



UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Sciences

Computer Science Department

MASTER'S THESIS

LOWER BOUNDS AND ALGORITHMS IN THE LINEAR DECISION TREE MODEL

Advisor

Prof. Jean Cardinal

Student

Aurélien Ooms

Academic Year 2014 - 2015

Abstract

This work exposes lower bounds and algorithms for problems in the Linear Decision Tree Model. All the problems we consider are in some way related to Sorting. We start with Sorting and Merging. Then we explain the more general problems of Sorting under Partial Information and Merging under Partial Information. The last part of this work concerns 3SUM, k -SUM, k -LDT, Sorting $X + Y$, and Point Location in an Arrangement of Hyperplanes.

Contents

Introduction	vii
1 Sorting	1
1.1 The Decision Tree Model	1
1.2 Nonuniformity of the Decision Tree Model	2
1.3 Definition of the Sorting Problem	3
1.4 Information-Theoretic Lower Bound	3
1.5 Stirling's Approximation	4
1.6 Algorithms for the Sorting Problem	5
2 Merging	13
2.1 Definition of the Merging Problem	14
2.2 Merging two Totally Ordered Sets	14
2.3 Merging k Totally Ordered Sets	18
3 Orders, Posets, Graphs, and Entropy	23
3.1 Orders	23
3.2 Posets	24
3.3 Examples of Orders, Sets, and Posets	25
3.4 Isomorphism	26
3.5 Basic Operations, Subposets, and Intervals	28
3.6 Covers and Hasse Diagrams	30
3.7 Least and Greatest Elements	31
3.8 Infimum, Supremum, and Lattices	32
3.9 Special Structures	33
3.10 Graphs, Cliques, Stable Sets, and Entropy	34
3.11 Linear Extensions and Entropy	37
4 Sorting under Partial Information	39
4.1 Definition of the Problem	40
4.2 History	41
4.3 Algorithms	42
4.4 Linial's Algorithm	44

4.5	Efficient Implementation of Linial's Algorithm	47
5	Related Problems	51
5.1	Sorting $X + Y$	51
5.2	Extending the Model	52
5.3	Three Set Sum Problems	53
5.4	Sorting is a 2LDT problem	53
5.5	Sorting $X + Y$ is a 4LDT problem	54
5.6	Similar Complexity for k -SUM and k -LDT	55
6	Sorting $X + Y$	57
6.1	A First Approach by Merging	57
6.2	Naively Looking at the Poset Structure	59
6.3	A Decision Tree for a Generic Sorting Problem	63
6.4	Counting Linear Extensions	68
6.5	Insight into Buck's Theorem	69
7	Decision Trees for 3SUM	71
7.1	3SUM-hard problems	71
7.2	The Conjecture	72
7.3	k -linear Decision Trees	73
7.4	s -linear Decision Trees	74
7.5	4-linear Decision Trees	74
7.6	$(2k - 2)$ -linear Decision Trees	76
8	Application of Meiser's Algorithm	79
8.1	Reduction to a Point Location Problem	79
8.2	Meiser's Algorithm	80
8.3	Strongly Polynomial Nonuniform Algorithm for Subset Sum	84
8.4	Solve k -SUM using only $o(n)$ -linear Queries	85
	Bibliography	87

Introduction

This work focuses on studying the complexity of problems and solving problems in the linear decision tree model. In the model we use, solving a problem amounts to retrieve information. We are allowed to retrieve information through an oracle by means of questions that can be answered “yes” or “no”. The kind of questions we ask ourselves about a problem is: “For an instance of size n , in the worst case, what is the minimum number of questions we need to ask the oracle to solve our problem?”. We have no interest in the storage of information we retrieve. In other words, in our model, the only operation having a nonzero cost is asking the oracle to answer one of our “yes/no” questions.

In the first two chapters, we give an overview on two well-studied problems: Sorting and Merging. The goal of this overview is to clearly define these two basic problems as well as the model we are working with. For example, we point out the nonuniformity of the decision tree model. We also give efficient algorithms for Sorting and Merging. This overview gives us the opportunity to introduce tools that are needed all along this document. Two examples of such tools are Stirling’s approximation of the factorial [29] and the MERGE SORT algorithm [36, 58]. Since all the problems we approach are in a way or another related to Sorting and Merging, these first two chapters are the starting point of our work.

The third chapter is dedicated to formalization of commonly used objects in order theory. We first familiarize ourselves with order relations and partially ordered sets (posets). We introduce Hasse diagrams that are used to represent posets throughout this document. This chapter also contains information about the structure of certain families of posets. The conclusion of this chapter consists in the definition of graph entropy [56] and poset entropy, one of the most useful tools we use to explain the results exposed in Chapter 4.

In the fourth chapter, we tackle the problem of Sorting under Partial Information (SUPI). SUPI is a more general problem than Sorting. When we want to sort an array of integers with the QUICKSORT algorithm for

example, we assume the array is shuffled and that no prior information can be extracted from the input without asking the oracle. In SUPI we do have some initial information on the input, modeled as a poset. We explain ways to find the linear extension of a poset using a smaller number of queries than what would be required by classical sorting algorithms. We first define the problem and explain its origins. We then explain techniques that can be used to solve this problem [48, 16]. The last two sections of this chapter concern algorithms that solve the problem of Merging under Partial Information (MUPI), a special case of SUPI. We first give a detailed explanation of an algorithm due to Linial [59] and then give an original, more practical implementation of this algorithm. This new implementation solves an MUPI instance with less than $1.44 \log e(\mathcal{P})$ comparisons and has a time complexity of $O(n^2 + n \log e(\mathcal{P}))$. The n^2 term of the time complexity measure can be moved to a preprocessing phase if \mathcal{P} is known in advance.

The fifth chapter introduces a variety of problems that are all related to each other. We introduce the classical set sum problems that are 3SUM, k -SUM, and subset sum. Other problems include a generalization of k -SUM known as linear degeneracy testing, abbreviated k -LDT, and Sorting $X + Y$. We explain how one can establish connections between those problems and Sorting. We also need to upgrade our decision tree model and for this purpose we introduce the notion of k -linear decision trees. This extended model is used in the last three chapters.

The sixth chapter focuses on Sorting $X + Y$. We make multiple attempts to solve this problem efficiently and analyze its complexity using two different sets of tools. We show how exploiting known information can be significant and the limitations of the partially ordered set tools we used to solve SUPI in the fourth chapter. We detail one of the first algorithms designed to solve SUPI [33] and show that using this algorithm, one can sort $X + Y$ using fewer comparisons than with the poset theory approach.

3SUM is the subject of the seventh chapter. We introduce the subject by demonstrating the interest of studying the 3SUM problem. Then we state a long-lived but freshly refuted conjecture concerning the complexity of 3SUM, and explain the implications. The third part consists of three sections. The first two contain previous results [28, 1] while the third one exposes recent results that refute the above-mentioned conjecture [38]. Finally, we explain how these results can be applied to solve the more generic k -LDT problem.

The last chapter concerns the application of a point location algorithm to the family of k -SUM problems. We show how one can model those subset sum problems as point location problems. We then detail an efficient algorithm [63, 13] solving these newly modeled problems. The originality of our approach lies in the use of the decision tree model as the main analysis tool.

This chapter gives us the opportunity to insist again on the nonuniformity of the decision tree model. We end our dissertation with an original trick that limits the size of queries to the oracle when using our point location algorithm.

Chapter 1

Sorting

Sorting data is a well-studied problem. Results in this domain are basic while extremely important because of the numerous applications.

There exist relatively simple algorithms that do the job efficiently. Hence, it is a smooth subject to begin with in algorithmics. Moreover, it is well suited for an introductory course in the domain as it exposes various algorithmic paradigms: recursion, divide-and-conquer, parallelism, randomization; as well as mathematical concepts: enumerating permutations, orders, decision trees.

1.1 The Decision Tree Model

An important remark to begin with is that we are generally not interested in the *machine world* time complexity of a problem. As always, we have to define a model of computation that tells us which operations are allowed, which are not and what are the costs for each of them.

From here on till the end we will work in the *decision tree model*: we are allowed to ask questions to an oracle \leq that are answered “yes” or “no”. Hence, each answer gives us at most one additional bit of information. Each question asked to the oracle costs us a single unit. Every other operation can be carried out for free.

Indeed, solving a problem in our model amounts to retrieve a certain quantity of information. We need to find ways to organize the retrieved information and exploit already known information. If done efficiently, we will solve our problems without having to ask too many questions to our oracle.

The goal of each of our analyses is to show that at least a certain number of questions are required to be asked to solve a given problem, or to provide

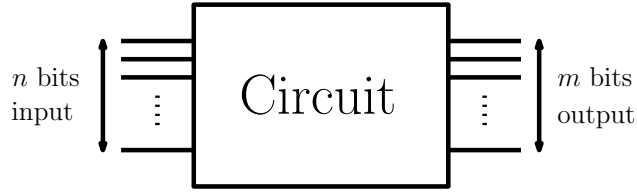


Figure 1.1: A Boolean circuit with n bits of input and m bits of output.

an algorithm solving this problem using at most a certain number of questions. Those two kinds of results are called lower bounds and upper bounds respectively. A lower bound or an upper bound is in general expressed as a function of the input size.

1.2 Nonuniformity of the Decision Tree Model

A *nonuniform model of computation* is a model where inputs of different lengths can be processed by different algorithms. In contrast with models equivalent to the Universal Turing Machine, where one algorithm can solve a problem on inputs of any size, in a nonuniform model one can have one algorithm per possible size of the input.

In our model, an algorithm is a decision tree. Since the size and shape of this tree depends on the size of the input, we have to conclude that the decision tree model is nonuniform.

Another example of such a model is circuit complexity. A *Boolean circuit*, see Figure 1.1, can be seen as an algorithm with fixed size input and fixed size output. Since we want to solve problems of any size, to each computational problem we assign a family of Boolean circuits $C_1, C_2, \dots, C_n, \dots$ that solve instances of the problem for every possible input size. We say that this circuit family is *P-uniform* if there exists a Turing machine running in time polynomial in n for which n is the input and C_n is the output.

For all the problems we study, one can make the parallel between the complexity analysis on the number of comparisons required to solve them and an analysis of their complexity in the Boolean circuit model. The goal of the next two paragraphs is to illustrate this equivalence.

As a general fact, we can design an algorithm for any problem in the decision tree model with fixed input size n and fixed output set Γ using the following approach. We generate the finite set containing all possible decision trees with $|\Gamma|$ leaves. To each node of the trees we generated corresponds a query of the form $q(x_1, \dots, x_n)$, where x_1, \dots, x_n is the input vector. We must assign queries to the nodes in a way that is consistent with the assignment on

the leaves. Moreover, a query must allow us to distinguish between at least one leaf and a nonempty set of other remaining possible outputs. Otherwise this query would have only one children and would be pointless. Note that all of this can be computed without an input. At each node we simply simulate both “yes” and “no” answers without bothering the oracle and generate the left and right subtrees according to which answer corresponds to which subtree. Since there are finitely many such decision trees, we can generate them all and keep the one with minimum height h . To solve an instance of size n we can walk through the decision tree we built, branching according to the answers we get from the oracle.

This is equivalent to building C_n , a circuit of depth h . For problems where h is polynomial in n , for each problem size we have a circuit C_n giving the correct output in polynomial time. Unfortunately, there exist such problems for which constructing C_n takes time exponential in n . Hence, the circuit families for those problems are not P-uniform. As a last remark, note that the complexity class consisting of problems for which h is at most polynomial in n while constructing C_n is at most exponential in n is $P/poly$.

1.3 Definition of the Sorting Problem

We give the definition of the sorting problem as a specification for algorithms solving this problem. If we were writing an algorithm to solve the sorting problem, the input and output conditions could be described as follows

Problem 1.1 (Sorting).

input An unknown total order \leq and a set of elements \mathcal{S} .

output The total order \leq on \mathcal{S} .

The unknown total order \leq is the oracle in our model. To solve our problem, we have to retrieve the complete information concealed by this unknown total order on \mathcal{S} . This is done by asking questions of the form $s_i \leq s_j$ to the oracle, where s_i and s_j are elements of the set \mathcal{S} .

1.4 Information-Theoretic Lower Bound

The *information-theoretic lower bound* (ITLB) of a problem is the minimal depth of any decision tree solving this problem. Generally speaking, the value of this lower bound is the logarithm¹ in base 2 of the number of feasible solutions for this problem.

¹Throughout this document, $\log x$ denotes the binary logarithm of x .

We give the ITLB for the sorting problem

Theorem 1.1 (ITLB for Sorting). *The ITLB for the sorting problem is $\log(n!)$. For any deterministic algorithm we may think of there always exists an instance of the problem that forces our algorithm to ask $\lceil \log(n!) \rceil$ questions to the oracle.*

Proof. A list \mathcal{S} of length n has $n!$ permutations. A decision tree that sorts \mathcal{S} has thus $n!$ leaves. Hence, the minimal height of such a tree is at least $\log(n!)$. \square

This lower bound is our goal to attain in terms of number of comparisons when designing an efficient algorithm solving our problem. The bound only means that we cannot do better than $\lceil \log(n!) \rceil$ comparisons, but it does not necessarily mean that there exists an algorithm or a decision tree achieving this bound in terms of complexity.

1.5 Stirling's Approximation

Later on in [Chapter 4](#) we will be interested in counting elements contained in some finite set. By counting we mean for an infinite collection of finite sets \mathcal{S}_n , where n ranges over some index set \mathcal{I} (for example, $\mathcal{I} = \mathbb{N}$), finding a counting function $f(n)$ that is valid for all \mathcal{S}_n . In general we only use the most satisfactory such formula. By *most satisfactory* we mean a completely explicit closed formula involving only well-known functions, and free from summation (and product) symbols. Unfortunately, only in rare cases will such a formula exist, and the more complicated the expression of $f(n)$ becomes the less we are willing to accept it as a determination (see Stanley [81], first chapter).

If we take the example of counting *the number of ways a pile of books can be rearranged*, that is, by permutation of the relative places of books, we can conclude that for \mathcal{S}_n being the set of all piles containing the same n books we could manage to build by rearranging these n books,

$$f(n) = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 = \prod_{k=1}^n k = n!$$

is an exact formula. However this formula contains a product over the range $[1, n]$ meaning that the computation of the value $f(n)$ would require n multiplications. In fact this is what we wanted to avoid when we said that there should not be such symbols in the most satisfactory formula.

So, is there a better way? Not always. Considering the general case, only a few rare counting functions admit a formula that satisfies us. But, for the example we gave, there is:

Theorem 1.2 (Stirling’s approximation 1730).

$$n! \simeq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Where \simeq means that the ratio of the two sides tends to 1 as n tends to infinity. This satisfactory² formula is known as Stirling’s approximation. Details about the proof can be found in Feller [29].

[Theorem 1.2](#) gives rise to several corollaries from which we retain two.

Corollary 1.3.

$$\ln n! = n \ln n - n + O(\ln n)$$

Corollary 1.4.

$$\log n! = \Theta(n \log n)$$

For example, [Corollary 1.4](#) is particularly useful for the analysis of the asymptotic complexity of sorting and merging algorithms, because of the way these problems are optimally tackled (divide and conquer).

Note that another way of looking at [Corollary 1.4](#) is saying that for some constant $c \in \mathbb{R}, c > 0$ and for some $N \in \mathbb{R}, N > 0$,

$$c \cdot n \log n \leq \log(n!) \leq n \log n, \forall n \geq N.$$

1.6 Algorithms for the Sorting Problem

We detail three algorithms that can be used to solve the sorting problem efficiently. According to the model defined in [Section 1.1](#), we only consider comparison-based sorting algorithms.

Before we start, let us make a quick observation. The sorting problem can easily be solved using at most $\binom{n}{2} = O(n^2)$ queries. A first way to achieve this bound is by iteratively computing and removing the minimum³ element from the input set. A second way is to iteratively extend a sorted list by insertion of

²We consider this formula satisfactory even though it is an approximation. Since we are interested in studying asymptotical complexity of problems and algorithms it does not matter.

³In the following paragraphs we use the words minimum, smaller and smallest as well as the symbol \leq . If one replaces minimum by maximum, smaller by larger, smallest by largest and \leq by \geq in those paragraphs, their content remains valid.

an element taken (and removed) from the input set. SELECTION SORT and BUBBLE SORT are two algorithms based on the first method. INSERTION SORT is an algorithm based on the second method. Because the complexity of these algorithms does not meet the ITLB, we need to think about more clever methods⁴.

1.6.1 Quicksort

The first efficient algorithm we explain is QUICKSORT [42]. The idea behind the QUICKSORT algorithm is to divide the input set \mathcal{S} , $|\mathcal{S}| = n$, into two disjoint subsets \mathcal{L} and \mathcal{U} that can be sorted independently from one another. To enforce this independence condition, the first part of the algorithm consists in computing \mathcal{L} and \mathcal{U} such that $l \leq u, \forall l \in \mathcal{L}, u \in \mathcal{U}$. Once \mathcal{L} and \mathcal{U} are known, the algorithm sorts these two sets recursively.

Algorithm 1.1 (QUICKSORT).

1. If $n < 2$ then \mathcal{S} is sorted and the algorithm stops.
2. Choose a pivot element $p \in \mathcal{S}$ and compare it with all other elements of \mathcal{S} by querying the oracle $n - 1$ times.
3. Partition $\mathcal{T} = \mathcal{S} \setminus \{p\}$ into two disjoint subsets $\mathcal{L} = \{t \in \mathcal{T} : t \leq p\}$ and $\mathcal{U} = \mathcal{T} \setminus \mathcal{L}$.
4. Sort \mathcal{L} .
5. Sort \mathcal{U} .

The QUICKSORT algorithm is a *divide-and-conquer algorithm*: it splits a bigger problem into several smaller problems whose solutions can be combined to build the solution of the bigger problem. Ideally, p would be chosen to be the median element of \mathcal{S} so that $|\mathcal{L}| = \lceil \frac{n-1}{2} \rceil$ and $|\mathcal{U}| = \lfloor \frac{n-1}{2} \rfloor$, because then the recursion depth of the algorithm would be bounded by $\lfloor \log |\mathcal{S}| \rfloor$ resulting in a worst-case complexity of $O(n \log n)$. However, if p can be any element of \mathcal{S} , then, in the worst case, p is the minimum (or maximum) element of \mathcal{S} , and \mathcal{L} (or \mathcal{U}) is empty. In this case, QUICKSORT behaves like the SELECTION SORT algorithm and its worst-case complexity degrades to $O(n^2)$. It is possible [6] to compute a good pivot or even the median element in $O(n)$ comparisons. However, it was shown [42] that selecting the pivot uniformly at random leads to an average case complexity of $1.39n \log n - O(n)$

⁴Although it is possible to obtain a $O(n \log n)$ comparisons algorithm by using binary search as the search method of the INSERTION SORT algorithm, this approach does not lead directly to a practical algorithm. Indeed, in the machine world, one must also take into account other types of operations. In the case of INSERTION SORT, the number of memory read and write operations in standard implementations is quadratic on average.

comparisons, and for real-world computers this approach tends to be more efficient than the safe one.

1.6.2 Merge sort

A second efficient algorithm is MERGE SORT [36, 58]. Like QUICKSORT, MERGE SORT is a divide-and-conquer algorithm. While in QUICKSORT one makes the comparisons from the partition step before solving the two subproblems, in MERGE SORT we first solve two subproblems and then recombine them with a merge step that requires $O(n)$ comparisons.

Algorithm 1.2 (MERGE SORT).

1. If $n < 2$ then \mathcal{S} is sorted and the algorithm stops.
2. Partition \mathcal{S} into two disjoint subsets \mathcal{L} and \mathcal{U} such that $|\mathcal{L}| = \lceil \frac{n}{2} \rceil$ and $|\mathcal{U}| = \lfloor \frac{n}{2} \rfloor$.
3. Sort \mathcal{L} .
4. Sort \mathcal{U} .
5. Merge \mathcal{L} and \mathcal{U} by iteratively removing the smallest element of \mathcal{S} .

Note that in step 5 there are at most $n - 1$ iterations. Since \mathcal{L} and \mathcal{U} are sorted, the smallest element of \mathcal{S} can be computed as the smallest element between the smallest element of \mathcal{L} and the smallest element of \mathcal{U} . Hence, each iteration requires a single comparison.

Unlike the previous algorithm, we always obtain a balanced partition of \mathcal{S} out of the partitioning step. Hence, the worst-case complexity of this algorithm is $O(n \log n)$. Another way to obtain this complexity measure is to solve the following recurrence relation (Sloane [80]):

$$M(n) = M(\lceil \frac{n}{2} \rceil) + M(\lfloor \frac{n}{2} \rfloor) + n - 1 \leq n \log n + n + O(\log n).$$

1.6.3 Heapsort

The last algorithm we explain is the HEAPSORT algorithm [88]. A *heap* is a data structure that allows one to efficiently lookup ($O(1)$) or remove ($O(\log n)$) the smallest element from a set \mathcal{S} . A heap can be seen as a complete binary tree where a node is smaller or equal to its children. The last condition is called the heap property.

There exist two main procedures to update the contents of a heap. The first one is SIFT-DOWN which consists in *sifting* a node down the tree until it is smaller or equal to all of its children. Below are the steps of the SIFT-DOWN algorithm with input node N

Algorithm 1.3 (SIFT-DOWN).

1. If N is a leaf of the tree the algorithm stops.
2. Compute $C = \min \text{children}(N)$.
3. If $N \leq C$ the algorithm stops.
4. Otherwise, swap C and N .
5. Sift down N again.

The second one is SIFT-UP. This operation is similar to SIFT-DOWN. We *sift* a node up the tree until it is larger or equal to its parent.

Algorithm 1.4 (SIFT-UP).

1. If N is the root of the tree the algorithm stops.
2. Compute $P = \text{parent}(N)$.
3. If $N \geq P$ the algorithm stops.
4. Otherwise, swap P and N .
5. Sift up N again.

For example, to remove the minimum from a binary heap one replaces the root node with the rightmost leaf of the complete binary tree and then sifts down the new root. To add a new node, one attaches this node to the tree to the right of the rightmost leaf (or as the leftmost node of a new level of the tree if the last level is complete) and then sifts up this new node. Since a complete tree is balanced, sifting down a node in a binary heap uses at most $2\lceil \log n \rceil$ comparisons while sifting up a node uses at most $\lceil \log n \rceil$ comparisons.

