }
    else
      square_sort(A+bucket_start[i],B+bucket_start[i],
        bucket_start[i+1] - bucket_start[i],
        col_start+new_size, col_end+new_size,
        bucket_start+new_size+2, pivots+new_size,
        buf_size - new_size-2);
  }
}
```