However, it is possible to build a heap using $o(n \log n)$ comparisons. Suppose that we have a complete tree that does not respect the heap property. This tree has $\lceil \frac{n}{2} \rceil$ leaves, at most $\lceil \frac{n}{4} \rceil$ nodes of height 1, at most $\lceil \frac{n}{8} \rceil$ nodes of height 2, ..., that is, at most $\lceil \frac{n}{2^{i+h}} \rceil$ nodes of height h . One applies SIFT-DOWN to each node of the tree right-to-left, bottom-to-top, starting from the rightmost leaf and ending at the root. The total number of comparisons is at most

$$2 \cdot (0 \cdot \lceil \frac{n}{2} \rceil + 1 \cdot \lceil \frac{n}{4} \rceil + 2 \cdot \lceil \frac{n}{8} \rceil + 3 \cdot \lceil \frac{n}{16} \rceil + \dots) = O(n).$$

To sort \mathcal{S} using a heap, one builds a heap with $O(n)$ comparisons and then iteratively extracts the minimum of this heap and restores the heap property with $2\lceil \log(n-1) \rceil$ comparisons. The maximum number of comparisons

involved in these extraction steps is

$$H(n) = 2 \sum_{i=2}^{n-1} \lceil \log i \rceil \leq 2(\log(n-1)! + n - 2),$$

which is approximately twice as much comparisons than what can be found by solving the recurrence relation for the MERGE SORT algorithm. Note that in the HEAPSORT algorithm we did not use the SIFT-UP operation. There exists [85] a variation of the extracting step leading to an algorithm called BOTTOM-UP HEAPSORT that uses at most $1.5 n \log n + O(n)$ comparisons.

1.6.4 Other Algorithms

There exist other interesting ideas to solve the sorting problem. For example: SHELL SORT [76], SMOOTHSORT [26], INTROSORT [69] and TIMSORT [61].

Considering practical uses of sorting algorithms, QUICKSORT is one of the most efficient algorithms on real-world computers due to its simplicity and cache locality of its operations, even though QUICKSORT has a higher multiplicative factor than MERGE SORT regarding the number of comparisons and even though QUICKSORT has a quadratic worst-case complexity. Standard libraries of the most popular programming languages (Java (up to Java 6), C, C++) use their own version of the QUICKSORT algorithm as their sorting procedure. An exception is Python which uses the TIMSORT algorithm since 2002. Note also that up to Java 6, Java used a MERGE SORT algorithm for sorting non-primitive types. The reason behind this choice is that objects in Java can be different while having the same comparison “key”. Since the Java MERGE SORT algorithm is implemented as a stable sorting procedure, it offers the users the guarantee that the sorting algorithm does not reverse the *input-order* of *equal-key* objects. The classical QUICKSORT algorithm is not used in this case because it is not stable.

Recently, there has been some interest in a dual-pivot QUICKSORT implementation by Yaroslavskiy [89]. It is not the first time this approach is considered. Previous attempts [73] had concluded that a dual-pivot QUICKSORT was worse than the classical one in practice. Analysis of the version of Yaroslavskiy [89] by Wild and Nebel [86] reveals that this new version uses $1.32n \log n - 2.46n + O(\log n)$ comparisons on average. This is less than the classical QUICKSORT algorithm which uses $1.39n \log n - 1.51n + O(\log n)$ comparisons on average. However, in practice one must also take into account other kinds of operations. For example, the number of element swaps in the dual-pivot QUICKSORT of Yaroslavskiy is twice the number of element swaps of the classic QUICKSORT by Hoare. This algorithm is used as the sorting procedure of arrays of primitive type for the Java language (since Java

7). Although it performs more swaps [86], and uses more Java bytecodes [87], the dual-pivot version of Yaroslavskiy [89] outperformed the previous QUICKSORT implementation experimentally. Note that the stable MERGE SORT algorithm of Java 7 was changed for a TIMSORT implementation, which is also stable. One of the subtleties of the TIMSORT algorithm is that it tries to exploit already sorted sublists of the input. On a nearly sorted (or reverse sorted) input the TIMSORT algorithm runs in $O(n)$ time.

A remark concerning SHELL SORT. The average time complexity of SHELL SORT is not known. Indeed, there are multiple ways of implementing this algorithm because of its definition. The SHELL SORT algorithm is configurable with a sequence of integers called a *gap sequence*. This gap sequence parameter leads to different time complexities. For example, depending on the implementation, one can achieve a complexity of either $O(n \log^2 n)$ or $O(n^{\frac{3}{2}})$. There are some practical variants of this algorithm for which the time complexity is not known. An open problem is to find whether there exists a gap sequence that makes the algorithm run in $O(n \log n)$.

In addition to classical sorting algorithms, we may also mention that any data structure allowing to retrieve inserted elements in an ordered fashion (according to some total order) with insertion and retrieve operations in $O(\log n)$ time complexity can be used to build an algorithm whose time complexity is $O(n \log n)$. Examples are BINARY TREE SORT which uses a *binary tree* and TOURNAMENT SORT which uses a *priority queue*. For example implementations of those data structures see Sleator and Tarjan [79] for the binary tree, and Leiserson et al. [58] for the priority queue.

1.6.5 Is the Sorting Problem a Solved Problem?

Although there has been an enormous amount of research directed towards sorting algorithms in the field of Computer Science, there still exists a linear term gap between the most efficient algorithms with respect to the number of comparisons and the ITLB for the sorting problem.

Out of the algorithms we explained, MERGE SORT is one of the sorting algorithms whose query complexity comes closest to the ITLB, that is,

$$n \log n - n + 1 \leq M(n) = n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \leq n \log n + n + O(\log n),$$

while

$$\text{ITLB} \simeq n \log n - 1.443n + \frac{1}{2} \log n + 1.326,$$

which leaves us with a $O(n)$ gap between the two bounds.

However, it is possible to do better. The Ford-Johnson algorithm [30, 45, 55] is the best-known sorting algorithm for this purpose. The bounds

achieved by the Ford-Johnson algorithm are

$$n \log n - 1.415n + \frac{1}{2} \log n + O(1) \leq F(n) \leq n \log n - 1.329n + \frac{1}{2} \log n + O(1),$$

where

$$F(n) = n \log n - n(1 + f + 2^{1-f} - \log 3) + \frac{1}{2} \log n + O(1)$$

and $n = \left(\frac{4}{3}\right) 2^{k+f}$ with k an integer and f a real number in the range $0 < f < 1$.

Manacher [60] proved that this algorithm is not optimal, that is, that there exists another sorting algorithm beating the Ford-Johnson algorithm in terms of number of comparisons for some inputs. The question of the existence of an optimal algorithm remains open.

Chapter 2

Merging

If we ignore the MERGE SORT algorithm which was presented in the previous chapter there are not so many examples of practical merging procedures. Indeed, this problem seems unpopular and outdated. For example, *Wikipedia* states that merging algorithms are “*a family of algorithms that run sequentially over multiple sorted lists, typically producing more sorted lists as output*”, which are not the only algorithms that we consider.

We explain our motivation with the following observation. If we want to build a sorted output list from two sorted input lists, a common, practical and simple way to do it is to read both input lists sequentially, as if they were priority queues, iteratively removing the smallest element out of the two input lists heads and adding it to the end of the output list. This algorithm is the merge procedure of the MERGE SORT algorithm and is called TAPE MERGE. TAPE MERGE has $O(n)$ complexity for both comparisons and other computation operations. However, the query complexity of TAPE MERGE does not match the ITLB of the merging problem for all inputs. In this chapter, we explain how we can beat the TAPE MERGE algorithm with cleverer techniques and how to apply these results to the problem of merging k sorted lists.

In the first section, we give a formal statement of the merging problem, that is, the problem of merging k sorted lists of different sizes. In the second section, we analyze the ITLB for the problem of merging two sorted lists of different sizes and give a description of the Hwang-Lin algorithm [46] that solves this problem using $O(\text{ITLB})$ queries. In the last section, we give the ITLB of the merging problem and a $O(\text{ITLB})$ algorithm solving this problem. This last algorithm is built on top of the Hwang-Lin algorithm.

2.1 Definition of the Merging Problem

We give the definition of the merging problem as a specification for algorithms solving this problem. An algorithm solving the merging problem must ensure the following conditions regarding its input and output

Problem 2.1 (Merging).

input *An unknown total order \leq and a set of totally ordered sets $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k\}$.*

output *A total order on $\mathcal{S}' = \cup_{i=1}^k \mathcal{S}_i$ compatible with the total orders on the \mathcal{S}_i .*

Note that there is no restriction in what order we are allowed to compare elements from \mathcal{S}_i and \mathcal{S}_j .

2.2 Merging two Totally Ordered Sets

We analyze the ITLB for the problem of merging two totally ordered sets of different sizes and give a description of a $O(\text{ITLB})$ algorithm.

2.2.1 ITLB

Analogously to what we did in [Chapter 1](#), we compute the logarithm of the number of possible solutions for this problem.

Theorem 2.1 (ITLB for the problem of merging two totally ordered sets). *The ITLB for the merging problem when $k = 2$ with $|\mathcal{S}_1| = m, |\mathcal{S}_2| = n$ is $\Omega(\log \binom{m+n}{m})$.*

Proof. We have to choose the m positions among $m+n$ in \mathcal{S}' for the elements of \mathcal{S}_1 and then fill the remaining $(m+n) - m = n$ positions with the elements of \mathcal{S}_2 . The number of leaves of the decision tree is $\binom{m+n}{m}$. Hence, the minimal height of the tree is at least $\log \binom{m+n}{m}$. \square

Note that giving $\log \binom{m+n}{m}$ in the form of Stirling's approximation when $n \geq m$ gives us $(m+n) \log(m+n) - m \log m - n \log n - O(\log n)$ which clearly expresses the information contained in the sorted list \mathcal{S}' of $m+n$ elements minus the information we already have.

2.2.2 The Hwang-Lin Algorithm

We analyze the ITLB of the Merging problem. Then we describe an algorithm whose complexity meets the ITLB.

We begin by computing the general case expression of the ITLB

Lemma 2.2.

$$\log \binom{m+n}{m} = \Theta \left(m \log \frac{m+n}{m} + n \log \frac{m+n}{n} \right)$$

Proof.

$$\begin{aligned} \log \binom{m+n}{m} &= \log \frac{(m+n)!}{m!n!} \\ &= \log(m+n)! - \log m! - \log n! \\ &= \Theta((m+n) \log(m+n) - m \log m - n \log n) \\ &= \Theta((m+n) \log(m+n) - m \log m - n \log n + (n \log m - n \log m)) \\ &= \Theta((m+n) \log(m+n) - (m+n) \log m - n(\log n - \log m)) \\ &= \Theta((m+n) \log \frac{m+n}{m} - n(\log n - \log m)) \\ &= \Theta(m \log \frac{m+n}{m} + n \log \frac{m+n}{m} - n(\log n - \log m)) \\ &= \Theta(m \log \frac{m+n}{m} + n \log \frac{m+n}{m/n \cdot n}) \\ &= \Theta(m \log \frac{m+n}{m} + n \log \frac{m+n}{n}) \end{aligned} \quad \square$$

In order to feed our intuition on this expression we analyze its behavior on several special cases. Without loss of generality, we consider the cases where $m \leq n$.

The first case we analyze is the case $m = n$. This case is similar to what happens during the merge phase of the MERGE SORT algorithm.

Lemma 2.3.

$$m = n \implies \log \binom{m+n}{m} = \Theta(m+n) = \Theta(n)$$

Proof.

$$\begin{aligned}
\log \binom{m+n}{m} &= \Theta\left(m \log \frac{m+n}{m} + n \log \frac{m+n}{n}\right) \\
&= \Theta\left(m \log \frac{2m}{m} + n \log \frac{2n}{n}\right) \\
&= \Theta(m \log 2 + n \log 2) \\
&= \Theta(m+n) \\
&= \Theta(n)
\end{aligned}
\quad \square$$

An algorithm using at most $n + m - 1$ queries is TAPE MERGE. TAPE MERGE is the merge procedure that MERGE SORT uses (see step **5** of the MERGE SORT algorithm in [Section 1.6](#)), that is, given two linearly ordered sets sort their union by iteratively removing the smallest element it contains. The union of the two linearly ordered sets contains $n + m$ elements. Hence, this algorithm runs in $n + m$ iterations. Because the algorithm only needs to choose between the smallest element of the first linearly ordered set and the smallest element of the second linearly ordered set at each iteration, each iteration uses at most one query. At the last iteration, there is only one element left. Hence, no query is used during the last iteration and the maximum number of queries this algorithm uses is $n + m - 1$.

Additionally we show what happens when $m = cn$ for some $0 < c \leq 1$.

Lemma 2.4.

$$m = cn \text{ such that } 0 < c \leq 1 \implies \log \binom{m+n}{m} = \Theta(n)$$

Proof.

$$\begin{aligned}
\log \binom{m+n}{m} &= \Theta\left(m \log \frac{m+n}{m} + n \log \frac{m+n}{n}\right) \\
&= \Theta\left(cn \log \frac{(1+c)n}{cn} + n \log \frac{(1+c)n}{n}\right) \\
&= \Theta\left((1+c)n \log(1+c)n - cn \log cn - n \log n\right) \\
&= \Theta\left((1+c)n \log(1+c)n - (1+c)n \log n - cn \log c\right) \\
&= \Theta\left((1+c)n \log \frac{(1+c)n}{n} - cn \log c\right) \\
&= \Theta\left((1+c)n \log(1+c) - cn \log c\right) \\
&= \Theta\left((1+c)n \log(1+c) + cn \log \frac{1}{c}\right) \\
&= \Theta(n)
\end{aligned}
\quad \square$$

In conclusion, the only way the ITLB could be $o(n)$ is if $m = o(n)$. This result will be useful for the next section.

We want to see what the bound becomes when n start to grow relatively to m . In the following proof we suppose that m is negligible compared to n .

Lemma 2.5.

$$m = o(n) \implies \log \binom{m+n}{m} = \Theta\left(m \log \frac{n}{m}\right)$$

Proof.

$$\begin{aligned} \log \binom{m+n}{m} &= \Theta\left(m \log \frac{m+n}{m} + n \log \frac{m+n}{n}\right) \\ &= \Theta\left(m \log \frac{n}{m} + n \log \frac{n}{n}\right) \\ &= \Theta\left(m \log \frac{n}{m}\right) \end{aligned} \quad \square$$

This could be intuitively understood as doing m binary searches on sets of size $\frac{n}{m}$.

We observe what happens when $m = 1$.

Lemma 2.6.

$$m = 1 \implies \log \binom{m+n}{m} = \Theta(\log n)$$

Proof.

$$\begin{aligned} \log \binom{m+n}{m} &= \Theta\left(m \log \frac{n}{m}\right) \\ &= \Theta(\log n) \end{aligned} \quad \square$$

An algorithm performing $O(\log n)$ queries is **BINARY SEARCH**.

In conclusion, we need an algorithm performing $O(m \log \frac{n}{m})$ queries. Ideally this algorithm should compete with Sequential Merge for the case $m = n$ and with **BINARY SEARCH** for the case $m = 1$.

So far we have explored special cases, it would be nice if we could combine the different solutions to build an algorithm of complexity $O(\log \binom{m+n}{m})$ that works for any input. The idea is that the algorithm should behave like:

- **TAPE MERGE** when $m = cn$, for some $0 < c \leq 1$;
- m binary searches on sets of size $\frac{n}{m}$ when $m = o(n)$;

- BINARY SEARCH when $m = 1$.

Hwang and Lin [46] give such an algorithm. Their algorithm merges two linearly ordered sets $\mathcal{A} = (a_1, \dots, a_m)$ and $\mathcal{B} = (b_1, \dots, b_n)$. Below is a description of the algorithm

Algorithm 2.1 (Hwang-Lin algorithm).

1. If $|\mathcal{A}| > |\mathcal{B}|$ swap the roles of \mathcal{A} and \mathcal{B} .
2. If $\mathcal{A} = \emptyset$ the algorithm stops.
3. Let $\alpha = \lfloor \log \frac{|\mathcal{B}|}{|\mathcal{A}|} \rfloor$ and $x = 2^\alpha$.
4. Compare a_1 with b_x .
5. If $a_1 \geq b_x$, remove all elements b_1, \dots, b_x from \mathcal{B} .
6. Otherwise, find the insertion position of a_1 in b_1, \dots, b_{x-1} using BINARY SEARCH. Remove a_1 from \mathcal{A} and remove from \mathcal{B} the set of all elements that lie before the insertion position of a_1 in \mathcal{B} .
7. Merge \mathcal{A} and \mathcal{B} .

In summary, the algorithm searches the position of the first element of \mathcal{A} in the 2^α smallest elements of \mathcal{B} . If a_1 is greater or equal to all these elements, which can be checked in a single comparison, then the algorithm discards these elements. Otherwise the algorithm uses BINARY SEARCH on $2^\alpha - 1$ elements to find the insertion position of a_1 in \mathcal{B} . In both cases, the last step of the algorithm is to recurse on the updated sets \mathcal{A} and \mathcal{B} . The first step of the algorithm ensures that $|\mathcal{A}| \leq |\mathcal{B}|$.

Hwang and Lin [46] prove this algorithm uses at most $\lceil \text{ITLB} \rceil + m - 1 = O(\text{ITLB})$ queries. The trick that allows to get so close to the ITLB with this algorithm resides in the choice of α . Indeed BINARY SEARCH uses at most α comparisons when $\alpha \in \mathbb{N}$ and when the number of elements to search from is $2^\alpha - 1$.

We can check that this algorithm matches the ITLB found for our special cases. For example, 2^α is *almost* $\frac{n}{m}$, which makes the Hwang-Lin algorithm match our description of a good algorithm for the case $m = o(n)$.

2.3 Merging k Totally Ordered Sets

We give the ITLB and a $O(\text{ITLB})$ algorithm for Merging. The algorithm is built on top of the Hwang-Lin algorithm.

2.3.1 ITLB

Again, we compute the ITLB as the logarithm of the number of possible solutions for this problem.

Theorem 2.7 (ITLB for Merging). *The ITLB for the merging problem when $k \geq 2$ with $|\mathcal{S}_i| = n_i$ and $n = \sum_{i=1}^k n_i$ is $\log \frac{n!}{n_1! n_2! \dots n_k!}$.*

Proof. We choose the n_1 positions among n in \mathcal{S}' for the elements of \mathcal{S}_1 and then recursively make this choice on the remaining $n - n_1$ positions with the elements of $\mathcal{S}_2 \cup \dots \cup \mathcal{S}_k$. The number of leaves of the decision tree is $\frac{n!}{n_1! n_2! \dots n_k!}$. Hence, the worst-case minimal height of the tree is $\log \frac{n!}{n_1! n_2! \dots n_k!}$. \square

Note that giving $\log \frac{n!}{n_1! n_2! \dots n_k!}$ in the form of the Stirling's approximation gives us $n \log n - \sum_{i=1}^k n_i \log n_i - O(n)$ which clearly expresses the information contained in the sorted list \mathcal{S}' of n elements minus the information we already have.

2.3.2 An Algorithm for the Merging Problem

We start with the pseudo-code description of our algorithm. We then prove this algorithm solves the merging problem using $O(\text{ITLB})$ queries.

Algorithm 2.2 (Algorithm for the merging problem).

1. Let \mathcal{S}_i and \mathcal{S}_j be two smallest sets from $\mathcal{S}_1, \dots, \mathcal{S}_k$.
2. Merge \mathcal{S}_i and \mathcal{S}_j with a $O(\text{ITLB})$ algorithm.
3. Merge $(\{\mathcal{S}_1, \dots, \mathcal{S}_k\} \setminus \{\mathcal{S}_i, \mathcal{S}_j\}) \cup \{\mathcal{S}_i \cup \mathcal{S}_j\}$.

We use tools from Information Theory to prove that this algorithm has a query complexity of $O(\text{ITLB})$. The first tool we use is Shannon's entropy of a random variable X

Definition 2.1 (Shannon's entropy).

$$H(X) \stackrel{\text{def}}{=} \sum_{x_i} p(x_i) \log \frac{1}{p(x_i)}.$$

The second is Huffman codes.

Let us take some time to expose the relation between constructions of Huffman codes and runs of our algorithm. First, we define the problem of optimal coding of a random variable with respect to the average length of the messages encoding events emitted by this random variable. We assume the channel we use to transmit our messages is perfect.

Problem 2.2 (Optimal coding of a random variable). *Given a random variable X with possible events x_1, \dots, x_n occurring with probability $0 < p(x_i) \leq 1$, $\sum_{x_i} p(x_i) = 1$, construct an instantaneous code, that is, a code where no code word is a prefix of another code word, to transmit events through a perfect channel with alphabet $\Sigma = \{0, 1\}$ that minimizes the average size of a message encoding an event.*

There are several results on this subject which we summarize in the following theorem.

Theorem 2.8 (Optimality of Huffman codes). *If we define $l(x_i)$ to be the code length for event x_i , then the average code length is*

$$L(X) = \sum_{x_i} p(x_i) l(x_i).$$

The average code length $L(X)$ of any instantaneous coding scheme cannot be less than the entropy of the random variable $H(X)$. With respect to the objective function we mentioned above, Huffman coding is an optimal instantaneous code and its average length $L_H(X)$ is bounded by

$$H(X) \leq L_H(X) \leq H(X) + 1.$$

$L_H(X)$ is achieved by constructing a Huffman tree. The method to build this tree is the following. We start with n leaves, one for each possible event x_i . We assign a weight of $p(x_i)$ to each of these leaves. The current structure is not quite a tree but a pool of disconnected nodes. We replace two nodes of the pool with smallest weight by a new one which acts as their parent in the Huffman tree. The weight of this new node is the sum of the weights of its children. We repeat this replacement operation until the pool contains only one element. This last element is the root of the tree. Since at each step we replace two nodes by one, after $n - 1$ of these steps the tree is constructed.

We make the parallel with our merging algorithm. If we want to merge multiple totally ordered sets $\mathcal{S}_1, \dots, \mathcal{S}_k$ having different lengths we can use a similar approach. To each set \mathcal{S}_i we assign a fake event with probability $\frac{|\mathcal{S}_i|}{n}$. The Shannon entropy of the random variable S having this probability

distribution is

$$\begin{aligned}
H(S) &= - \sum_i \frac{|\mathcal{S}_i|}{n} \log \frac{|\mathcal{S}_i|}{n} \\
&= - \sum_i \frac{|\mathcal{S}_i|}{n} (\log |\mathcal{S}_i| - \log n) \\
&= \sum_i \frac{|\mathcal{S}_i|}{n} \log n - \sum_i \frac{|\mathcal{S}_i|}{n} \log |\mathcal{S}_i| \\
&= \frac{1}{n} \sum_i |\mathcal{S}_i| \log n - \frac{1}{n} \sum_i |\mathcal{S}_i| \log |\mathcal{S}_i| \\
&= \frac{1}{n} (n \log n - \sum_i |\mathcal{S}_i| \log |\mathcal{S}_i|).
\end{aligned}$$

The definition of $p(\mathcal{S}_i)$ is consistent with the invariant of the Huffman tree construction algorithm since merging two totally ordered sets \mathcal{S}_i and \mathcal{S}_j produces a new one of size $|\mathcal{S}_i| + |\mathcal{S}_j|$.

We prove that for a merging algorithm that would iteratively merge the two smallest remaining totally ordered sets, the number of comparisons used is at most $nL_H(\mathcal{S})$. To make this explicit, we rearrange the terms of $L_H(\mathcal{S})$,

$$L_H(\mathcal{S}) = \sum_i \frac{|\mathcal{S}_i|}{n} l(\mathcal{S}_i) = \frac{1}{n} \sum_i l(\mathcal{S}_i) |\mathcal{S}_i|.$$

In the equation above, $l(\mathcal{S}_i)$ corresponds to the number of times elements of \mathcal{S}_i are taking part in a merge operation. When merging two totally ordered sets \mathcal{S}_i and \mathcal{S}_j the standard TAPE MERGE algorithm uses less than $|\mathcal{S}_i| + |\mathcal{S}_j|$ comparisons. Moreover, the totally ordered set associated to a node of the Huffman tree is the union of the totally ordered sets associated to u leaves of the tree. These leaves are $\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_u}$ and have lengths $|\mathcal{S}_{i_1}|, \dots, |\mathcal{S}_{i_u}|$. When two nodes are merged, it requires thus less than $|\mathcal{S}_{i_1}| + \dots + |\mathcal{S}_{i_u}| + |\mathcal{S}_{j_1}| + \dots + |\mathcal{S}_{j_v}|$ comparisons. We can rearrange these $|\mathcal{S}_{i_w}|$ and conclude that $\sum_i l(\mathcal{S}_i) |\mathcal{S}_i|$ is an upper bound on the number of comparisons used by our algorithm. Since

$$nH(\mathcal{S}) \leq nL_H(\mathcal{S}) \leq nH(\mathcal{S}) + n,$$

our algorithm requires at most $n \log n - \sum_i |\mathcal{S}_i| \log |\mathcal{S}_i| + n$ comparisons which is ITLB + $O(n)$.

To handle sublinear ITLB we need to resort to a last trick. The only way the ITLB can be $o(n)$ is if the last two nodes \mathcal{U} and \mathcal{L} we merge have sizes

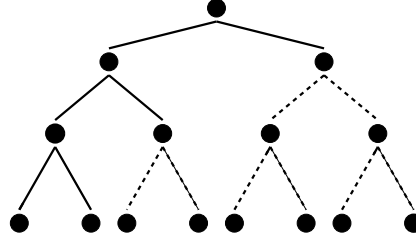


Figure 2.1: Complexity of the merging problem as a difference between the whole merge process and its subprocesses.

m and n respectively such that $m = o(n)$. Otherwise merging them would require $\Omega(n)$ comparisons. If $m = o(n)$, then we can use the Hwang-Lin algorithm to merge those two nodes using $O(m \log \frac{n}{m}) = O(\text{ITLB})$ comparisons. Moreover, the set \mathcal{L} can only be a leaf of the Huffman tree. Indeed, it cannot be the result of a merge operation between two $O(m)$ sized nodes. Since this is the only kind of merge operation that is allowed before having merged the set \mathcal{U} with another one, \mathcal{L} is necessarily a leaf. Since producing the totally ordered set of size m uses at most $O(\text{ITLB})$ comparisons using our previous algorithm, after trading TAPE MERGE for the Hwang-Lin algorithm we get an algorithm solving the merging problem using $O(\text{ITLB})$ queries.

Our algorithm is thus building the Huffman tree and then merges the two deepest leaves of the Huffman tree using the Hwang-Lin algorithm until there is only one leaf left.

We provide a visual way of understanding these results. Looking at Figure 2.1 we can deduce the complexity of the algorithm. The dashed lines in Figure 2.1 represent steps producing information we already have. Since those steps do not need to be processed, and since they normally would have required us to use $O(n \log n)$ queries, we have a total complexity of $\log n! - \sum_{i=1}^k \log n_i! = \text{ITLB}$, where n_i is the cardinality of the i -th totally ordered set of the input.

Chapter 3

Orders, Posets, Graphs, and Entropy

In the two previous chapters, we discussed orders and ordered sets without defining them. Now that we have some background on the subjects of sorting and merging it is a good time to formalize the objects we are using.

That is why we introduce new terminology¹ that allows us to identify the objects we manipulate and describe the algorithms we present in a precise and rigorous way.

3.1 Orders

This chapter discusses mathematical orders because they provide a standard way to think about our intuitive notion of order.

Orders carry information about the set of objects we manipulate. An order is a way to abstract the information that can be extracted from pairs of elements.

More formally we can look at an order by following the definition of a *binary relation*, that is, an ordered triple (X, Y, G) , where X is the domain, Y the codomain² and G the (directed) graph of the relation. We can say that elements that are contained in a binary relation R are of the type $x \in X$ is R -related to $y \in Y \iff (x, y) \in G$.

This abstraction is useful for our special cases, that is, Sorting, Merging and, in the next chapter, Sorting under Partial Information, because the only information we are interested in is the information concealed in some total order over a given finite set.

¹Most definitions can be found in Stanley [81].

²In this chapter, we usually consider $\mathcal{S} = X = Y$.

3.2 Posets

The word poset stands for *Partially Ordered Set*. Starting from what we already know, the only new term is “partial”. “Partial” can both reflect our ignorance about a more “complete”³ order on our data set and the “incomplete” nature of the ordered set we consider. By “incomplete” we mean that there could exist some $x, y \in \mathcal{S}$ for which neither $(x, y) \in G$ nor $(y, x) \in G$. We say that (x, y) is an *incomparable pair* of the partially ordered set \mathcal{S} . When however a partial order is “complete”, we say that it is a *total order*.

Definition 3.1 (Partially ordered set). A partially ordered set \mathcal{P} is a set (which by abuse of notation we also call \mathcal{P}), together with a binary relation denoted \leq (or $\leq_{\mathcal{P}}$ when there is a possibility of confusion), satisfying the following three axioms:

1. For all $t \in \mathcal{P}$, $t \leq t$ (reflexivity);
2. If $s \leq t$ and $t \leq s$, then $s = t$ (antisymmetry);
3. If $s \leq t$ and $t \leq u$, then $s \leq u$ (transitivity).

We say that two elements s and t of \mathcal{P} are comparable if $s \leq t$ or $t \leq s$; otherwise, s and t are incomparable, denoted $s \parallel t$.

We use the obvious notation $t \geq s$ to mean $s \leq t$, $s < t$ to mean $s \leq t$ and $s \neq t$, and $t > s$ to mean $s < t$.

Taking other properties aside, the most interesting feature of a partial order (with respect to our point of view) is the transitivity axiom it must satisfy. Let us explain why it is interesting. The goal of sorting and merging is to, starting from a partially ordered set, find the underlying total order. To find this total order, we are allowed to query an oracle that answers to “yes/no” questions of the type “is x R -related to y ?” in the unknown total order. This has to be done efficiently. By efficiently we mean that we do not want to ask more questions than necessary.

Before continuing further, note also that since we are trying to find a total order, at least one of the questions $(x, y) \in^? G$ and $(y, x) \in^? G$ has a positive answer. Hence, we can already discard half of all the questions we could ask.

Once we have restricted the number of useful questions to approximately half of all possible questions, there are still $O(n^2)$ questions we might want to ask. That is where transitivity comes into play. Without it we would have had to ask all the $\frac{n(n-1)}{2}$ questions to unveil the total order.

³Do not mix completeness of partial orders with the informal “completeness” we are referring to, “complete” and “incomplete” are words that help to express what we mean by partial.

For example, due to the transitivity axiom, every time we ask the two questions $(x, y) \in^? G$ and $(y, z) \in^? G$ and both give the same answer it is not necessary anymore to ask the third question $(x, z) \in^? G$.

This is what we do when we sort a list of elements efficiently with MERGE SORT or any other $O(n \log n)$ comparison sort algorithm. We exploit the transitive nature of partial orders.

3.3 Examples of Orders, Sets, and Posets

Using the following examples we try to remove any doubt one might still have on the concepts we just introduced. These examples can be found in Stanley [81]. For all examples, $[n] = \{1, 2, \dots, n\}$, \mathbb{N} is the set of natural numbers and \mathbb{P} is the set of positive numbers.

Example 3.1

- a. Let $n \in \mathbb{P}$. The set $[n]$ with its usual order, namely the ordering on natural numbers of this set, forms an n -element poset with the special property that any two elements are comparable. This poset is denoted \mathbf{n} . Of course \mathbf{n} and $[n]$ coincide as sets, but we use the notation \mathbf{n} to emphasize the order structure.
- b. Let $n \in \mathbb{N}$. We can make the set $2^{[n]}$ of all subsets of $[n]$ into a poset B_n by defining $\mathcal{S} \leq \mathcal{T}$ in B_n if $\mathcal{S} \subseteq \mathcal{T}$ as sets. One says that B_n consists of the subsets of $[n]$ “ordered by inclusion”.
- c. Let $n \in \mathbb{P}$. The set of all positive integer divisors of n can be made into a poset D_n in a natural way by defining $i \leq j$ in D_n if j is divisible by i (denoted $i \mid j$).
- d. Let $n \in \mathbb{P}$. We can make the set Π_n of all partitions of $[n]$ into a poset (also denoted Π_n) by defining $\pi \leq \sigma$ in Π_n if every part of π is contained in a part of σ . For instance, if $n = 9$, π has parts $\{1, 3, 7\}, \{2\}, \{4, 6\}, \{5, 8\}, \{9\}$, and σ has parts $\{1, 3, 4, 6, 7\}$ and $\{2, 5, 8, 9\}$, then $\pi \leq \sigma$. We then say that π is a *refinement* of σ and that Π_n consists of the partitions of $[n]$ “ordered by refinement”.

In Example 3.1.a, $[n]$ is thus the set, the “ordering on natural numbers of this set” is the order and “the conjunction of the set together with the order” is the poset. A poset is thus a relational structure (\mathcal{S}, R) and in this special case $\mathcal{S} = \mathbb{N}$ and $R = \leq$. Any two elements from this poset are comparable since the order of this poset is a (weak) total order. We say that this order is weak because it is not strict, where a strict order is a trichotomy. An

example of trichotomy is $R = <$, because then only one of xRy , $x = y$ and yRx can be true.

[Example 3.1.b](#) is a classical example to show that posets do not only work with numbers but also with more complex objects like sets. However, unlike [n](#) ([Example 3.1.a](#)), the order of this poset is not a total order. Indeed, let us consider \mathcal{S} and \mathcal{T} two subsets of $[n]$ that each contain an element of $[n]$ that is not in the other subset ($\exists s \in \mathcal{S}, s \notin \mathcal{T}$ and $\exists t \in \mathcal{T}, t \notin \mathcal{S}$). We observe $\mathcal{S} \not\subseteq \mathcal{T}$ and $\mathcal{T} \not\subseteq \mathcal{S}$ which means that \mathcal{S} and \mathcal{T} are *incomparable*, and since a totally ordered set cannot contain incomparable elements the set B_n is only partially ordered. This is a good example since we now understand what incomparable can mean for a pair of elements.

When we started to explain [Example 3.1.b](#) we said that we were working with sets of more complex objects, but this is not the reason why we found incomparable objects. With [Example 3.1.c](#) we show that even simple objects like numbers can be considered incomparable. For this to be true, we simply need to change the relation we consider. If we take for example $n = 12$, we have $D_n = \{1, 2, 3, 4, 6, 12\}$ and neither $3 \mid 4$ nor $4 \mid 3$.

[Example 3.1.d](#) is yet another example of a poset that is not total. With this example we also see that we could give a more abstract meaning to the order relation (we describe the relation with words that do not refer to a mathematical operation).

3.4 Isomorphism

When we compare two posets \mathcal{P} and \mathcal{Q} we might come to the conclusion that they “behave the same way”, that they “have the same structure”, meaning that to go from poset \mathcal{P} to poset \mathcal{Q} only a bijection-like renaming operation on the relation and the elements would have to be performed. In such a case, we say that \mathcal{P} and \mathcal{Q} are isomorphic.

Let us try to build an example. We want two posets $\mathcal{P} = (\mathcal{S}, \leq_{\mathcal{S}})$ and $\mathcal{Q} = (\mathcal{T}, \leq_{\mathcal{T}})$ such that they are intrinsically different (because the case $\mathcal{P} = \mathcal{Q}$ is trivial) while their structures look the same. If we want two posets to be different, at least one of $\mathcal{S} \neq \mathcal{T}$ or $\leq_{\mathcal{S}} \neq \leq_{\mathcal{T}}$ must be true.

For example, if we take the notations used in [Example 3.1.a](#) and we define \mathcal{P} a poset on the set $\{1, 2\}$ and \mathcal{Q} a poset on the set $\{1, 2, 3\}$ we would not be able to find a bijection between the two posets \mathcal{P} and \mathcal{Q} because the cardinality of their respective sets differ.

Since we need sets of the same cardinality, let us define the sets $\mathcal{S} = \{1, 2, 3\}$ (the set of \mathcal{P}) and $\mathcal{T} = \{1, 2, 4\}$ (the set of \mathcal{Q}). If we consider the relation $\leq_{\mathcal{S}} = \leq_{\mathcal{T}} = \leq$ we can construct a table mapping elements of \mathcal{P} to

Table 3.1: Mapping ϕ of \mathcal{P} to \mathcal{Q} .

x	$\phi(x)$
1	1
2	2
3	4

Table 3.2: Applying mapping ϕ shows that the structure of the two posets is the same.

\mathcal{P}	\mathcal{Q}
$1 \leq 1$	$1 \leq 1$
$1 \leq 2$	$1 \leq 2$
$1 \leq 3$	$1 \leq 4$
$2 \not\leq 1$	$2 \not\leq 1$
$2 \leq 2$	$2 \leq 2$
$2 \leq 3$	$2 \leq 4$
$3 \not\leq 1$	$4 \not\leq 1$
$3 \not\leq 2$	$4 \not\leq 2$
$3 \leq 3$	$4 \leq 4$

elements of \mathcal{Q} , we name this mapping ϕ , in such a way that $x \leq_{\mathcal{S}} y \iff \phi(x) \leq_{\mathcal{T}} \phi(y)$. Tables 3.1 and 3.2 prove that the construction of this mapping is possible.

Figure 3.1 shows two small diagrams⁴ for the example we just gave. Those diagrams provide an illustration of what we mean by “posets having the same structure”. Looking at a poset (\mathcal{P} or \mathcal{Q}), an edge from an upper node x to a lower node y implies that xRy holds, where R is the order of the considered poset.

⁴The diagrams of Figure 3.1 are Hasse diagrams. In those diagrams, we notice that edges that can be deduced by the transitivity property are not drawn. The topic of Hasse diagrams is covered in detail in Section 3.6.

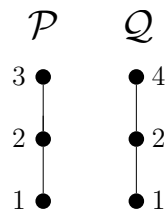


Figure 3.1: Illustration of two isomorphic posets.

We currently only gave examples of finite posets. We could show what can be achieved with infinite posets. For this example we choose $\mathcal{S} = \mathcal{T} = \mathbb{R}$, $\leq_{\mathcal{S}} = \leq$ and $\leq_{\mathcal{T}} = \geq$. We notice that by negating the relation $\leq_{\mathcal{S}}$ and thus mapping $x \in \mathbb{R}$ in \mathcal{S} to $-x$ in \mathcal{T} we get a bijection between the sets of \mathcal{S} -relations and \mathcal{T} -relations, namely for all $x, y \in \mathbb{R}$ we have $x \leq y \iff -x \geq -y$.

In general, posets must have the “same size” to be isomorphic. This remark is trivial for finite sets, but for infinite sets, countable sets cannot be isomorphic to uncountable sets and vice versa. Another general remark is that those posets must have the same bounds, for example, a closed interval cannot be isomorphic to an open interval and vice versa because a closed interval has a least and greatest element, while an open one has not (see Section 3.7 for an illustration).

We now understand what isomorphism means for posets and can give a formal definition.

Definition 3.2 (Isomorphic posets). *Two posets \mathcal{P} and \mathcal{Q} are isomorphic, denoted $\mathcal{P} \cong \mathcal{Q}$, if there exists an order-preserving bijection $\phi : \mathcal{P} \rightarrow \mathcal{Q}$ whose inverse is order-preserving; that is,*

$$s \leq t \text{ in } \mathcal{P} \iff \phi(s) \leq \phi(t) \text{ in } \mathcal{Q}.$$

For example, if $B_{\mathcal{S}}$ denotes the poset of all subsets of the set \mathcal{S} ordered by inclusion, then $B_{\mathcal{S}} \cong B_{\mathcal{T}}$ whenever $|\mathcal{S}| = |\mathcal{T}|$.

3.5 Basic Operations, Subposets, and Intervals

Since a poset is the combination $(\mathcal{S}, \leq_{\mathcal{S}})$ of a set \mathcal{S} and an order $\leq_{\mathcal{S}}$, we might be interested in applying standard set operations to posets.

The binary operations Union (\cup), Intersection (\cap) and Complement (\setminus) for posets $\mathcal{P} = (\mathcal{S}, \leq_{\mathcal{S}})$ and $\mathcal{Q} = (\mathcal{T}, \leq_{\mathcal{T}})$ only make sense when $\leq_{\mathcal{S}} = \leq_{\mathcal{T}}$. For example, when we run the algorithm MERGE SORT at each recursion step we merge (read “make the union of”) two posets having the same total order. We thus only define those operations for posets having the same order relation, and the result of any of those three binary operations on \mathcal{P} and \mathcal{Q} is a new poset $(\mathcal{U}, \leq_{\mathcal{U}})$, where $\mathcal{U} = \mathcal{S} \text{ op } \mathcal{T}$, $\text{op} \in \{\cup, \cap, \setminus\}$ and $\leq_{\mathcal{U}} = \leq_{\mathcal{S}} = \leq_{\mathcal{T}}$.

In the same way we handled the binary operations we might look at how we can define what a subposet is. For what follows, $\mathcal{P} = (\mathcal{S}, \leq_{\mathcal{S}})$ is a poset and $\mathcal{Q} = (\mathcal{T}, \leq_{\mathcal{T}})$ a subposet of \mathcal{P} .

Unlike the binary operations, we might consider the case $\leq_{\mathcal{T}} \neq \leq_{\mathcal{S}}$. Two main types of subposets emerge: subposets for which $\leq_{\mathcal{T}} = \leq_{\mathcal{S}}$ and others. The former are *induced subposets* and the latter *weak subposets*.

Definition 3.3 (Induced subposet). *By an induced subposet of \mathcal{P} , we mean a subset \mathcal{Q} of \mathcal{P} and a partial ordering of \mathcal{Q} such that for $s, t \in \mathcal{Q}$ we have $s \leq t$ in \mathcal{Q} if and only if $s \leq t$ in \mathcal{P} . We then say the subset \mathcal{Q} of \mathcal{P} has the induced order.*

This definition is similar to the definition of an induced subgraph. If we represents a poset \mathcal{P} as a directed graph G (see Section 3.10), each edge represents an element of an order relation and when we remove a vertex from one of those graphs we also have to remove incident edges since one of their endpoints is missing. Considering that \mathcal{P} is finite, there are $2^{|V(G)|}$ induced subgraphs of G , thus the poset \mathcal{P} has exactly $2^{|\mathcal{P}|}$ induced subposets. Note that when using the expression “a subposet of \mathcal{P} ”, we always mean “an induced subposet of \mathcal{P} ”.

The set of weak subposets is a superset of the set of induced subposets where arbitrary edge deletion in the graph associated with a poset is allowed. All induced subposets are also weak subposets.

Definition 3.4 (Weak subposet). *A weak subposet of \mathcal{P} is thus a subset \mathcal{Q} of the elements of \mathcal{P} and a partial ordering of \mathcal{Q} such that if $s \leq t$ in \mathcal{Q} , then $s \leq t$ in \mathcal{P} . If \mathcal{Q} is a weak subposet of \mathcal{P} with $\mathcal{P} = \mathcal{Q}$ as sets, then we call \mathcal{P} a refinement of \mathcal{Q} (only the order relations differ).*

We give the remaining definitions for the sake of completeness.

Definition 3.5 (Closed interval). *A special type of subposet of \mathcal{P} is the (closed) interval $[s, t] = \{u \in \mathcal{P} : s \leq u \leq t\}$, defined whenever $s \leq t$. The interval $[s, s]$ consists of the single point s . By this definition the empty set is not regarded as a closed interval since partial order relations are reflexive.*

Definition 3.6 (Open interval). *We give a similar definition to the open interval. The open interval $(s, t) = \{u \in \mathcal{P} : s < u < t\}$, so $(s, s) = \emptyset$.*

Definition 3.7 (Locally finite poset). *If every interval of \mathcal{P} is finite, then \mathcal{P} is a locally finite poset, for example, $\mathcal{P} = (\mathbb{N}, \leq)$ is locally finite while $\mathcal{Q} = (\mathbb{R}, \leq)$ is not.*

Definition 3.8 (Convex subposet). *We also define a subposet \mathcal{Q} of \mathcal{P} to be convex if $t \in \mathcal{Q}$ whenever $s < t < u$ in \mathcal{P} and $s, u \in \mathcal{Q}$. Thus, an interval is convex. The difference with a closed interval is that the number of minimal/maximal elements can be greater than 2.*

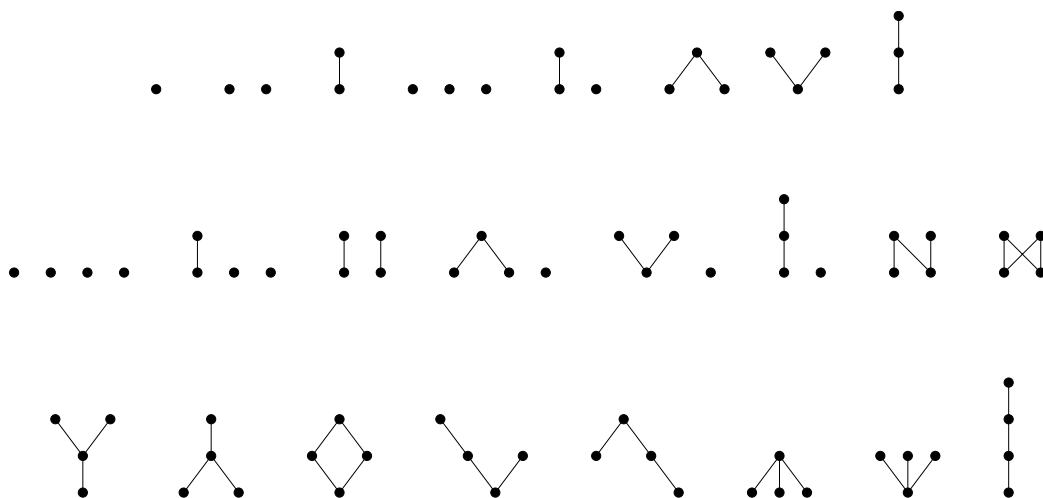


Figure 3.2: The posets with at most four elements, from Stanley [81].

3.6 Covers and Hasse Diagrams

We introduce an easy tool to “draw” a poset. A poset has an order that (by definition) has the transitivity property. We would like to exploit this transitivity property to help us simplify our drawing. What we could do is only explicitly represent the relations that are necessary to “communicate” the order. Indeed, we do not need to draw all the relations. For example, if we have a poset $\mathcal{P} = (\mathcal{S}, \leq)$ and $x, y, z \in \mathcal{S}$, writing $x < y$ and $y < z$ would imply $x < z$ which we would not need to draw.

What we draw is a *Hasse diagram*. With this diagram we only draw relations between any x and z for which there is no y such that $x < y < z$. What is drawn in a Hasse diagram is the *transitive reduction of the relation graph* and the relations contained in the drawing are the *cover relations*. Furthermore, if $x < z$ is a cover relation then x is covered by z and z covers x , and we write $x \lessdot z$ and $z \gtrdot x$.

If we take an interval (see earlier definition in [Section 3.5](#)) $I = [x, z]$ and $x < z$ is a cover relation then we have that our interval I is the set of size 2 containing x and z , that is, $I = \{x, z\}$. Note that a locally finite poset \mathcal{P} is completely determined by its cover relations.

In the Hasse diagram we make the convention that if $x < z$ then z is drawn *above* x (that is, with a higher vertical coordinate). Examples of such diagrams are shown in [Figure 3.2](#), where all posets (up to isomorphism) with at most four elements are drawn.

Some care must be taken in “recognizing” posets from their Hasse dia-

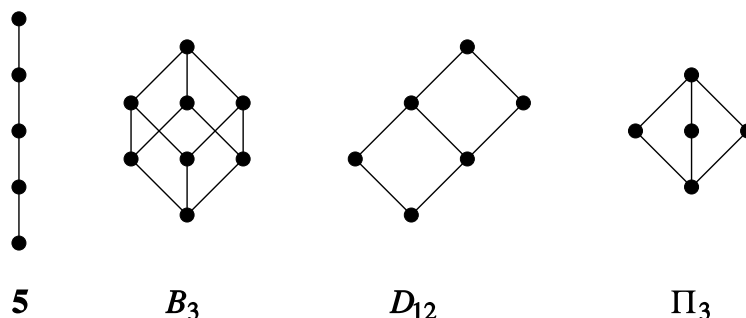



Figure 3.3: Some examples of posets, from Stanley [81].

grams. For instance,  is a perfectly valid Hasse diagram, while not drawn in Figure 3.2. This is because it is another way of drawing one that is already present in Figure 3.2 (last one on the second line). This is what we mean by *up to isomorphism*, that is, that we only consider one version of isomorphic diagrams. In fact, there are many ways to draw a Hasse diagram, and some may be better at expressing structural properties of a poset (for example, symmetries).

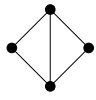
Similarly, the drawing  is not a Hasse diagram. This is because we defined earlier that the edges of the Hasse diagram were the cover relations of the poset.

Figure 3.3 illustrates the Hasse diagrams of all the posets considered in Examples 3.1.

3.7 Least and Greatest Elements

Another term that we might want to add to our vocabulary is a term that refers to an element $x \in \mathcal{P}$ for which $y \geq x, \forall y \in \mathcal{P}$. In this case, x is the *least element* of \mathcal{P} . Note that not every poset has such an element. The least element is written $\hat{0}$ and if there is a $\hat{0}$ in \mathcal{P} we say that \mathcal{P} has a $\hat{0}$. Similarly we define the *greatest element* $\hat{1}$ to be such that $x \leq \hat{1}$ for all $x \in \mathcal{P}$.

We denote by $\hat{\mathcal{P}}$ the poset obtained from \mathcal{P} by adjoining a $\hat{0}$ and $\hat{1}$ (in spite of a $\hat{0}$ or $\hat{1}$ which \mathcal{P} may already possess). In Figure 3.4 we show an example of this operation applied to some posets.

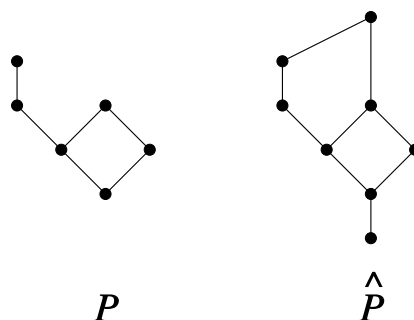
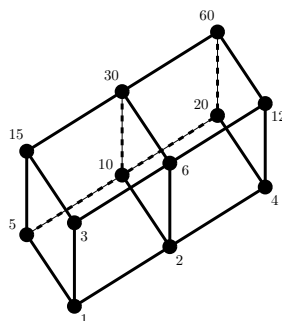
Figure 3.4: Adjoining a $\hat{0}$ and $\hat{1}$, from Stanley [81].

Figure 3.5: Hasse diagram of the set of integer divisors of 60 partially ordered by divisibility.

3.8 Infimum, Supremum, and Lattices

We introduce the notion of infimum and supremum. The *infimum* of a subset \mathcal{S} of a poset \mathcal{P} is an element l of \mathcal{P} satisfying the two following conditions: l is a lower bound of \mathcal{S} , meaning that $l \leq s$ for all $s \in \mathcal{S}$, and l is the largest such lower bound. From this definition, it is easy to deduce the definition of the *supremum* of a subset \mathcal{S} of a poset \mathcal{P} .

From these two new terms we can build the definition of a lattice. A *lattice* is a poset \mathcal{P} in which any subset \mathcal{S} that is a pair of two elements of \mathcal{P} has a unique infimum and a unique supremum. In lattice theory, infima and suprema are sometimes called *greatest lower bounds* and *least upper bounds* respectively.

An example of a lattice is the set of natural numbers (including zero) partially ordered by divisibility. In this example, the infimum and the supremum of a pair of elements are the greatest common divisor and the least common multiple respectively. Figure 3.5 illustrates this example with the integer divisors of 60.

3.9 Special Structures

We define a few special structures a poset might have and we define properties of those structures.

3.9.1 Chains and Antichains

We describe two special structures that can be found in posets. Those two structures are chains and antichains. They convey the intuitive idea of respectively *a sorted list of elements* and *a bag of incomparable elements*.

Formally, a *chain* is a poset in which any two elements are comparable. Thus, in [Example 3.1.a](#), the poset \mathbf{n} and all of its subposets are chains. A subset \mathcal{C} of a poset \mathcal{P} is a *chain* if \mathcal{C} is a chain when regarded as a subposet of \mathcal{P} .

Similarly, we define an *antichain* as a subset \mathcal{A} of a poset \mathcal{P} such that any two distinct elements of \mathcal{A} are incomparable.

The words chain and antichain have many synonyms; a chain is also called *totally ordered set* or *linearly ordered set* and an antichain is also called *Sperner family* or *clutter*.

3.9.2 Maximal and Saturated Chains

A chain \mathcal{C} of \mathcal{P} is *maximal* if it is not contained in a larger chain of \mathcal{P} . For example, in [Example 3.1.a](#), $[1, n] = \mathbf{n}$ is the only chain of \mathbf{n} that is maximal.

A chain \mathcal{C} of \mathcal{P} is *saturated* (or *unrefinable*) if there does not exist $y \in \mathcal{P} \setminus \mathcal{C}$ that we could insert in chain \mathcal{C} . If $x < y < z$ for some $x, z \in \mathcal{C}$ and $\mathcal{C} \cup \{y\}$ is a chain, then we can insert y in \mathcal{C} and \mathcal{C} is not saturated. In [Example 3.1.a](#), the poset \mathbf{n} and all of its intervals are saturated chains.

Maximal chains are thus saturated, however the converse is not true. In a locally finite poset, a chain $x_0 < x_1 < \cdots < x_n$ is saturated if and only if x_{i-1} is covered by x_i for $1 \leq i \leq n$.

3.9.3 Lengths and Ranks

Chains can be thought of as ordered lists, and this notion of list makes us want to introduce the concept of length of a chain. We define the *length of a finite chain* \mathcal{C} by $\ell(\mathcal{C}) = |\mathcal{C}| - 1$. Using this definition, we express the *length (or rank) of a finite poset* \mathcal{P} by the following formula

$$\ell(\mathcal{P}) \stackrel{\text{def}}{=} \max \{ \ell(\mathcal{C}) : \mathcal{C} \text{ is a chain of } \mathcal{P} \}.$$

The length of an interval $[x, y]$ is denoted $\ell(s, t)$. In the case where every maximal chain of \mathcal{P} has the same length n we say that \mathcal{P} is *graded of rank*

n . In this case there is a unique *rank function* $\rho: \mathcal{P} \rightarrow \{0, 1, \dots, n\}$ such that $\rho(x) = 0$ if x is a minimal element of \mathcal{P} ($x \leq y$ or $y \not\leq x$ for all $y \in \mathcal{P}$), and $\rho(t) = \rho(s) + 1$ if $t \succ s$ in \mathcal{P} . If $s \leq t$ then we also write $\rho(s, t) = \rho(t) - \rho(s) = \ell(s, t)$. If $\rho(s) = i$, then we say that s has *rank* i . This means that we can associate a rank to every element in the poset \mathcal{P} .

3.9.4 Order Ideals and Dual Order Ideals

Here we look at structures that can be intuitively understood as generated by selecting some subset \mathcal{I}' of elements of a poset \mathcal{P} and then extending this selection by iteratively including all elements $y \in \mathcal{P}$ that are either above an element of \mathcal{I}' or below an element of \mathcal{I}' . These structures are order ideals and dual order ideals, respectively.

Formally we have the following two definitions

Definition 3.9 (Order ideal). *An order ideal of \mathcal{P} is a subset \mathcal{I} of \mathcal{P} such that if $x \in \mathcal{I}$ and $y \leq x$, then $y \in \mathcal{I}$.*

Definition 3.10 (Dual order ideal). *A dual order ideal is a subset \mathcal{I} of \mathcal{P} such that if $x \in \mathcal{I}$ and $y \geq x$, then $y \in \mathcal{I}$.*

Again, many other words exist to refer to those structures. Synonyms for order ideal include *semi-ideal*, *down-set* and *decreasing subset*. Synonyms for dual order ideal include *up-set*, *increasing subset* and *filter*.

Note that, when \mathcal{P} is finite, there is a one-to-one correspondence between antichains \mathcal{A} of \mathcal{P} and order ideals \mathcal{I} . Namely, \mathcal{A} is the set of maximal elements of \mathcal{I} , while

$$\mathcal{I} = \{x \in \mathcal{P} : x \leq z \text{ for some } z \in \mathcal{A}\}. \quad (3.1)$$

The set of all order ideals of \mathcal{P} , ordered by inclusion, forms a poset denoted $J(\mathcal{P})$. If \mathcal{I} and \mathcal{A} are related as in Equation 3.1, then we say that \mathcal{A} *generates* \mathcal{I} . If $\mathcal{A} = \{z_1, \dots, z_k\}$, then we write $\mathcal{I} = \langle z_1, \dots, z_k \rangle$ for the order ideal generated by \mathcal{A} . The order ideal $\langle z \rangle$ is the *principal order ideal generated by z* , denoted Λ_z . Similarly V_z denotes the *principal dual order ideal generated by z* , that is, $V_z = \{x \in \mathcal{P} : x \geq z\}$.

3.10 Graphs, Cliques, Stable Sets, and Entropy

We focus on how to define the information a poset contains. Obviously, in our decision tree model, a totally ordered set contains maximum information while a totally unordered set contains no information at all. We have to

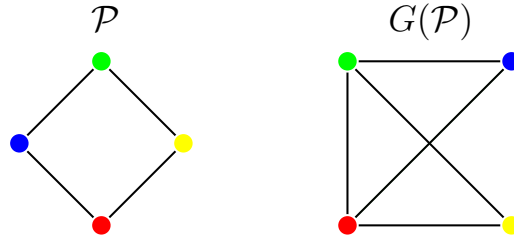


Figure 3.6: A Hasse diagram and its comparability graph.

understand it this way: If we want to sort a poset, sorting it means acquiring information on the relation \leq . If our poset is totally ordered, then we have all the information needed to describe the relation \leq completely. If the poset contains no information at all, then it means any first $x \leq^? y$ question we might ask gives us exactly 1 bit of information. In other words, we cannot deduce any $x \leq y$ relation from this poset. Hence, it is totally unordered.

3.10.1 Comparability Graphs

To help understand the structure of a poset we introduce the comparability graph $G(\mathcal{P})$ ⁵ and the incomparability graph $\tilde{G}(\mathcal{P})$ of a poset \mathcal{P} .

Definition 3.11 (Comparability graph). *The comparability graph $G(\mathcal{P})$ of a poset \mathcal{P} is an undirected graph whose vertices are the poset elements, and in which two vertices are adjacent if and only if the elements are comparable.*

Definition 3.12 (Incomparability graph). *The incomparability graph $\tilde{G}(\mathcal{P})$ of a poset \mathcal{P} is an undirected graph whose vertices are the poset elements, and in which two vertices are adjacent if and only if the elements are incomparable.*

Those graphs explicitly represent all relations (or pairs of incomparable elements) which may be inferred from the Hasse diagram of \mathcal{P} . For example, the comparability graph of a totally ordered set is the complete graph. See Figure 3.6 for a visual example.

3.10.2 Cliques and Stable Sets

We are interested in analyzing remarkable structures of comparability graphs like we did before for Hasse diagrams. Indeed, we define hereunder the counterparts of chains and antichains in posets for comparability graphs.

Definition 3.13 (Clique). *A clique \mathcal{C} of graph G is a subset of pairwise adjacent vertices in G , that is, the subgraph induced by \mathcal{C} is complete.*

⁵Note that, by abuse of notation, we write G to mean $G(\mathcal{P})$.

A clique in a comparability graph G is a subset of comparable elements in \mathcal{P} , that is, a chain in \mathcal{P} . A clique in an incomparability graph \tilde{G} is a subset of incomparable elements in \mathcal{P} , that is, an antichain in \mathcal{P} .

Definition 3.14 (Stable set). *A set \mathcal{S} is stable (or independent) if the vertices in \mathcal{S} are pairwise nonadjacent, that is, the subgraph induced by \mathcal{S} has $|\mathcal{S}|$ components.*

A stable set in a comparability graph G is a subset of incomparable elements in \mathcal{P} , that is, an antichain in \mathcal{P} . A stable set in an incomparability graph \tilde{G} is a subset of comparable elements in \mathcal{P} , that is, a chain in \mathcal{P} .

A stable set in G is a clique in \tilde{G} and vice versa.

3.10.3 STAB(G)

In what follows we show that from the internal structure of a graph G expressed using its stable sets we can deduce a discrete distribution look-alike object. With the help of this object we are able to compute the entropy of the graph G .

We look at our graph G as a “composition of the stable sets it contains”. We define $\text{STAB}(G)$ to be the set of convex combinations of stable sets of G . This set contains all convex combinations (positive, coefficients summing up to 1) of characteristic vectors $\chi^{\mathcal{S}}$, where \mathcal{S} is a stable set. The characteristic vector $\chi^{\mathcal{S}}$ is a point in \mathbb{R}^n where

$$\chi_v^{\mathcal{S}} = \begin{cases} 1, & \text{if } v \in \mathcal{S} \\ 0, & \text{otherwise.} \end{cases}$$

In other words $\chi^{\mathcal{S}}$ is a binary inclusion/exclusion representation of the stable set \mathcal{S} .

$$\text{STAB}(G) \stackrel{\text{def}}{=} \text{conv} \{ \chi^{\mathcal{S}} \in \mathbb{R}^{V(G)} : \mathcal{S} \text{ stable set in } G \} \quad (3.2)$$

Formally, Equation 3.2 is the definition of the *stable set polytope* of an arbitrary graph G with vertex set $V(G) = \{v_1, \dots, v_n\}$ and order n ($\mathbb{R}^{V(G)} = \mathbb{R}^n, n = |V(G)|$) that is, the convex hull (an n -dimensional polytope) of stable set characteristic vectors $\chi^{\mathcal{S}} \in \mathbb{R}^n$.

3.10.4 Entropy for Comparability Graphs

The entropy of a graph can be defined in several ways, see Mowshowitz and Dehmer [68] and Simonyi [77].

Here we define the entropy of a comparability graph G as a function of the stable sets G possesses. In order to do so, we look at every vector in

$\text{STAB}(G)$ and we keep the vector that gives the lowest entropy value for a given entropy function. This vector is the discrete distribution look-alike object we discussed in the previous subsection and the entropy value is called the entropy of graph G , denoted $H(G)$.

Formally, we define the entropy of comparability graph G as

Definition 3.15 (Entropy of a graph).

$$H(G) \stackrel{\text{def}}{=} \min_{x \in \text{STAB}(G)} -\frac{1}{n} \sum_{v \in V(G)} \log x_v. \quad (3.3)$$

Note that any x^* which minimizes Equation 3.3 also minimizes the sum on $\log \frac{1}{x_v}$.

The definition of entropy of graphs was first introduced by Körner [56]. The definition we use is equivalent and is due to Csiszár et al. [23].

3.10.5 Entropy for Posets

As we defined $H(G)$ in Equation 3.3, we write $H(\mathcal{P})$ to mean $H(G(\mathcal{P}))$. We give a similar definition for $H(\tilde{\mathcal{P}})$.

Definition 3.16 (Entropy of a poset).

$$H(\mathcal{P}) \stackrel{\text{def}}{=} H(G(\mathcal{P})).$$

Definition 3.17 (Entropy of the incomparability graph of a poset).

$$H(\tilde{\mathcal{P}}) \stackrel{\text{def}}{=} H(\tilde{G}(\mathcal{P})).$$

Note that since comparability graphs are perfect [5], we have

Theorem 3.1 (Complementarity of entropies for posets). *For any poset of order n ,*

$$H(\mathcal{P}) + H(\tilde{\mathcal{P}}) = \log n.$$

3.11 Linear Extensions and Entropy

A *linear extension* of a partial order \leq is a total order \leq^* compatible with \leq . *Linear* means that the order \leq^* must be total while *extension* means that the original order \leq is “preserved” that is, when $x \leq^* y$ for all pair (x, y) such that $x \leq y$.

In our decision tree model, the number of linear extensions $e(\mathcal{P})$ of a poset \mathcal{P} is strongly related to the information contained in this poset. If we want to

sort \mathcal{P} for example, the total information we need is $\log n!$, the information we have is $\log e(\mathcal{P})$, and the information we miss is $\log n! - \log e(\mathcal{P})$.

Computing $e(\mathcal{P})$ in the general case is #P-complete (see Brightwell and Winkler [9]). In the next chapter we discuss a problem called Sorting under Partial Information. This problem can be solved using O(ITLB) queries by an algorithm that iteratively chooses a “good” query. We will see later that a “good” query $a \leq? b$ can be found by computing the ratio of $e(\mathcal{P}(a \leq b))$ to $e(\mathcal{P})$ for all (a, b) pairs, where $\mathcal{P}(a \leq b)$ denotes the poset obtained from \mathcal{P} after adding the constraint that $a \leq b$. Since it is unlikely that a polynomial-time algorithm for computing $e(\mathcal{P})$ exists, it is unlikely that the algorithm sketched above can be implemented so that it runs in time polynomial in the input size.

Fortunately, Kahn and Kim [48] give a good approximation of $e(\mathcal{P})$ using the definition of entropy of a poset we introduced earlier.

Theorem 3.2 (Kahn and Kim [48]). *For any poset \mathcal{P} of order n ,*

$$\log e(\mathcal{P}) \leq nH(\tilde{\mathcal{P}}) \leq \min \{ \log e(\mathcal{P}) + \log e \cdot n, c_1 \log e(\mathcal{P}) \},$$

where $c_1 = (1 + 7 \log e) \approx 11.1$.

It is a good approximation because we have the guarantee that $nH(\tilde{\mathcal{P}}) = \Theta(\log e(\mathcal{P}))$ and because we can compute $H(\tilde{\mathcal{P}})$ in polynomial time by solving the convex minimization problem given by Equation 3.3.

Cardinal et al. [16] improve the results of Theorem 3.2 by showing that one can take $c_1 = 2$, which is best possible.

Theorem 3.3 (Cardinal et al. [16]). *For any poset \mathcal{P} of order n ,*

$$nH(\tilde{\mathcal{P}}) \leq 2 \log e(\mathcal{P}).$$

In the next chapter we will see what kind of algorithms can be built upon these results.

Chapter 4

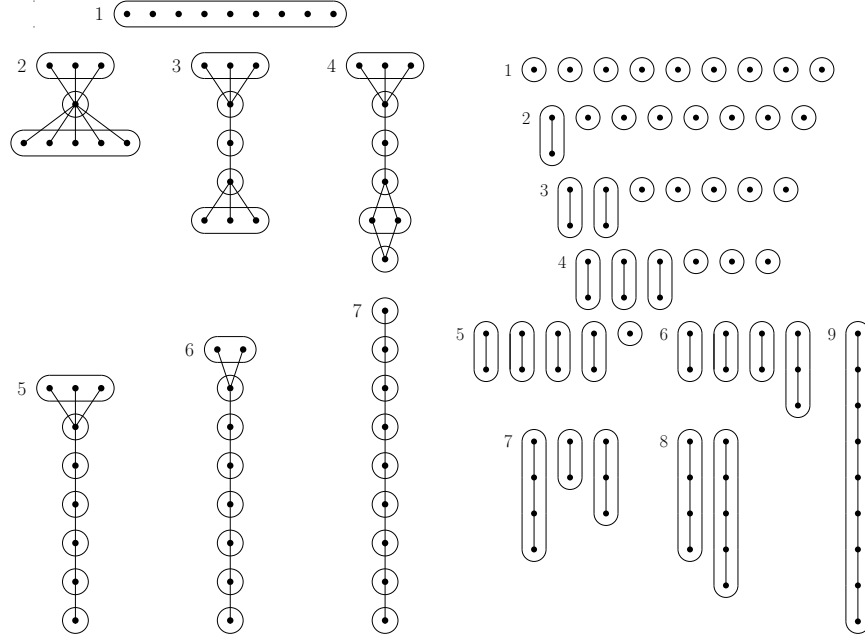
Sorting under Partial Information

We would like to generalize the sorting problem. What if, when sorting a set of elements, we already had some information on the order of these elements?

It is clear that we could use this information to ease the overall computation process. If we look at any $O(\text{ITLB})$ algorithm, they all start from 0 bits of information and end up producing a totally ordered set containing $\text{ITLB} = O(n \log n)$ bits of information. We already showed in [Section 1.4](#) that for any algorithm, there exists an instance of the problem which forces the algorithm to ask $\Omega(n \log n)$ questions. Starting from a poset whose order is completely unknown, such an algorithm thus iteratively updates this poset by retrieving information and make sure it reuses it well. By reusing the information efficiently, it is able to choose “good” questions to ask.

In QUICKSORT this is achieved by recursively building linear extensions of weak orders. In weak orders, elements of a poset are layered in sets of incomparable elements (maximal stable sets) that are disjoint from each other. Thus, QUICKSORT never has to ask a question about elements from disjoint sets. However, the way QUICKSORT chooses questions does not guarantee that every step of the algorithm evenly divides stable sets. In conclusion, while QUICKSORT can sometimes ask poor quality questions, it also reuses information it has optimally.

We take as a second example the algorithm MERGE SORT. MERGE SORT works by iteratively merging totally ordered sets \mathcal{P} and \mathcal{Q} , and for the sake of simplicity we consider that those sets have the same cardinality n (in reality their cardinalities can only differ by 1). We saw in [Section 2.2](#) that each of these steps can be computed with at most $2n - 1$ queries. For each merge operation, the input contains $2 \log n!$ bits of information while the output contains $\log(2n)!$ bits of information, so we must find $\log(2n)! -$



(a) Steps of the QUICKSORT algorithm represented as Hasse diagrams (maximal antichains / stable sets are circled).

(b) Steps of the MERGE SORT algorithm represented as Hasse diagrams (maximal chains / cliques are circled).

$2 \log n!$ bits of information. For asymptotically large n , it corresponds to approximately $2n - \frac{1}{2} \log n - 0.826$ bits of information, which is consistent with our previous results. So, regarding reuse of retrieved information the MERGE SORT algorithm is asymptotically optimal.

Figures 4.1a and 4.1b show the iterative poset generation processes of a run of QUICKSORT and MERGE SORT. They sum up what we observed about those algorithms, that they only work on a particular subset of posets.

What we are interested in in this chapter are algorithms that can sort any poset, that is, not only those handled by classical sorting algorithms, optimally with respect to the number of comparisons.

4.1 Definition of the Problem

We already explained the idea of the problem in the introduction, hereunder we formally define Sorting under Partial Information (SUPI). We quote the definition from Cardinal et al. [16]

Problem 4.1 (Sorting under Partial Information). *Let $\mathcal{S} = \{s_1, \dots, s_n\}$ be a set equipped with an unknown linear order. Given a subset of the relations*

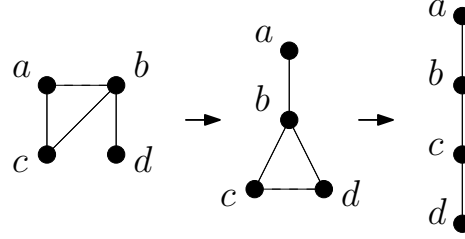


Figure 4.2: An instance of the problem of Sorting under Partial Information. In this example, we use 2 comparisons (dashed edges). At every step, the Hasse diagram of the currently known partial order is shown.

$s_i \leq s_j$, determine the complete linear order by queries of the form $s_i \leq^? s_j$.

Figure 4.2 shows an example of an instance of Sorting under Partial Information. Note that in this example we only ask the minimal number of questions needed to solve the problem. We could have asked $a \leq^? d$ as the first question, but then we would have still had to ask two more questions.

4.2 History

The first appearance of this problem dates back to 1976. In Fredman [33], an algorithm making $O(\log e(\mathcal{P}) + 2n)$ queries was featured. Fredman applies this algorithm to a problem he attributes to Elwyn Berlekamp: sorting the set $X + Y = \{x + y : x \in X, y \in Y\}$. However, since $\log e(\mathcal{P})$ can be quite small, this algorithm was not matching the ITLB for $\log e(\mathcal{P})$ sublinear in n . Moreover, the algorithm needs exponential time to choose the right comparison to perform.

From that point, further research had to be made in order to get an algorithm needing only $O(\log e(\mathcal{P}))$ comparisons as well as an algorithm running in polynomial time.

In Kahn and Saks [50], it is shown that for any finite poset \mathcal{P} there always exists a query of the form $a \leq^? b$ with $a, b \in \mathcal{P}$ such that the fraction of linear extensions in which a precedes b lies in the interval $(3/11, 8/11)$. This is a relaxation of the well-known $1/3$ – $2/3$ conjecture, a conjecture formulated independently by Fredman, Linial, and Stanley, see Linial [59]. Note that, a simpler proof yielding weaker bounds was given by Kahn and Linial [49] and better bounds were later given by Brightwell et al. [10], and Brightwell [8]. In fact, the only requirement for a practical use (without proving the conjecture) was to prove that it was true for an interval $(q, 1-q)$, because then iteratively choosing such a comparison yields an algorithm that performs $O(\log e(\mathcal{P}))$ comparisons (precisely $\log_{(1-q)^{-1}} e(\mathcal{P})$). However, the algorithm proposed by

Kahn and Saks [50] does not find this good query in polynomial time.

For the sake of completeness, we hereunder state the $1/3$ – $2/3$ conjecture.

Conjecture 4.1. *In any finite poset \mathcal{P} that is not totally ordered, it is always possible to find a pair (a, b) of elements of \mathcal{P} such that the number of linear extensions of \mathcal{P} in which a comes before b is between $1/3$ and $2/3$ of the total number of linear extensions of \mathcal{P} .*

The question of its correctness remains open and the $1/3$ bound would be best possible. Linial [59] gives a proof for width-2 posets. Also, there are results due to Peczarski [71] for some classes of partial orders verifying the conjecture showing that the proportion of partial orders contained in those classes approaches 1 as n grows. We can conclude that the proportion of n -element partial orders obeying the $1/3$ – $2/3$ conjecture approaches 1 as n tends to infinity.

Recent findings in the domain include Zaguia [90] and Peczarski [72].

4.3 Algorithms

Kahn and Kim [48] gave the first polynomial-time algorithm achieving the bound $O(\log e(\mathcal{P}))$ in 1995. The idea of their algorithm is to solve a convex optimization problem at each step to find a good query to perform. The answer to a good query reduces the entropy $H(\tilde{\mathcal{P}})$ by at least c/n , where c is at least $\log(1 + 17/112) \approx 0.2$. Kahn and Kim [48] show that a good query always exists (provided \mathcal{P} is not already sorted). This guarantees that the algorithm uses $O(nH(\tilde{\mathcal{P}}))$ queries. By Theorem 3.2 we have that $nH(\tilde{\mathcal{P}})$ approximates $e(\mathcal{P})$ up to a constant factor and thus the algorithm uses $O(\log e(\mathcal{P}))$ queries. Kahn and Kim [48] show that a good query can be found in $O(n^2)$ time by looking at the primal solution given by a convex optimization algorithm when computing $H(\mathcal{P})$ (using Equation 3.3 as the objective function). The algorithm completes after $O(\log e(\mathcal{P})) = O(n \log n)$ steps. Since a convex optimization problem can be solved in polynomial time, the algorithm runs in polynomial time. However, this algorithm is not practical as (for the moment) it requires to use the ellipsoid algorithm as a subroutine.

Finally, in 2013, Cardinal et al. [16] provide new methods. Three new algorithms are studied. Those three algorithms all have the same canvas: a first polynomial-time preprocessing phase followed by a $O(\log e(\mathcal{P}))$ query phase. Caution has to be made though: only the last algorithm matches the ITLB for all SUPI instances.

Table 4.1 highlights the properties of the different algorithms, that is,

Table 4.1: We denote by $\text{EA}(n)$ the time needed for the ellipsoid algorithm to compute the entropy of a poset of order n . The original bound given by Kahn and Kim [48] on the number of comparisons performed by their algorithm is $54.45 \cdot \log e(\mathcal{P})$. The improved bound given in the table is a byproduct of the results of Cardinal et al. [16]. The notation $O_\epsilon(n)$ means that the hidden constant may depend on ϵ .

Algorithm	Global Complexity	Number of comparisons
Kahn and Kim [48]	$O(n \log n \cdot \text{EA}(n))$	$\leq 9.82 \cdot \log e(\mathcal{P})$
Cardinal et al. [16] 1	$O(n^2)$	$O(\log n \cdot \log e(\mathcal{P}))$
Cardinal et al. [16] 2	$O(n^{2.5})$	$\leq (1 + \epsilon) \log e(\mathcal{P}) + O_\epsilon(n)$
Cardinal et al. [16] 3	$O(n^{2.5})$	$\leq 15.09 \cdot \log e(\mathcal{P})$

- If $\log e(\mathcal{P})$ is super-linear in n , the number of comparisons of Cardinal et al. [16] **2** is lower than that of Kahn and Kim [48]. By optimizing over ϵ , it can be shown that the number of comparisons is actually $\log e(\mathcal{P}) + o(\log e(\mathcal{P})) + O(n)$ in this case, a number of comparisons comparable to that of Fredman’s algorithm;
- If $\log e(\mathcal{P})$ is linear or sub-linear in n , the number of comparisons of Cardinal et al. [16] **3** is comparable to that of Kahn and Kim [48], although the constant in front of $\log e(\mathcal{P})$ is still far from the best constant achieved by a super-polynomial algorithm via balancing pairs (see Brightwell et al. [10] and Brightwell [8]);
- Algorithms from Cardinal et al. [16] have the following useful property: they compute information that guides the sorting and can then be reused to solve any given instance with the same partial information \mathcal{P} , in time proportional to the number of comparisons, plus a term linear in n .

The idea of Cardinal et al. [16] is to remove some information from the input poset to produce a weak subposet \mathcal{P}' (whose refinement is \mathcal{P}) for which finding a linear extension can be done using $O(\log e(\mathcal{P}'))$ queries with a specialized algorithm and whose entropy is not too far from the entropy of the original poset. Since $nH(\tilde{\mathcal{P}}) = \Theta(\log e(\mathcal{P}))$, as it was proved by Kahn and Kim [48], one can analyze what “not too far” means for a given information removal technique.

For example, we can iteratively extract a maximum chain of the input poset computing a so-called greedy chain decomposition. Once this is done,

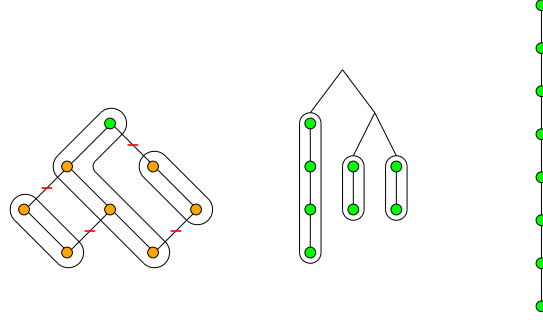


Figure 4.3: Illustration of the second algorithm in Cardinal et al. [16].

we erase all the relations of the input existing between elements of two different chains of the decomposition. One can then merge the chains using the optimal merging algorithm we described in Section 2.3. Figure 4.3 illustrates this technique and is in fact a description of Algorithm 2 of Cardinal et al. [16]. The two other algorithms of Cardinal et al. [16] are variants of this approach.

The same idea was already present in Cardinal et al. [15]. Then, the problem to solve was Partial Order Production which is the complementary problem to Sorting under Partial Information. In this case, we look at antichains rather than chains of the input poset and instead of discarding known information we ask the algorithm to retrieve more information than required. The relation between those problems is explained in detail in [14].

Note that randomized algorithms also exist for this purpose. The idea is to evaluate the quality of a query $x \leq^? y$ by generating a sample of possible linear extensions, count the number of times $x \leq y$ in the sample and then only ask the question if the ratio of this count over the size of the sample lies within a fixed interval $[q, 1 - q]$, $0 < q \leq \frac{1}{3}$. See Huber [43] for more.

4.4 Linial's Algorithm

A useful procedure is one that merges two linearly ordered sets without discarding the information existing between elements of these two sets. In Cardinal et al. [16] an implementation of such a procedure is defined. Their implementation allows one to concentrate the computational complexity in the preprocessing phase of their other algorithms. We detail another possible implementation of such a procedure that is due to Linial [59]. Unfortunately, in this implementation the computational complexity cannot be split into a preprocessing phase and a query phase. Nevertheless, it is still interesting to understand the ideas behind it.

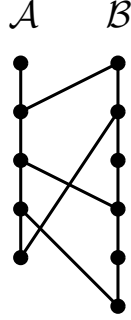


Figure 4.4: A poset \mathcal{P} covered by two chains \mathcal{A} and \mathcal{B} , input of the MUPI problem.

This procedure solves the problem of Merging under Partial Information (MUPI). Note that MUPI is a special case of SUPI. The difference with SUPI is that together with the input poset \mathcal{P} we are given two chains $\mathcal{A} = (a_1 < \dots < a_m)$ and $\mathcal{B} = (b_1 < \dots < b_n)$ such that $\mathcal{A} \cup \mathcal{B} = \mathcal{P}$, that is, $\{\mathcal{A}, \mathcal{B}\}$ is a chain cover of \mathcal{P} , implying that \mathcal{P} is of width 2. We give a formal definition of this problem

Problem 4.2 (Merging under Partial Information). *Given a poset \mathcal{P} and two disjoint chains \mathcal{A} and \mathcal{B} of \mathcal{P} such that $\mathcal{A} \cup \mathcal{B} = \mathcal{P}$, find a linear extension of \mathcal{P} .*

Linial [59] first proves that the $1/3$ – $2/3$ conjecture holds for width-2 posets, that is, posets that can be covered by two chains. Since one can always find a good query that when answered invalidates at least one-third of the possible linear extensions of \mathcal{P} , it is possible to design a $O(\log e(\mathcal{P}))$ algorithm that solves the MUPI problem.

Theorem 4.1 (Linial [59]). *Given a poset \mathcal{P} covered by two chains \mathcal{A} and \mathcal{B} , we can always find a query $x \leq^? y$ with $x \in \mathcal{A}, y \in \mathcal{B}$ such that the probability that $x \leq y$ lies in the interval $[1/3, 2/3]$.*

Moreover, in his proof, Linial [59] shows that if we consider that a_1 and b_1 are incomparable and $\frac{e(\mathcal{P}(a_1 < b_1))}{e(\mathcal{P})} < 1/3$ then there must exist an r for which either

$$1/3 \leq \frac{e(\mathcal{P}(a_1 < b_{r-1}))}{e(\mathcal{P})} \leq 1/2,$$

or

$$1/2 \leq \frac{e(\mathcal{P}(a_1 < b_r))}{e(\mathcal{P})} \leq 2/3.$$

All of this is without loss of generality. If a_1 and b_1 are comparable then either a_1 or b_1 is the unique minimal element of \mathcal{P} . Hence, this minimal

element can be removed. We can do so until a_1 and b_1 are incomparable. If $\frac{e(\mathcal{P}(a_1 < b_1))}{e(\mathcal{P})} > 2/3$ we can simply revert the roles of \mathcal{A} and \mathcal{B} and if $1/3 \leq \frac{e(\mathcal{P}(a_1 < b_1))}{e(\mathcal{P})} \leq 2/3$ then we simply do not have to search any further. Also, one can build the same proof using a_m and b_n instead of a_1 and b_1 .

Linial [59] proposes the following polynomial-time algorithm. We compute the number of linear extensions of the width-2 poset \mathcal{P} using the determinant counting formula (which we explain later). We do so also for every query we could use to retrieve more information on the total order \leq . We then use those counts to find a pair (x, y) that satisfies $1/3 \leq \frac{e(\mathcal{P}(x \leq y))}{e(\mathcal{P})} \leq 2/3$. Once this pair is identified, we can effectively make the query and update poset \mathcal{P} with the actual answer. We can then recurse on the newly obtained poset. Because of the successive uses of good queries, the recursion depth, and thus the number of queries made, is $O(\log e(\mathcal{P}))$.

The proof for the determinant counting formula Linial [59] uses can be found in Mohanty [66], we give hereunder the statement of this formula in the context of Merging under Partial Information.

Theorem 4.2 (Mohanty [66]). *Let $\mathcal{P} = \mathcal{A} \cup \mathcal{B}$, where $\mathcal{A} = (a_1 < \dots < a_m)$ and $\mathcal{B} = (b_1 < \dots < b_n)$, and assume $m \geq n$ without loss of generality. Define the integers $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_m$ as follows, $\beta_i = \min\{t : b_t > a_i\}$, $\alpha_j = \max\{t : b_t < a_j\}$ and where the minimum and maximum of an empty set are taken to be $n + 1$ and 0 respectively. Let $\binom{n}{k}_+$ be defined as*

$$\binom{n}{k}_+ = \begin{cases} 0 & \text{if } n < 0 \text{ or } k < 0 \text{ or } k > n \\ 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n}{k} & \text{otherwise} \end{cases},$$

then the number of linear extensions of \mathcal{P} is given by

$$e(\mathcal{P}) = \left| \binom{\beta_i - \alpha_j}{j - i - 1}_+ \right|_{1 \leq i, j \leq m},$$

the determinant of a $m \times m$ matrix.

Mohanty [66] gives this formula in the context of counting lattice paths. Let us explain. We want to count the number of paths that go from $(0, 0)$ to (m, n) in a $(m + 1) \times (n + 1)$ grid. Such a path is a list of $m + n + 1$ integer positions $((i_0, j_0), (i_1, j_1), \dots, (i_{m+n}, j_{m+n}))$ such that $i_k \geq i_{k-1}$, $j_k \geq j_{k-1}$ and $i_k + j_k = i_{k-1} + j_{k-1} + 1$ for all $1 \leq k \leq m + n$. Obviously $(i_0, j_0) = (0, 0)$ and $(i_{m+n}, j_{m+n}) = (m, n)$. With these constraints alone we obtain a count of $\binom{m+n}{n} = \binom{m+n}{m}$ different paths¹. Mohanty [66] adds the constraint that

¹Note that if $m = n$ and if we add the condition that $i_k \leq j_k \forall 1 \leq k \leq m + n$ then the path count corresponds to the n^{th} Catalan number.

$\alpha_i \leq j < \beta_i$ for all path positions (i, j) with $i \neq 0$, where $0 \leq \alpha_1 \leq \dots \leq \alpha_m \leq n+1$, $0 \leq \beta_1 \leq \dots \leq \beta_m \leq n+1$ and $\alpha_i < \beta_i \forall 1 \leq i \leq m$ and finds this determinant counting formula as an answer. The reason the cases $i = 0$ can be ignored is because the constraints explained above suffice to fix the possible values that j can take when $i = 0$.

This problem of counting paths is equivalent to counting the number of possible outcomes of a merging algorithm on input $(\mathcal{S}_1, \mathcal{S}_2)$, $|\mathcal{S}_1| = m$, $|\mathcal{S}_2| = n$, provided we know the insertion position j of the i^{th} element of \mathcal{S}_1 into \mathcal{S}_2 is bounded by $\alpha_i \leq j < \beta_i$.

We make a few observations to show that Linial's algorithm runs in polynomial time:

1. The values $\beta_i - \alpha_j$ and $j - i + 1$ are bounded linearly in the size of the input. Hence, their size is bounded logarithmically in the size of the input;
2. The size of the value of the binomial coefficient $\binom{n}{k}$ is bounded by a polynomial in k and n ;
3. Using Bareiss algorithm [2], the computation of a determinant can be done in polynomial time. Moreover, the size of all computed values is bounded by some polynomial in the size of the matrix and in the starting size of the values;
4. For each step there is a polynomial number of possible queries to look for;
5. There are $O(\log e(\mathcal{P}))$ steps and $\log e(\mathcal{P})$ is $O(n \log n)$.

However, the algorithm *as is* is not practical. The complexity added by the computation of binomial coefficients matrices and determinants in Linial's algorithm is at least quadratic. In the next section, we present an original implementation of this algorithm that uses dynamic programming to avoid the computation of binomial coefficients and determinants.

4.5 Efficient Implementation of Linial's Algorithm

We present original developments on Linial's work [59]. We explain how to reduce the complexity of Linial's algorithm to $O(n^2)$ preprocessing time and $O(n \log e(P))$ running time.

As an anonymous referee pointed out in the peer review of Cardinal et al. [16], it is possible to implement Linial's algorithm using dynamic programming instead of binomial coefficients and determinants. We expose a recurrence relation that can be used to compute $e(\mathcal{P})$ using dynamic programming.

We look at a_1 and b_1 , the smallest elements of \mathcal{A} and \mathcal{B} respectively. There are three ways a_1 and b_1 can be related in the partial order \mathcal{P} we receive as input. Either we know $a_1 <_{\mathcal{P}} b_1$ or $b_1 <_{\mathcal{P}} a_1$, or a_1 and b_1 are incomparable in \mathcal{P} . In the first two cases, we know that one of a_1, b_1 must be the smallest element of the set $\mathcal{A} \cup \mathcal{B}$ when totally ordered. In these cases, $e(\mathcal{P}) = e(\mathcal{P} \setminus a_1)$ or $e(\mathcal{P}) = e(\mathcal{P} \setminus b_1)$ for $a_1 < b_1$ or $b_1 < a_1$ respectively. In the last case, if a_1 and b_1 are incomparable in \mathcal{P} , in the unknown total order we are trying to unveil one of $a_1 < b_1$ or $b_1 < a_1$ holds true. Hence, $e(\mathcal{P}) = e(\mathcal{P} \setminus a_1) + e(\mathcal{P} \setminus b_1)$ in this case, since both outcomes for $a <^? b$ are possible. Lastly, if $|\mathcal{A}| = 0$ or $|\mathcal{B}| = 0$ then $e(\mathcal{P}) = 1$. We can thus express $e(\mathcal{P})$ using the following recurrence relation

$$e(\mathcal{P}) = \begin{cases} 1 & \text{if } |\mathcal{A}| = 0 \text{ or } |\mathcal{B}| = 0 \\ e(\mathcal{P} \setminus a_1) & \text{if } a_1 <_{\mathcal{P}} b_1 \\ e(\mathcal{P} \setminus b_1) & \text{if } b_1 <_{\mathcal{P}} a_1 \\ e(\mathcal{P} \setminus a_1) + e(\mathcal{P} \setminus b_1) & \text{if } a_1 \text{ and } b_1 \text{ are incomparable in } \mathcal{P}, \end{cases}$$

which can be computed using dynamic programming. Note that posets \mathcal{Q} that are incompatible with the original poset \mathcal{P} , are assigned a value $e(\mathcal{Q}) = 0$.

If we compute $e(\mathcal{P})$ using this recurrence relation in a dynamic program, we also get all values of $e(\mathcal{P}(a_1 < b_r))$ and $e(\mathcal{P}(b_1 < a_r))$ for free. Without loss of generality we prove that this holds for all values $e(\mathcal{P}(a_1 < b_r))$, $1 \leq r \leq |\mathcal{B}|$.

Proof. Let K be the tableau of our dynamic program, that is,

$$K = \{K_{i,j} = e(\mathcal{P} \cap (\{a_i, \dots, a_{|\mathcal{A}|}\} \cup \{b_j, \dots, b_{|\mathcal{B}|}\}))\}$$

with $1 \leq i \leq |\mathcal{A}| + 1$ and $1 \leq j \leq |\mathcal{B}| + 1$. Since

$$K_{1,r+1} = e(\mathcal{P} \setminus \{b_1, \dots, b_r\})$$

and

$$e(\mathcal{P}(a_1 < b_r)) = e(\mathcal{P}) - e(\mathcal{P}(b_r < a_1)) = e(\mathcal{P}) - e(\mathcal{P} \setminus \{b_1, \dots, b_r\})$$

we have

$$e(\mathcal{P}(a_1 < b_r)) = K_{1,1} - K_{1,r+1}. \quad \square$$

Note that other values $e(\mathcal{P}(a_i < b_j))$ for any pair (a_i, b_j) would be trickier to compute. The value of $e(\mathcal{P})$ and the number of linear extensions of all poset extensions $\mathcal{P}(a_1 < b_r)$ and $\mathcal{P}(b_1 < a_r)$ can thus be computed in $O(|\mathcal{A}||\mathcal{B}| \log e(\mathcal{P}))$ time using dynamic programming with a $|\mathcal{A}| \times |\mathcal{B}|$ memory tableau containing numbers with values up to $\log e(\mathcal{P})$.

We can thus use the dynamic program to find a good query to perform instead of computing determinants. Moreover, we prove that we can update the information obtained via the dynamic program with the answer to a query in $O(n)$ time.

Proof. Suppose that we have computed $e(\mathcal{P})$ and all $e(\mathcal{P}(a_1 < b_r))$ and $e(\mathcal{P}(b_1 < a_r))$ using dynamic programming. According to these values and thanks to Linial's proof of the $1/3$ - $2/3$ conjecture for width-2 posets, we find a good query $a_1 <^? b_r$ or $b_1 <^? a_r$ to perform. Depending on the outcome of this query we update the tableau of the dynamic program. We may assume the good query was of the type $a_1 <^? b_r$ without loss of generality. If $b_r < a_1$ then we do not have to update anything, the chain $b_1 < \dots < b_r$ can simply be ignored. Otherwise, $a_1 < b_r$ and unless $r = 1$ we have to update at most $|\mathcal{A}|$ cells of the dynamic program tableau. The cells to update are $K_{1,1}$ through $K_{1,r}$. The new value to assign to $K_{1,r}$ is $K_{2,r}$ because if $a_1 < b_r$ then $e(\mathcal{P} \setminus \{b_1, \dots, b_{r-1}\}) = e(\mathcal{P} \setminus (\{b_1, \dots, b_{r-1}\} \cup \{a_1\}))$. Since we changed $K_{1,r}$ we also need to update all upstream values $K_{1,1}, \dots, K_{1,r-1}$. This is the only case where we have to update the tableau and the number of cells to update is $O(n)$. \square

We detail an implementation of the processing part of the algorithm. For the sake of simplicity, we consider a recursive implementation. This implementation outputs a linear extension compatible with the unknown total order $<$. The algorithm receives as input the poset \mathcal{P} covered by two chains $\mathcal{A} = (a_1 < \dots < a_{|\mathcal{A}|})$ and $\mathcal{B} = (b_1 < \dots < b_{|\mathcal{B}|})$, the oracle $<^?$ and a precomputed $|\mathcal{A}| \times |\mathcal{B}|$ tableau K .

Algorithm 4.1 (Efficient implementation of Linial's algorithm).

1. If $|\mathcal{A}| = 0$ and $|\mathcal{B}| = 0$ the algorithm stops.
- 2.1. If $|\mathcal{B}| = 0$ or $a_1 <_{\mathcal{P}} b_1$ swap² the roles of \mathcal{A} and \mathcal{B} .
- 2.2. If $|\mathcal{A}| = 0$ or $b_1 <_{\mathcal{P}} a_1$ output b_1 , run the algorithm with $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_1\}$, then stop.

²Several times during the algorithm we use an instruction that swaps the roles of \mathcal{A} and \mathcal{B} . The intent is to avoid duplication of instructions that would bloat the description. A side effect of this swap instruction is that it exchanges the roles of rows and columns in the tableau K .

- 3.1. If $\frac{e(\mathcal{P}(a_1 < b_1))}{e(\mathcal{P})} > 2/3$ swap the roles of \mathcal{A} and \mathcal{B} .
- 3.2. If $\frac{e(\mathcal{P}(a_1 < b_1))}{e(\mathcal{P})} < 1/3$,
 - 3.2.1. Compute r such that $1/3 \leq \frac{e(\mathcal{P}(a_1 < b_r))}{e(\mathcal{P})} \leq 2/3$.
 - 3.2.2. Query the oracle³ with $a_1 <^? b_r$.
 - 3.2.3. If $b_r < a_1$ output b_1, \dots, b_r , run the algorithm with $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_1, \dots, b_r\}$, then stop.
 - 3.2.4. Otherwise,
 - 3.2.4.1. Update $\mathcal{P} \leftarrow \mathcal{P}(a_1 < b_r)$.
 - 3.2.4.2. Update $K_{1,j} \leftarrow K_{2,j}$ for $1 \leq j \leq r$.
 - 3.2.4.3. Run the algorithm using the updated \mathcal{P} and K , then stop.
- 3.3. Otherwise,
 - 3.3.1. Query the oracle⁴ with $a_1 <^? b_1$.
 - 3.3.2. If $a_1 < b_1$ swap the roles of \mathcal{A} and \mathcal{B} .
 - 3.3.3. Output b_1 , run the algorithm with $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b_1\}$, then stop.

As a final remark, it is possible to reduce the computational complexity of both the construction and the update steps of the dynamic program. If we limit ourselves to storing *limited precision integers* and perform *limited precision arithmetic* when adding those integers, then we can shave off the $\log e(\mathcal{P})$ factor in both cases. Hence, if we divide the execution of the algorithm into a preprocessing phase and a processing phase, the complexity of the preprocessing phase is $O(|\mathcal{A}||\mathcal{B}|)$ while doing no comparisons, and the complexity of the processing phase is $O(N \log e(\mathcal{P}))$, where $N = \max\{|\mathcal{A}|, |\mathcal{B}|\}$. The algorithm uses at most $\log(\frac{1+\sqrt{5}}{2})^{-1} \log e(\mathcal{P}) \approx 1.44$ ITLB queries as shown by Linial [59].

³The only steps where we query the oracle are **3.2.2** and **3.3.1**.

⁴See footnote 3.

Chapter 5

Related Problems

We give more details about the problem of Sorting $X + Y$. Then we define a series of problems and establish connections between those problems, Sorting and Sorting $X + Y$.

5.1 Sorting $X + Y$

We are given the following problem

Problem 5.1 (Sorting $X + Y$). *Given two sets of numbers X and Y each of cardinality n , we want to sort the set*

$$X + Y = \{x + y : x \in X, y \in Y\}.$$

The set $X + Y$ is called the *sumset* (or *Minkowski sum*) of the two sets X and Y .

The earliest known reference is Fredman [33], who attributes the problem to Elwyn Berlekamp. As we can see in Figure 5.1, it looks tempting to solve

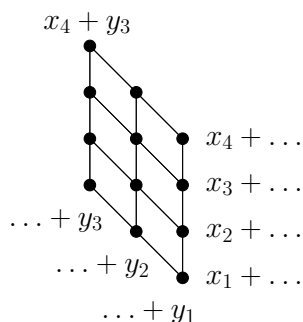


Figure 5.1: Typical Hasse diagram for the Sorting $X + Y$ problem.

Sorting $X + Y$ using the SUPI tools. However, it is not a special case of SUPI. We show later that there are better ways to look at the structure of $X + Y$.

Several geometric problems are “Sorting- $(X + Y)$ -hard” (see Barrera [4], Barequet and Har-Peled [3]). Specifically, there is a subquadratic-time transformation from sorting $X + Y$ to each of the following problems: computing the Minkowski sum of two orthogonally-convex polygons, determining whether one monotone polygon can be translated to fit inside another, determining whether one convex polygon can be rotated to fit inside another, sorting the vertices of a line arrangement, or sorting the interpoint distances between n points in \mathbb{R}^d .

Fredman [33] also mentions an immediate application to multiplying sparse polynomials. When we multiply polynomials, each of the operands can be decomposed in terms. If polynomials p and q are of size n and m , then the resulting polynomial contains nm terms and the degree of each term in the resulting polynomial is the sum of the degree of the operands terms. Since sparse polynomials are usually stored in sorted order with respect to the degree of their terms, we can easily see how the two problems are related.

Concerning sorting $X + Y$, the question of an $O(n^2)$ algorithm, optimal in a RAM model, still remains. Some additional references are Harper et al. [40], Kahn and Kim [48], Dietzfelbinger [25], Steiger and Streinu [82], Lambert [57], Erickson [28], Bremner et al. [7], O’Rourke [70].

5.2 Extending the Model

In the previous chapters, we considered that, in the decision tree model we use, the only kind of query that we were allowed to ask was of the form $a \leq^? b$, where a and b are elements of some input set.

It is time to upgrade our model by widening the concept of element comparison. In the following sections we consider that a comparison can be made between more than two elements. A comparison between three elements could be for example $2a - 3b + c \leq^? 0$. Decisions based on the result of such a comparison depend on the sign of the expression $2a - 3b + c$ that is, whether this expression is less than, equal to or greater than 0.

In this model, we are only allowed to query for linear combinations of input elements and those linear combinations can involve at most k of those elements. This generic model is the *k-linear decision tree model*.

As a concrete example, let us take the problem of sorting $X + Y$ we just introduced. The set $X + Y$ contains all pairwise sums of elements of X with elements of Y . Thus, if we want to sort $X + Y$ we could make queries of the

form $(x_i + y_j) \leq^? (x_{i'} + y_{j'})$ and this can be rewritten as $x_i + y_j - x_{i'} - y_{j'} \leq^? 0$ to fit our new model. Solving the problem this way would involve 4-linear queries. Therefore, we would be working in the 4-linear decision tree model. Of course, we take the conditional here because the definition of the problem is in no way forcing us to use $k = 4$.

5.3 Three Set Sum Problems

The three problems given in this section are about finding subsets whose elements sum up to a certain constant. The first one is the subset sum problem which is NP-complete [52]. The two others, k -SUM and k -variate linear degeneracy testing (abbreviated k -LDT), are variants of the first one where we fix the size k of the subsets to search for. k -LDT is different from k -SUM as it does not compute a sum in the strict sense. It computes the image of a k -variate linear function $\phi(s_1, \dots, s_k) = \alpha_0 + \sum_{i=1}^k \alpha_i s_i$, allowing other values than 0 or 1 for the coefficients $\alpha_0, \alpha_1, \dots, \alpha_k$. It is trivial to see that k -SUM is contained in k -LDT. We explain in the next sections how these set sum problems relate to sorting problems. Below are the formal definitions of the problems.

Problem 5.2 (Subset sum). *Given a set $\mathcal{U} \subset \mathbb{R}$, $|\mathcal{U}| = n$, decide whether there exists $\mathcal{S} \subseteq \mathcal{U}$, such that $\sum_{s \in \mathcal{S}} s = 0$.*

Problem 5.3 (k -SUM). *Given a set $\mathcal{U} \subset \mathbb{R}$, $|\mathcal{U}| = n$, decide whether there exists $\mathcal{S} \subseteq \mathcal{U}$, $|\mathcal{S}| = k$, such that $\sum_{s \in \mathcal{S}} s = 0$.*

Problem 5.4 (k -LDT). *Given a set $\mathcal{U} \subset \mathbb{R}$, $|\mathcal{U}| = n$, and $k+1$ real coefficients $\alpha_0, \alpha_1, \dots, \alpha_k \in \mathbb{R}$, decide whether there exists a k -tuple $\mathcal{S} = (s_1, \dots, s_k)$, $s_i \in \mathcal{U}$, such that $\alpha_0 + \sum_{i=1}^k \alpha_i s_i = 0$.*

5.4 Sorting is a 2LDT problem

We show that Sorting is a 2LDT problem. Let us define 2LDT

Problem 5.5 (2LDT). *Given a set $\mathcal{S} \subset \mathbb{R}$, $|\mathcal{S}| = n$, and real coefficients $\alpha_0, \alpha_1, \alpha_2$, decide whether there exists a pair $(a, b) \in \mathcal{S}^2$ such that $\alpha_0 + \alpha_1 a + \alpha_2 b = 0$.*

In the sorting problem we need to determine whether $a \leq b$ for all (a, b) pairs, this amounts to determining the sign of $a - b$. If we would like to rewrite the sorting problem definition in the same fashion we would come up with the following definition

Problem 5.6 (Sorting as a 2LDT problem). *Given a set $\mathcal{S} \subset \mathbb{R}$, $|\mathcal{S}| = n$, decide whether there exists a pair $(a, b) \in \mathcal{S}^2$, $a \neq b$ such that $a - b = 0$.*

The intuition is correct even though $a \neq b$ implies $a - b \neq 0$ for all pairs (a, b) , and so there cannot be such pair (a, b) , but, the important bit is that computing $a - b$ reveals the answer to the question $a \leq^? b$.

Let us prove why this definition maps to the definition of the Sorting problem.

Theorem 5.1 (Sorting is a 2LDT problem). *Given a set $\mathcal{S} \subset \mathbb{R}$, deciding whether there exists a pair $(a, b) \in \mathcal{S}^2$, $a \neq b$ such that $a - b = 0$ amounts to finding the answers of all $a \leq^? b$ queries.*

Proof. We may assume there is no such pair without loss of generality. Suppose that we have a pair $(a, b) \in \mathcal{S}^2$, $a \neq b$ for which we know that $a - b \neq 0$ and assume $a \leq b$ without loss of generality, then at least one of the following propositions holds:

1. We directly made the queries $a - b \leq^? 0$ and $b - a \leq^? 0$, revealing the sign of the expression $a - b$.
2. There is a $x \in \mathcal{S}$ such that $a \leq x \leq b$ for which we know that $a - x \leq 0$, $x - b \leq 0$, and at least one of $a - x \neq 0$ or $x - b \neq 0$.

Since $a - x \leq 0 \wedge x - b \leq 0 \iff a - x + x - b \leq 0 \iff a - b \leq 0$, in all cases knowing $a - b \neq 0$ implies knowing the sign of $a - b$ and thus the answer to the query $a \leq^? b$. \square

All sorting problem instances are thus instances of 2LDT with $\alpha_0 = 0, \alpha_1 = 1, \alpha_2 = -1$.

5.5 Sorting $X + Y$ is a 4LDT problem

Similarly to how the sorting problem is a linear degeneracy testing problem, Sorting $X + Y$ is a 4LDT problem. Let us define 4LDT

Problem 5.7 (4LDT). *Given a set $\mathcal{S} \subset \mathbb{R}$, $|\mathcal{S}| = n$, and real coefficients $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4$, decide whether there exists a 4-tuple $(a, b, c, d) \in \mathcal{S}^4$ such that $\alpha_0 + \alpha_1 a + \alpha_2 b + \alpha_3 c + \alpha_4 d = 0$.*

When sorting $X + Y$ we need to determine whether $a + b \leq c + d$ for all $(a, b, c, d) \in (X \times Y)^2$, this sums up to determining the sign of $(a + b) - (c + d)$, that is, asking questions of the kind $a + b - c - d \leq^? 0$. We can thus reformulate sorting $X + Y$ the following way

Problem 5.8 (Sorting $X + Y$ as a 4LDT problem). *Given a set $\mathcal{S} \subset \mathbb{R}$, $|\mathcal{S}| = n$, decide whether there exists a 4-tuple $(a, b, c, d) \in \mathcal{S}^4$, $a \neq c \vee b \neq d$ such that $a + b - c - d = 0$.*

Again, we want to prove that this problem is equivalent to Sorting $X + Y$. We state and prove a more general theorem than [Theorem 5.1](#).

Theorem 5.2 (The search and decision versions of the point location problem are equivalent). *Given a point $p = (p_1, \dots, p_n)$ and an arrangement of hyperplanes \mathcal{H} in \mathbb{R}^n , deciding whether p lies on one of the hyperplanes of the arrangement amounts to finding the cell of the arrangement containing p .*

Proof. We may assume p does not lie on any of the hyperplanes without loss of generality. Suppose that we have a hyperplane $H \in \mathcal{H}$, $H = a_1x_1 + \dots + a_nx_n = 0$ for which we know that $p \notin H$ and assume $a_1p_1 + \dots + a_np_n \leq 0$ without loss of generality, then at least one of the following propositions holds:

1. We directly made the queries $a_1p_1 + \dots + a_np_n \leq 0$ and $-a_1p_1 + \dots + -a_np_n \leq 0$, revealing the sign of the expression $a_1p_1 + \dots + a_np_n$.
2. There is a subset of the arrangement $\{H_1, \dots, H_k\}$ with $H_i = a_{i_1}x_1 + \dots + a_{i_n}x_n = 0$ such that H is a linear combination of these hyperplanes with coefficients $\beta_i \neq 0$ for which we know that $a_{i_1}p_1 + \dots + a_{i_n}p_n \leq 0$ if $\beta_i > 0$ or $-a_{i_1}p_1 + \dots + -a_{i_n}p_n \leq 0$ if $\beta_i < 0$, and at least one of $a_{i_1}p_1 + \dots + a_{i_n}p_n \neq 0$.

In all cases knowing $a_1p_1 + \dots + a_np_n \neq 0$ implies knowing the sign of $a_1p_1 + \dots + a_np_n$ and thus the answer to the query $a_1p_1 + \dots + a_np_n \leq 0$. Knowing all these answers one finds the cell of the arrangement containing p . \square

We can see that Sorting $X + Y$ is a special case of the point location problem in an arrangement of hyperplanes by mapping each possible query $a + b \leq c + d$ to a hyperplane $a + b - c - d = 0$.

Sorting $X + Y$ instances are thus instances of 4LDT with $\alpha_0 = 0, \alpha_1 = \alpha_2 = 1, \alpha_3 = \alpha_4 = -1$ and $\mathcal{S} = X \cup Y$.

5.6 Similar Complexity for k -SUM and k -LDT

The only important consequence of the difference between the definitions of k -SUM and k -LDT is that all the permutations of the k -tuples need to be

taken into account in k -LDT. In k -SUM we have $\binom{n}{k}$ k -tuples to consider, while in k -LDT we have $\binom{n}{k}k! = \frac{n!}{(n-k)!}$ k -tuples.

However, in [Section 7.6](#) we show that it is possible to translate k -LDT instances to k -SUM instances and obtain the same upper bounds as k -SUM for k -linear decision trees. If $k \geq 4$ is even, we can translate a k -LDT instance of size n to a balanced 2SUM instance of size $2n^{k/2}$ and obtain a $O(n^{k/2} \log n)$ algorithm. In the other case, when $k \geq 3$ is odd, a k -LDT instance of size n can be translated to an unbalanced 3SUM instance, where $|\mathcal{A}| = |\mathcal{B}| = n^{\frac{k-1}{2}}$ and $|\mathcal{C}| = n$, that can be solved in $O(n^{\frac{(k+1)}{2}})$ time.

We show later that Grønlund and Pettie [\[38\]](#) improve the upper bound for the 3SUM problem. The translation of k -LDT to 3SUM when k is odd still applies and allows to match the same $O(n^{k/2} \sqrt{\log n})$ bound as the one established for 3SUM in the same paper.

Chapter 6

Sorting $X + Y$

We describe three approaches to sort $X + Y$. For the sake of simplicity, we restrict ourselves to the case $|X| = |Y| = n$.

6.1 A First Approach by Merging

In our first approach¹, we sort $X + Y$ using half the comparisons required by the ITLB for Sorting. We do so by exploiting the poset structure of $X + Y$. We thus solve this problem in a strictly faster way than sorting a set of size n^2 with a classical sorting algorithm, by taking advantage of some of the information we already have.

The ITLB for sorting a set of size N is

$$\log N! \simeq N \log N - 1.44N + O(\log N).$$

Sorting $X + Y$ without information requires thus at least

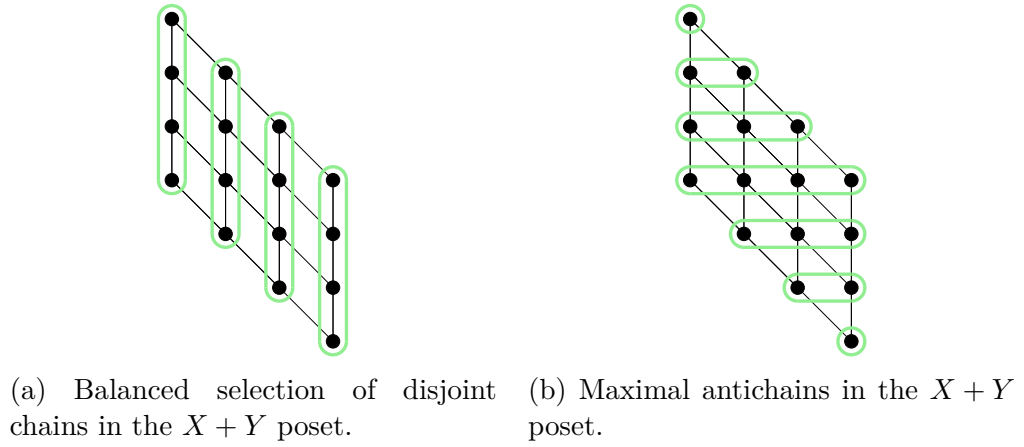
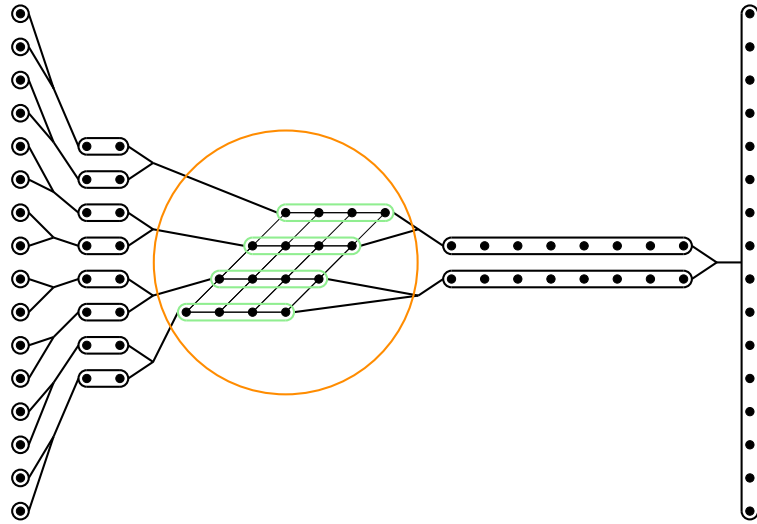
$$\log n^2! \simeq 2n^2 \log n - 1.44n^2 + O(\log n)$$

queries. In the next paragraphs we show that it is possible to sort $X + Y$ with at most $n^2 \log n + O(n^2)$.

We define *the poset structure of $X + Y$* (or simply *the $X + Y$ poset*) to be the poset containing the information that we can extract by applying the transitivity axiom on $X + Y$, using the total order on $X \cup Y$. If we exploit the fact that

$$X_i \leq X_{i'} \wedge Y_j \leq Y_{j'} \implies X_i + Y_j \leq X_{i'} + Y_{j'}$$

¹The ideas behind the first two sections can be found in Harper et al. [40].

Figure 6.1: Structure of the $X + Y$ poset.Figure 6.2: Resolution of the MERGE SORT algorithm. The circled part corresponds to the starting point of our first Sorting $X + Y$ algorithm on an input of size 4×4 .

we obtain the poset represented by the Hasse diagram in Figure 5.1 (Section 5.1). Chains and antichains in the $X + Y$ poset obey a regular pattern. To illustrate this, we take the 4×4 $X + Y$ poset. We highlight a possible selection of disjoint chains in Figure 6.1a and the set of maximal antichains found by a greedy algorithm (by iteratively removing a maximum chain) in Figure 6.1b.

To illustrate our approach, we show in Figure 6.2 the steps of the MERGE SORT algorithm on an input of size 16. What is highlighted in Figure 6.2 is a subset of the information we have when knowing the structure of $X + Y$, that is, the chains of Figure 6.1a. It means that the step we highlighted can be attained without any comparison if the set to sort is $X + Y$ and if we know its structure. Note that if not known a priori, this structure can be revealed for the cost of $2n \log n + O(n)$ comparisons, using a standard $n \log n + O(n)$ comparison sorting algorithm to sort sets X and Y .

In MERGE SORT there are $\log N$ steps. At step i we have built 2^i total orders on disjoint subsets of size $2^{\log N - i}$. Since the step represented in Figure 6.2 shows n total orders on disjoint subsets of size n , and that the total number of elements in $X + Y$ is $N = n^2$ we can conclude that Figure 6.2 shows us step $1/2 \log N$. Indeed, we give here an example where n is a power of 2 but it is not difficult to convince oneself that starting the algorithm at the median step saves us half the comparisons, for n large enough. However we rely on the results we gave in Section 2.3 to prove that it is possible to reuse this subset of the partial information optimally.

Theorem 6.1 (A first upper bound for Sorting $X + Y$). *Sorting $X + Y$ requires at most $n^2 \log n + O(n^2)$ queries to be made.*

Proof. Note that if $\log n^2!$ is the total amount of information to retrieve, then recycling the information contained in the n chains of length n gives us $n \cdot \log n! \simeq n^2 \log n - O(n^2)$ bits of information already in our possession. We saw in Section 2.3 that we can reuse this information optimally (up to a linear term in the input size) using the Huffman code based merging algorithm. \square

Since this algorithm reuses only a fraction of the available information, the question of the optimality of this approach remains. We give an answer to this question in the next sections.

6.2 Naively Looking at the Poset Structure

In the previous section, we have shown how one could derive a poset structure from the $X + Y$ set. We also saw that one can sort this $X + Y$ poset with merging algorithm of Section 2.3 using at most $n^2 \log n + O(n)$ comparisons.

Indeed, what we did in this section was modeling the Sorting $X + Y$ problem as the problem of Sorting under Partial Information. We then solved the problem with a straightforward algorithm.

In [Chapter 4](#), we have seen that there are algorithms solving the SUPI problem in $O(\log e(\mathcal{P}))$ comparisons. We wonder if it is possible to prove that, if we are only allowed to exploit the grid structure of the $X + Y$ poset, then Sorting $X + Y$ cannot be solved in less than $n^2 \log n + O(n)$ comparisons. In other words, we want to prove that the approach taken in the previous section is optimal up to a linear term.

We already saw that the Hasse diagram of the poset $X + Y$ can be drawn as a grid. This grid is finite and has a $\hat{0}$ and a $\hat{1}$. Moreover, any two elements of this grid have a unique infimum and supremum. When this three conditions are met for a poset, we call this poset a bounded lattice. We say that the poset $X + Y$ is a $n \times n$ bounded lattice.

We show that

Theorem 6.2 (Number of linear extensions of the $n \times n$ lattice). *Given a $n \times n$ lattice \mathcal{P} ,*

$$e(\mathcal{P}) = \frac{n^2!}{\prod_{k=1}^{2n-1} k^{n-|n-k|}}.$$

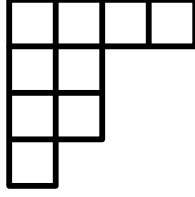
Note that an alternative way of writing $e(\mathcal{P})$ is

$$e(\mathcal{P}) = n^2! \cdot \left(\prod_{k=1}^{n-1} k! \right)^2 \cdot \left(\prod_{k=1}^{2n-1} k! \right)^{-1}.$$

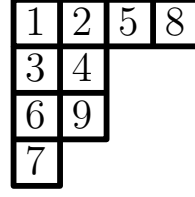
To prove this theorem, we introduce the notion of Ferrers diagrams and standard Young tableaux.

A *Ferrers diagram* can be seen as a set of square tiles arranged in a rectangular box divided in rows and columns. They are arranged so that there cannot be any space between two tiles lying on the same row or column and there can never be a row that contains more tiles than another row above it. Moreover, there cannot be space left between the left side of the box and the first tile of each row. An example of a Ferrers diagram is given in [Figure 6.3a](#).

A *Young tableau* is a special way of labelling the n tiles of a Ferrers diagram. If a Ferrers diagram is made of n tiles, then a Young tableau drawn on this diagram labels its tiles with the numbers from 1 to n . We call *standard* a Young tableau having the following property: if one reads tile labels of rows (from left to right) or columns (from top to bottom) of this tableau, one should only encounter increasing sequences of numbers. An example of a standard Young tableau is given in [Figure 6.3b](#).

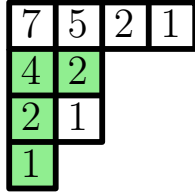


(a) Example of a Ferrers diagram with 9 tiles. The shape of this diagram is $\lambda = (4, 2, 2, 1)$.

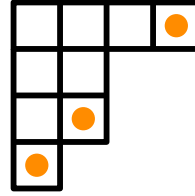


(b) Example of a standard Young tableau in the Ferrers diagram of Figure 6.3a.

Figure 6.3: Ferrers diagrams and Young tableaux.



(a) Ferrers diagram with hook length the hook bending at $(2, 1)$.



(b) Ferrers diagram with corners highlighted.

Figure 6.4: Hooks and corners in Ferrers diagrams.

The number of standard Young tableaux for a given Ferrers diagram shape λ can be computed using the hook-length formula d_λ .

Theorem 6.3 (Frame et al. [31]). *Let $\lambda = (\lambda_1, \dots, \lambda_m)$ be a partition of the natural number n , that is, such that $\sum_i \lambda_i = n$. The number of standard Young tableaux of shape λ is*

$$d_\lambda = \frac{n!}{\prod_{(i,j) \in \lambda} h_\lambda(i,j)}$$

$h_\lambda(i, j)$ denotes the length of an imaginary hook bending at (i, j) . The *hook* $H_\lambda(i, j)$ is a subset of the tiles of λ containing all tiles (a, b) such that $a = i$ and $b \geq j$ or $b = j$ and $a \geq i$. Figure 6.4a shows the hook length at each tile and highlights the hook bending at $(2, 1)$.

The hook-length formula² is attributed to Frame et al. [31]. Multiple ways to prove Theorem 6.3 are available in the literature. For more information,

²As a side note, the hook-length formula is closely related to multidimensional Catalan numbers. The hook-length formula generates the numbers on the diagonal of the matrix containing the n^{th} m -dimensional Catalan number at position (m, n) [39]. The n^{th} m -dimensional Catalan numbers gives the number of shortest paths in the m -dimensional unit grid from $(0, \dots, 0)$ to (n, \dots, n) such that if (x_1, \dots, x_m) is on a path then $x_1 \leq \dots \leq x_m$.

9	8	7	6	5
8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4	3	2	1

Figure 6.5: Hook lengths for the $n \times n$ lattice.

one could start with Greene et al. [37] which gives a nice probabilistic proof and pointers for further reading.

We explain the idea behind the proof of Greene et al. [37]. Figure 6.4b highlights *corners* of a Ferrers diagram. A standard Young tableau drawn in this diagram can only have its largest label given to one of these corner tiles, otherwise we would obtain a row or a column that is not increasing. The number of ways of arranging the standard Young tableau in the Ferrers diagram of shape λ is thus the sum of possible tableaux of size $n - 1$ over all shapes $\lambda \setminus \gamma$ where the largest label of our tableau of size n was given to corner γ of λ . Hence, the recurrence

$$d_\lambda = \sum_{\gamma \text{ is a corner of } \lambda} d_{\lambda \setminus \gamma} \quad (6.1)$$

holds. We can rewrite Equation 6.1 as

$$\sum_{\gamma \text{ is a corner of } \lambda} \frac{d_{\lambda \setminus \gamma}}{d_\lambda} = 1 \quad (6.2)$$

Greene et al. [37] define a hook walk to be a random walk starting at any tile in the Ferrers diagram such that at each step, one moves uniformly at random to another tile of the hook bending at the current tile. They show that each term of the sum in Equation 6.2 is the probability for a single hook walk to end in each of the corners, proving the theorem.

We return to our original problem. A $n \times n$ bounded lattice can be regarded as a $n \times n$ standard Young tableau, that is, a standard Young tableau of shape $\lambda = \{\lambda_1, \dots, \lambda_n\}$ with $\lambda_i = n, \forall i \in \{1, \dots, n\}$. Figure 6.5 shows the hook lengths for the $n \times n$ lattice, and thus for the $X + Y$ poset. We need to show that counting tableaux is the same as counting linear extensions. We label all n^2 nodes of \mathcal{P} , the Hasse diagram of $X + Y$, with distinct etiquettes $1, \dots, n^2$. Suppose that we are given a linear extension of \mathcal{P} that can be represented as a tuple $\omega = (j_1, \dots, j_{n^2})$ with all j_i distinct and between 1

and n^2 using our etiquettes. If we are given n^2 distinct numbers x_1, \dots, x_{n^2} then, according to our proof, there are d_λ ways to fill our tableau with those numbers and to each filled tableau corresponds a unique linear extension designated by ω . Hence,

$$e(\mathcal{P}) = d_\lambda.$$

We prove that

Theorem 6.4 (ITLB for the problem of sorting a $n \times n$ lattice). *Given a $n \times n$ lattice \mathcal{P} ,*

$$\log e(\mathcal{P}) = \Omega(n^2 \log n)$$

Proof. Using Stirling's approximation,

$$\begin{aligned} e(\mathcal{P}) &= \frac{(n^2)!}{1 \cdot 2 \cdot 2 \cdot \dots \cdot (2n-2) \cdot (2n-2) \cdot (2n-1)} \\ e(\mathcal{P}) &\geq \frac{(n^2)!}{(2n)^{n^2}} \\ \ln e(\mathcal{P}) &\geq \ln \left(\frac{(n^2)!}{(2n)^{n^2}} \right) \\ \lim_{n \rightarrow \infty} \ln e(\mathcal{P}) &\geq \ln \left(\sqrt{2\pi n^2} \left(\frac{n^2}{e} \right)^{n^2} \right) - n^2 \ln n - n^2 \ln 2 \\ \lim_{n \rightarrow \infty} \ln e(\mathcal{P}) &\geq \ln \sqrt{2\pi} + \ln n - n^2 \ln e + 2n^2 \ln n - n^2 \ln n - n^2 \ln 2 \\ \lim_{n \rightarrow \infty} \ln e(\mathcal{P}) &\geq n^2 \ln n - n^2 \ln(2e) + \ln n + \ln \sqrt{2\pi} \quad \square \end{aligned}$$

Thus, solving $X + Y$ in $\mathcal{O}(n^2 \log n)$ comparisons is not possible if we only consider the information contained in the poset structure. Moreover, to sort a poset having the same structure, the merging technique explained earlier is optimal. In [Section 6.4](#) we show that there is indeed more information available.

6.3 A Decision Tree for a Generic Sorting Problem

We expose an algorithm due to Fredman [33] that solves sorting problems making at most $\log |\Gamma| + 2n$ queries, where Γ is the set of possible linear extensions of the input. In the next section we show how this algorithm can be used in the context of Sorting $X + Y$.

Fredman [33] defines a generic sorting problem as follows

Problem 6.1 (Generic sorting problem). Let $\mathcal{S} = \{x_1, \dots, x_n\}$ be an n -element set. We define a sorting problem on these element to be a pair (Γ, P) , where Γ is a subset of the $n!$ possible linear orderings on \mathcal{S} , and P is a family $\langle \mathcal{A}_1, \dots, \mathcal{A}_r \rangle$ of disjoint nonempty sets that partition Γ so that $\Gamma = \mathcal{A}_1 \cup \dots \cup \mathcal{A}_r$. Given an (unknown) ordering $\omega \in \Gamma$, we want to determine to which set \mathcal{A}_j the ordering ω belongs by performing a sequence of comparisons between pairs of elements, $x_i \leq? x_j$. A comparison algorithm is said to solve (Γ, P) if the following condition is satisfied. Upon representing the algorithm as a comparison tree T , we associate with each leaf L of T the subset Γ_L consisting of the orderings in Γ that are consistent with the path through T ending at L . For each L we must have $\Gamma_L \subseteq \mathcal{A}_j$ for some $\mathcal{A}_j \in P$.

As usual, we are interested in the decision tree complexity of this problem, that is, for fixed Γ and P the minimum height of a decision tree allowing to find \mathcal{A}_j for any of the possible inputs ω compatible with Γ . We denote this complexity by $N(\Gamma, P)$.

Since the final goal is to sort $X + Y$, we are only interested in the case where P is a partition of Γ into singleton sets, as we desire to compute the correct linear extension of $X + Y$ out of all its feasible linear extensions. Since P is the same for every Γ in this case, we always write $N(\Gamma)$ to mean $N(\Gamma, P)$.

Fredman [33] proves the following theorem

Theorem 6.5 (Fredman [33]). A poset of cardinality n , whose linear extension is known to be contained in Γ can be sorted with $N(\Gamma) = \log |\Gamma| + 2n$ pairwise comparisons.

For our use case, n is the cardinality of the set $X + Y$. In the previous section we already found a lower bound for the problem of finding a linear extension of a $n \times n$ lattice. We also showed that if we only use the information contained in the poset structure of $X + Y$, then sorting $X + Y$ is the same as finding a linear extension of a $n \times n$ lattice. In the next section we compute the actual information-theoretic lower bound for Sorting $X + Y$. This new lower bound is our $|\Gamma|$ in Theorem 6.5.

To prove the correctness of the theorem, Fredman [33] builds a biased insertion sort algorithm that we explain in the next paragraphs.

The algorithm we consider is an iterative algorithm running in n iterations. Suppose that at iteration k of the algorithm we have already sorted a subset of \mathcal{S} of size $k - 1$.

$$\underbrace{x_1, \dots, x_{k-1}}_{\text{subset}} : \underbrace{x_{i_1} < \dots < x_{i_{k-1}}}_{\text{relative ordering}}$$

What we do at iteration k is insert x_k at the appropriate location in the already existing relative ordering to extend it to $x_{i_1} < \dots < x_{i_k}$.

We want to reformulate this insertion process using an equivalent formulation that involves location numbers b_1, \dots, b_k . The location number b_k means that at iteration k , x_k is inserted at position b_k implying that $i_{b_k} = k$.

The location number b_k is defined as $b_k = |\{x_j : j \leq k, x_j \leq x_k\}|$, that is, the number of elements from the relative ordering of iteration k that are smaller or equal to x_k . Since $x_k \leq x_k$ is always true, b_k is at least 1 and since $|\{x_j : j \leq k\}| = k$, b_k is at most k . We thus have,

$$1 \leq b_k \leq k.$$

We say b_k is the location number of x_k after its insertion into $x_{i_1} < \dots < x_{i_{k-1}}$.

We have established a correspondence between orderings on sets of cardinality n and n -tuples (b_1, \dots, b_n) with $1 \leq b_i \leq i$. Hence, Γ can be represented as a set of n -tuples of b_k and inserting x_k is tantamount to the determination of b_k . In other words, asking whether $x_k \leq^? x_{i_j}$ is equivalent to asking whether $b_k \leq^? j$.

To prove the theorem, Fredman [33] gives an algorithm whose existence implies an even more general result.

Lemma 6.6 (Fredman [33]). *Given $n \geq 1$, let \mathcal{S} be a finite set of n -tuples with unrestricted positive integer coordinates. Given an unknown n -tuple b in \mathcal{S} , we can determine its components, b_1, b_2, \dots , in that order, by making queries of the form $b_i \leq^? j, j \in \mathbb{N}$, and with a total of no more than $\log |\mathcal{S}| + 2n$ such queries.*

We show how to implement an algorithm matching this description. First, in order to simplify the notation, we describe a recursive version of our algorithm. The k^{th} recursion step of the recursive algorithm processes the k^{th} iteration. A complete execution of the algorithm would determine the n -tuple (b_1, \dots, b_n) . At the beginning of iteration k , we already have determined a prefix (b_1, \dots, b_{k-1}) of this n -tuple. What remains to be computed is the suffix (b_k, \dots, b_n) of the n -tuple. In the recursive version of the algorithm we forget about the already computed prefix since it cannot change and instead of writing (b_k, \dots, b_n) we write (b'_1, b'_2, \dots) , where $b'_1 = b_k, b'_2 = b_{k+1}$, etc. From here on till the end of this section we write b_i to mean b'_i for the sake of simplicity.

We list the steps of the recursive algorithm

Algorithm 6.1 (Fredman [33]).

input \mathcal{S} , a set of n -tuples with unrestricted positive integer coordinates.

1. Let $\mathcal{S}(k) \stackrel{\text{def}}{=} |\{(b_1, \dots) \in \mathcal{S} : b_1 = k\}|$, since \mathcal{S} is a finite set, there are finitely many values $i_1 < \dots < i_l$ such that $\mathcal{S}(i_j) > 0$. We map each i_j to j and we write $N_j = \mathcal{S}(i_j)$. Using this mapping we can, without loss of generality, assume $i_j = j$.
2. Partition the interval $(0, 1]$ into l adjacent intervals each of size $\frac{N_j}{|\mathcal{S}|}$. For each $r \in (0, 1]$, we define $0 \leq J(q) \leq l$ to be the number of interval midpoints lying to the left of r .
3. Search and find $b_1 = i$ by applying binary search to the partitioned interval using queries of the form $b_1 \leq^? J(q)$.

induction If $n > 1$, we call the procedure again with

$\{(b_2, \dots, b_n) : (b_1, b_2, \dots, b_n) \in \mathcal{S} \text{ and } b_1 = i\}$ as the input set.

The goal of step 1 is to offer a mapping between outputs of $J(q)$ and the possible original values for b_1 . After the substitution of i_j with j , asking $b_1 \leq^? J(q)$ means asking $b_1 \leq^? i_{J(q)}$ before the substitution takes place. Hence, without loss of generality we can assume $i_j = j$. The value N_j expresses the frequency or *number of occurrences* of $b_1 = j$ in the set \mathcal{S} , therefore

$$\sum_{j=1}^l N_j = |\mathcal{S}|.$$

In step 2 we partition $(0, 1]$ in a way that allows binary search to be efficient with respect to the complexity of the remaining induction steps. The partitioning procedure assigns interval sizes to possible values for b_1 proportional to their frequency in \mathcal{S} . Hence, if the interval associated with $b_1 = i$ is large, we have to make a small number of queries to find it and the cardinality of \mathcal{S} at the next induction step remains large. Otherwise, if the same interval is small, the number of queries required to find it is large, but, in this case, the induction step handles a smaller set \mathcal{S} .

The only step where we make queries is 3. We use binary search as follows. First we ask if b_1 lies in the first or the second half of $(0, 1]$, by asking $b_1 \leq^? J(\frac{1}{2})$, and then we continue this procedure with the interval designated by the answer to our question as our query interval. If the designated interval is the first half of $(0, 1]$, that is, $(0, \frac{1}{2}]$, because $b_1 \leq J(\frac{1}{2})$ the next question we ask is $b_1 \leq^? J(\frac{1}{4})$. Otherwise the designated interval is $(\frac{1}{2}, 1]$ and the next question $b_1 \leq^? J(\frac{3}{4})$, etc. We stop when we have ascertained that the query interval we are left with contains only one of the interval midpoints of our partitioned $(0, 1]$ interval.

We prove that if we have to perform r queries before finding such an interval, $b_1 = J(q)$ for all q in this interval, then $N_{b_1} \leq \frac{4|\mathcal{S}|}{2^r}$. Suppose that

$b_1 = i$ and that the length of the i^{th} interval, containing the i^{th} midpoint, is $\frac{N_i}{|\mathcal{S}|} > \frac{4}{2^r}$, thus has radius greater than $\frac{2}{2^r} = \frac{1}{2^{r-1}}$.

After having made $r - 1$ queries, our current query interval has length $1/2^{r-1}$. This query interval must be one of all the possible 2^{r-1} intervals of the partition of $(0, 1]$ in adjacent intervals of length $1/2^{r-1}$ starting at 0. We prove a rather simple lemma

Lemma 6.7. *All query intervals we consider can be written $(\frac{c}{2^{r-1}}, \frac{c+1}{2^{r-1}}]$ for some integer c with $0 \leq c \leq 2^{r-1}$. Moreover, for all such intervals containing the i^{th} midpoint*

$$J\left(\frac{c}{2^{r-1}}\right) < i \leq J\left(\frac{c+1}{2^{r-1}}\right).$$

Proof. At the beginning of the binary search algorithm, the search interval is $(0, 1]$. It excludes 0 because $J(0) = 0$ while $b_1 \geq 1$. Suppose that the current query interval is $(L, R]$. If during the search a question $b \leq J(q)$ gets a “yes” answer then we update the upper bound and get a new query interval $(L, q]$. Otherwise, $b > J(q)$ and we can update the lower bound and the query interval becomes $(q, R]$. In both cases the lower bound remains excluded. At the beginning of the algorithm, $r - 1 = 0$, $L = 0$ and $R = 1$, hence $c = 0$. For any query interval $(\frac{c}{2^{r-1}}, \frac{c+1}{2^{r-1}}]$ the query point is $q = \frac{(c + (c+1))/2}{2^{r-1}} = \frac{2c+1}{2^r}$. Depending on the outcome of the query $b \leq J(q)$ the new query interval is either $(\frac{2c}{2^r}, \frac{2c+1}{2^r}]$ or $(\frac{2c+1}{2^r}, \frac{2c+2}{2^r}]$. Lemma 6.7 follows by induction. \square

To be valid, the current query interval must contain the i^{th} midpoint. If this query interval contained more than the i^{th} midpoint, the algorithm would have to use at least one additional query. Since the radius of the i^{th} interval is greater than $1/2^{r-1}$ it is not possible for the query interval, which has radius $\frac{c+1-c}{2^{r-1}} = \frac{1}{2^{r-1}}$ to contain the i^{th} midpoint and at the same time escape the i^{th} interval. Hence, the i^{th} midpoint is the only midpoint inside the query interval. We conclude that,

$$i - 1 = J\left(\frac{c}{2^{r-1}}\right) < b_1 \leq J\left(\frac{c+1}{2^{r-1}}\right) = i,$$

thus $b_1 = i$ and the search ends before performing a r^{th} query.

We have proved that if the search for b_1 requires at least r queries then we can be sure that $N_{b_1} \leq \frac{4|\mathcal{S}|}{2^r}$. Letting \mathcal{S}_k denote the set of feasible solutions at the beginning of step k and r_k denote the number of queries made at step k , this means that at step $k + 1$, $|\mathcal{S}_{k+1}| \leq \frac{4|\mathcal{S}_k|}{2^{r_k}}$. After the last recursion step we have determined all components of the n -tuple to be found and hence the size of the feasible solutions set is 1. We have

$$1 \leq \frac{4|\mathcal{S}_n|}{2^{r_n}} \leq \frac{4^2|\mathcal{S}_{n-1}|}{2^{r_n+r_{n-1}}} \leq \dots \leq \frac{4^n|\mathcal{S}_1|}{2^{\sum_{i=1}^n r_i}},$$

and since $\mathcal{S}_1 = \mathcal{S}$ we get

$$\sum_{i=1}^n r_i \leq \log(4^n |\mathcal{S}|) = \log |\mathcal{S}| + 2n.$$

Moran and Yehudayoff [67] give an average-case analysis of this algorithm when the distribution over the inputs is not uniform.

6.4 Counting Linear Extensions

We state a theorem due to Buck [12] that allows us to conclude this chapter with the result that the decision tree complexity of sorting $X + Y$ is $O(n^2)$.

The theorem goes as follows

Theorem 6.8 (Buck [12]). *Consider the partition of space defined by an arrangement of m hyperplanes in \mathbb{R}^d . The number of regions of dimension $k \leq d$ is at most*

$$\binom{m}{d-k} \left(\binom{m-d+k}{0} + \binom{m-d+k}{1} + \cdots + \binom{m-d+k}{k} \right)$$

and the number of regions of all dimensions is $O(m^d)$.

To use this theorem, we model Sorting $X + Y$ as a point location problem in an arrangement of hyperplanes. Each instance of the problem is consisting of two sets of real numbers X and Y , both of cardinality n . An instance can thus be encoded as a vertex in \mathbb{R}^{2n} . Here is a possible encoding,

$$x = (X_1, \dots, X_n, Y_1, \dots, Y_n).$$

For every combination $(X_{i_1}, X_{j_1}, Y_{i_2}, Y_{j_2})$ of 4 components of this vertex we want to know whether $X_{i_1} + Y_{i_2} \leq X_{j_1} + Y_{j_2}$. This can be rewritten as,

$$X_{i_1} + Y_{i_2} - X_{j_1} - Y_{j_2} \stackrel{?}{\leq} 0$$

and it is equivalent to asking if x lies above, on or under an hyperplane of equation $X_{i_1} + Y_{i_2} - X_{j_1} - Y_{j_2} = 0$. There are $O(n^4)$ of those hyperplanes, and thanks to Theorem 6.8 we know that an arrangement built from them would contain $O((n^4)^{2n}) = O(n^{8n})$ regions. Hence, for $X + Y$, $|\Gamma| = O(n^{8n})$ and by using the algorithm of Fredman to find the correct n^2 -tuple, the decision tree complexity is $O(n \log n + n^2) = O(n^2)$.

In the next section we give a visual proof for a simplified version of Buck's theorem.

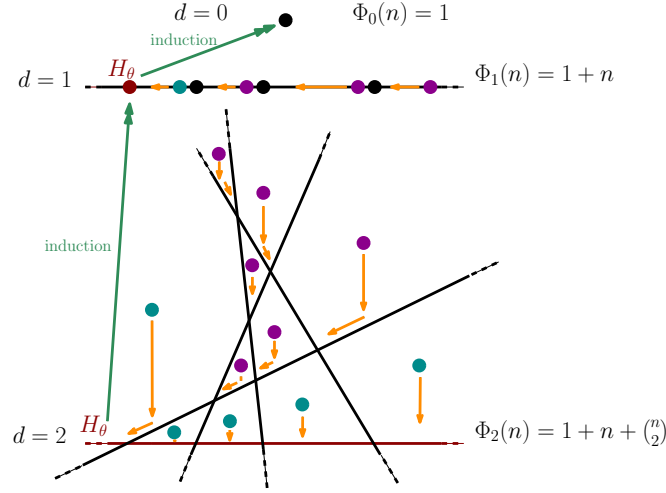


Figure 6.6: Induction steps in the second proof of Matoušek [62].

6.5 Insight into Buck's Theorem

Matoušek [62] gives two proofs for a simpler version of [Theorem 6.8](#). The second one is nice and visual, we sketch it to give an insight into why [Theorem 6.8](#) holds true.

The simpler theorem is the following

Theorem 6.9 (Number of d -cells in an arrangement of hyperplanes). *Consider the partition of space defined by an arrangement of n hyperplanes in \mathbb{R}^d . The number of cells (or d -cells, regions of dimension d) is at most*

$$\Phi_d(n) = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{d}$$

Proof. We assume a partition of space \mathbb{R}^d by a simple arrangement of n hyperplanes, that is, all d hyperplanes intersect in exactly one point and no more than d hyperplanes are incident to a point.

[Figure 6.6](#) shows that we can divide the set of cells in two disjoint subsets, \mathcal{L} and \mathcal{U} . The subset \mathcal{L} is constituted of cells that are intersected by an imaginary additional hyperplane H_θ having all $\binom{n}{d}$ intersections of the arrangement on one of its sides (either H_θ^+ or H_θ^-) and thus intersecting all the n hyperplanes of the arrangement on one side of the vertices. The subset \mathcal{U} contains the remaining cells.

$$|\mathcal{L}| = \Phi_{d-1}(n) \text{ since this is exactly the configuration for } \mathbb{R}^{d-1}$$

$$|\mathcal{U}| = \binom{n}{d} \text{ since ...}$$

Imagine we drew a ball in each cell of the arrangement. If we pretend H_θ is the ground and we activate gravity, the balls in the cells of \mathcal{L} would have fallen on H_θ . Each of them would be lying on a $(d-1)$ -cell of the partition of H_θ by the arrangement of the intersections of each hyperplane of the arrangement with H_θ . The balls in the cells of \mathcal{U} cannot fall on the ground since H_θ does not intersect the cells containing them. Balls in \mathcal{U} cells can thus only end up in one of the $\binom{n}{d}$ intersections of the n hyperplanes. \square

Matoušek [62] also shows that

$$\Phi_d(n) = \Phi_d(n-1) + \Phi_{d-1}(n-1)$$

in the first, more rigorous, proof. The term $\Phi_d(n-1)$ can be understood as the number of cells of an arrangement of $n-1$ hyperplanes that were existing before the addition of an n^{th} hyperplane. Then, the term $\Phi_{d-1}(n-1)$ counts the number of cells that are cut in two by the n^{th} hyperplane.

Chapter 7

Decision Trees for 3SUM

We discuss lower bounds for 3SUM implied by a reasoning on the decision tree model and also present more general results for k -LDT. The 3SUM problem is of great interest because a myriad of problems have been proved to be 3SUM-hard. Hence, a strong lower bound for the 3SUM problem would imply strong lower bounds for a multitude of other problems.

For a long time we thought that it was not possible to solve the 3SUM problem in subquadratic time. This took the form of a conjecture named *the 3SUM conjecture*. In 2014, Grønlund and Pettie [38] proved that this conjecture was false, pushing away the boundaries and leaving us with new paths to explore.

On this topic, we first illustrate an example of a 3SUM-hard problem and detail a linear-time mapping reduction from 3SUM for this problem. In the second section we formally state the 3SUM conjecture. The last four sections of this chapter are dedicated to results in the linear decision tree model. The third one explains results due to Erickson [28] commenting the lack of power of k -linear decision trees when it comes to solving k SUM. Then we cite results by Ailon and Chazelle [1] using s -linear decision trees, where $s > k$. In the fifth section we discuss recent results, due to Grønlund and Pettie [38], that refute the 3SUM conjecture using $(2k - 2)$ -linear decision trees. Lastly, we detail a method that allows to solve k -LDT using algorithms for 3SUM.

7.1 3SUM-hard problems

We already mentioned that many problems are reducible from 3SUM in such a way that an efficient algorithm for those problems would provide efficient algorithms for the 3SUM problem. Hence, a lower bound for 3SUM would imply lower bounds for those other problems.

One of these problems is the 3-points-on-line problem in the plane.

Problem 7.1 (3-points-on-line problem). *Given a finite set \mathcal{U} of points in \mathbb{R}^2 , decide if \mathcal{U} contains three collinear points.*

Theorem 7.1 (Hardness of the 3-points-on-line problem). *The 3-points-on-line problem in the plane is 3SUM-hard.*

Proof. A 3SUM instance \mathcal{S} is reducible to a 3-points-on-line instance $\mathcal{S}' = \{(x, x^3) : x \in \mathcal{S}\}$. This reduction takes linear time in the input size. Moreover, there is a direct mapping from a solution to the 3-points-on-line instance to a solution of the original 3SUM problem.

Indeed, if $a', b', c' \in \mathcal{S}'$ are collinear we have

$$\begin{aligned} a'_1(b'_2 - c'_2) + b'_1(c'_2 - a'_2) + c'_1(a'_2 - b'_2) &= 0 \\ a(b^3 - c^3) + b(c^3 - a^3) + c(a^3 - b^3) &= 0 \end{aligned}$$

which if treated as a polynomial in c has roots $\{-(a+b), a, b\}$. Roots a and b can be checked trivially. When $a + b + c = 0 \iff c = -(a+b)$ we have

$$\begin{aligned} &ab^3 + a(a^3 + 3a^2b + 3ab^2 + b^3) \\ &-b(a^3 + 3a^2b + 3ab^2 + b^3) - a^3b - a^4 - a^3b + b^4 + ab^3 \\ &= \underline{ab^3} + \underline{a^4} + \underline{3a^3b} + \underline{3a^2b^2} + \underline{ab^3} - \overline{a^3b} - \overline{3a^2b^2} \\ &\quad - \overline{3ab^3} - \overline{b^4} - \overline{a^3b} - \overline{a^4} - \overline{a^3b} + \underline{b^4} + \underline{ab^3} = 0 \end{aligned}$$

Since we will only check if distinct points are collinear we cannot have $c = a$ or $c = b$. Hence, the only realisable root of our polynomial is $c = -(a+b)$. \square

For a more exhaustive list, King [53] and Gajentaan and Overmars [34] propose a review of 3SUM-hard problems.

7.2 The Conjecture

With the lack of an algorithm that would perform better than the naive $O(n^2)$ one, a conjecture eventually emerged

Conjecture 7.1. *The complexity of the 3SUM problem in the decision tree model is $\Omega(n^2)$.*

It was indeed proved that this lower bound holds for a particular case of the decision tree model. By restricting himself to a 3-linear decision tree model, Erickson [28] proves that 3SUM is $\Omega(n^2)$ in this particular model. A second proof for this particular case is given by Ailon and Chazelle [1].

If this conjecture is true, it would mean that all the 3SUM-hard problems are doomed to have a $\Omega(n^2)$ complexity lower bound in the linear decision tree model.

7.3 k -linear Decision Trees

In Erickson [28], it is shown that we cannot solve 3SUM in subquadratic time in the 3-linear decision tree model. A general proof for the k -LDT problem in the k -linear decision tree model is featured. In the case of k even, a bound of $\Omega(n^{\frac{k}{2}})$ is given. For odd k , the bound is $\Omega(n^{\frac{k+1}{2}})$.

Their result is stated as follows

Theorem 7.2 (Erickson [28]). *The optimal depth of a k -linear decision tree that solves a k -LDT problem is $\Theta(n^{\lceil k/2 \rceil})$.*

The proof uses an adversary argument which can be explained geometrically. As we show later, we can solve k -LDT problems by modeling them as point location problems in an arrangement of hyperplanes. Solving a point location problem in an arrangement of hyperplanes amounts to determine which cell of the arrangement contains the input point. The adversary argument of Erickson [28] is that there exists a cell having $\Omega(n^{\lceil k/2 \rceil})$ boundary facets and thus if the input point is inside that cell, then an algorithm must check if the point is on any of the facets of that cell. If the algorithm stops without doing so, an adversary could freely move the point from the inside of the cell to its boundary, changing the output without the algorithm noticing it.

For information, we hereunder detail the classical $O(n^2)$ algorithm for 3SUM, successfully attaining the lower bound proved in [28] for 3-linear decision trees. In this description we handle the three-set version of 3SUM, that is, the case where a , b and c are taken from the three sets \mathcal{A} , \mathcal{B} and \mathcal{C} instead of a single set \mathcal{U} . The description comes from Grønlund and Pettie [38].

Algorithm 7.1 ($O(n^2)$ algorithm for 3SUM).

1. Sort \mathcal{A} and \mathcal{B} in increasing order as $\mathcal{A}_1 < \dots < \mathcal{A}_{|\mathcal{A}|}$ and $\mathcal{B}_1 < \dots < \mathcal{B}_{|\mathcal{B}|}$
2. For each $c \in \mathcal{C}$,
 - 2.1. Initialize $lo \leftarrow 1$ and $hi \leftarrow |\mathcal{B}|$
 - 2.2. Repeat until $lo > |\mathcal{A}|$ or $hi < 1$:
 - 2.2.1. If $\mathcal{A}_{lo} + \mathcal{B}_{hi} + c = 0$, report witness $(\mathcal{A}_{lo}, \mathcal{B}_{hi}, c)$
 - 2.2.2. If $\mathcal{A}_{lo} + \mathcal{B}_{hi} + c > 0$ then decrement hi , otherwise increment lo .

The running time complexity of this algorithm is $O(|\mathcal{C}|(|\mathcal{A}| + |\mathcal{B}|))$. Hence, for the case $\mathcal{A} = \mathcal{B} = \mathcal{C}$, that is, the one-set version of 3SUM, its complexity is $O(n^2)$.

At the end of [28], it is asked whether other kinds of decision trees would prove to be more powerful. The next section cites a progress in this direction made by Ailon and Chazelle [1].

7.4 s -linear Decision Trees

Ailon and Chazelle [1] study s -linear decision trees to solve the k -SUM problem when $s > k$. In particular, they give an additional proof for the $\Omega(n^{\lceil \frac{k}{2} \rceil})$ lower bound of Erickson [28] and generalize the lower bound for the s -linear decision tree model when $s > k$.

Note that the exact lower bound given by Erickson [28] for $s = k$ is $\Omega((nk^{-k})^{\lceil \frac{k}{2} \rceil})$ while the one given by Ailon and Chazelle [1] is $\Omega((nk^{-3})^{\lceil \frac{k}{2} \rceil})$. Their result improves therefore the lower bound for $s = k$ when k is large.

The lower bound they prove for $s > k$ is the following

Theorem 7.3 (Ailon and Chazelle [1]). *For any instance of k -LDT, the tree depth is at least*

$$\Omega\left((nk^{-3})^{\frac{2k-s}{2\lceil \frac{s-k+1}{2} \rceil}(1-\epsilon_k)}\right),$$

where $\epsilon_k > 0$ tends to 0 as $k \rightarrow \infty$.

We can interpret this result as a manifestation of the fact that allowing $s > k$ can lead to more powerful models of computation, as we will see in the next section and in the last chapter.

However, this new lower bound breaks down when $k = n^{1/3}$ or $s = \Omega(k)$ and the cases where $k < 6$ give trivial lower bounds. For example, in the case of 3SUM with $s = k + 1$ we get a lower bound that is $O(n)$. This is worse than the general lower bound of $\Omega(n \log n)$.

In the next section we discuss the first successful attempt at finding a subquadratic algorithm for the 3SUM problem.

7.5 4-linear Decision Trees

Grønlund and Pettie [38] build a decision tree solving the 3SUM problem using a subquadratic number of linear queries. Moreover, two algorithms solving the 3SUM problem in subquadratic time on a Random Access Machine are given. These results refute the long-lived conjecture stating that 3SUM cannot be solved in subquadratic time on a *Random Access Machine*.

In the classical quadratic algorithm for 3SUM, 3-linear queries of the form *do these 3 numbers sum up to 0* are used. In their first algorithm, breaking the $\Omega(n^2)$ bound on the number of queries, Grønlund and Pettie use 4-linear queries. We list hereunder the steps of this algorithm. The algorithm decides the one-set version of 3SUM with input set \mathcal{A} , $|\mathcal{A}| = n$. By looking at the number of queries this algorithm requires to finish, we prove the decision tree complexity of 3SUM is $O(n^{3/2}\sqrt{\log n})$.

Algorithm 7.2 (Grønlund and Pettie [38]).

1. Sort \mathcal{A} in increasing order as $\mathcal{A}(1) < \dots < \mathcal{A}(n)$.
2. Partition \mathcal{A} into $\lceil n/g \rceil$ groups $\mathcal{A}_1, \dots, \mathcal{A}_{\lceil n/g \rceil}$ of size at most g , where $\mathcal{A}_i \stackrel{\text{def}}{=} \{\mathcal{A}(1 + (i-1)g), \dots, \mathcal{A}(ig)\}$ and $\mathcal{A}_{\lceil n/g \rceil}$ may contain less than g elements. The first and last elements of \mathcal{A}_i are $\min(\mathcal{A}_i) = \mathcal{A}(1 + (i-1)g)$ and $\max(\mathcal{A}_i) = \mathcal{A}(ig)$.
3. Sort $\mathcal{D} \stackrel{\text{def}}{=} \bigcup_{i=1}^{\lceil n/g \rceil} (\mathcal{A}_i - \mathcal{A}_i) = \{a - a' : a, a' \in \mathcal{A}_i\}$.
4. Sort the sets $\mathcal{A}_{i,j} \stackrel{\text{def}}{=} \mathcal{A}_i + \mathcal{A}_j = \{a + b : a \in \mathcal{A}_i \text{ and } b \in \mathcal{A}_j\}$ for all $i, j \in \{1, \dots, \lceil n/g \rceil\}$.
5. For k from 1 to n ,
 - 5.1. Initialize $lo \leftarrow 1$ and $hi \leftarrow \lceil k/g \rceil$ to be the group index of $\mathcal{A}(k)$.
 - 5.2. Repeat until $hi < lo$:
 - 5.2.1. If $-\mathcal{A}(k) \in \mathcal{A}_{lo,hi}$, report “solution found” and halt.
 - 5.2.2. If $\max(\mathcal{A}_{lo}) + \min(\mathcal{A}_{hi}) > -\mathcal{A}(k)$ then decrement hi , otherwise increment lo .
6. Report “no solution” and halt.

The algorithm is arranged in two phases. In the first phase, steps **1** through **4**, we build a data structure for efficient lookups in preparation of the second phase. The second phase, steps **5** through **6**, consists of a loop where we search for all $a, b \leq \mathcal{A}(k)$ such that $a + b + \mathcal{A}(k) = 0$ using the efficient lookup data structure built during the first phase.

Step **1** requires $O(n \log n)$ queries using any optimal sorting algorithm. Step **2** explains the indexing convention we use throughout the algorithm and requires no query. We skip the explanation of step **3** for the moment. If done as is, step **4** would require to sort $(\frac{n}{g})^2$ $\mathcal{A}_i + \mathcal{A}_j$ sets, each of which, by [Theorem 6.5](#), would require $O(g^2)$ queries to sort. In total, step **4** would need $O(n^2)$ queries to be executed. Clearly this way of doing would not qualify as a subquadratic algorithm. We will come back at it later after analyzing the complexity of the second phase.

The rest of the algorithm is straightforward. The loop of step **5** is $O(n)$. The loop of step **5.2** is $O(\frac{n}{g})$ since we always either increment lo or decrement hi . The lookup at step **5.2.1** can be achieved with $\log g^2 = O(\log g)$ queries using a standard binary search procedure. This gives us a total complexity of $O(\frac{n^2}{g} \log g)$ for the second phase.

The catch in Grønlund and Pettie [38] is that they show it is possible to achieve step 4 in a subquadratic number of queries. The idea is to use a simple property referred to as “Fredman’s trick”. The property is that $a + b < c + d \iff a - c < d - b$.

The trick is materialized as step 3, by computing the sorted order on \mathcal{D} , we can execute step 4 without making any additional query since if $a_i, a'_i \in \mathcal{A}_i$ and $a_j, a'_j \in \mathcal{A}_j$, $a_i + a_j < a'_i + a'_j \iff a_i - a'_i < a'_j - a_j$. And this time, it can be subquadratic. Applying the results of Theorem 6.5 again, the total number of queries used during step 3 is $\log |\Gamma| + 2N$, where Γ is the set of cells of an arrangement of hyperplanes of the kind $a_i - a'_i + a_j - a'_j = 0$ in \mathbb{R}^{2n} . In this case $N = \binom{n}{g} g^2 = gn$ and $|\Gamma| \leq ((\frac{n}{g})^2 (g^2)^2)^{2n} = (n^2 g^2)^{2n} \leq (n^4)^{2n} = n^{8n}$. Hence, the number of queries used during step 3 is $O(n \log n + gn)$. We say that step 3 is a comparison efficient way of accomplishing step 4.

Grønlund and Pettie [38] generalize this result further by reducing k -LDT to an unbalanced 3SUM problem, and apply a three-set version of their decision tree to solve it. The details of this reduction technique are explained in the next section.

Finally, for the case of $k = 3$, they provide two subquadratic 3SUM algorithms. A deterministic one running in $O(n^2 / (\log n / \log \log n)^{2/3})$ time and a randomized one running in $O(n^2 (\log \log n)^2 / \log n)$ time with high probability, the first one refuting the 3SUM conjecture. We do not detail those here, our focus being the query complexity.

7.6 $(2k - 2)$ -linear Decision Trees

We first show a transformation to construct an instance of 3SUM from an instance of k -LDT for the case where k is odd. Then we show that in the k -linear decision tree model one can solve k -LDT optimally using the standard quadratic algorithm for 3SUM. After that, we give a short argument to explain that the same kind of technique can be applied when k is even. Finally, we illustrate how the decision tree for 3SUM of Grønlund and Pettie [38] gives a better lower bound, for odd k , using $(2k - 2)$ -linear decision trees.

The goal of this section is to prove the following result

Theorem 7.4 (Grønlund and Pettie [38]). *When $k \geq 3$ is odd, there is a $(2k - 2)$ -linear decision tree for k -LDT with depth $O(n^{k/2} \sqrt{\log n})$.*

The standard three-set version of the quadratic 3SUM algorithm for input sets \mathcal{A} , \mathcal{B} , and \mathcal{C} completes after $O(|\mathcal{C}|(|\mathcal{A}| + |\mathcal{B}|))$ steps. We show how to solve any instance of a k -LDT problem with $k \geq 3$ odd in time $O(n^{\frac{k+1}{2}})$.

Given an input set \mathcal{S} of n real numbers, a set of k real coefficients

$\{\alpha_1, \dots, \alpha_k\}$, and a real term α_0 we construct the sets \mathcal{A} , \mathcal{B} , and \mathcal{C} as follows

$$\begin{aligned}\mathcal{A} &= \left\{ \alpha_0 + \alpha_1 s_1 + \dots + \alpha_{\frac{k-1}{2}} s_{\frac{k-1}{2}} : s_i \in \mathcal{S} \right\}, \\ \mathcal{B} &= \left\{ \alpha_{\frac{k+1}{2}} s_{\frac{k+1}{2}} + \dots + \alpha_{k-1} s_{k-1} : s_i \in \mathcal{S} \right\}, \\ \mathcal{C} &= \left\{ \alpha_k s_k : s_k \in \mathcal{S} \right\}.\end{aligned}$$

Each element of \mathcal{A} and \mathcal{B} has exactly $\frac{k-1}{2}$ linear terms. Elements of \mathcal{A} have an additional α_0 independent term. Looking at all possible combinations of s_i with α_j we know that $|\mathcal{A}| = |\mathcal{B}| = O(n^{\frac{k-1}{2}})$, whereas $|\mathcal{C}| = n$. As a remark, note that by simply omitting α_j factors we obtain a similar reduction from k -SUM.

If we use the quadratic 3SUM algorithm to solve the instance we constructed, it makes k -linear queries since summing an element of \mathcal{A} with an element of \mathcal{B} and comparing this sum to an element of \mathcal{C} involves $2\frac{k-1}{2} + 1$ linear terms. The running time of the 3SUM algorithm on this input is $O(n \cdot 2 \cdot n^{\frac{k-1}{2}}) = O(n^{\frac{k+1}{2}})$.

Note that to use this algorithm we need to construct sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$. Moreover we have to provide sorted structures for \mathcal{A} and \mathcal{B} . However the complexity of this preprocessing is only $O(n^{\frac{k-1}{2}} \log n)$.

For $k \geq 4$ even one constructs only two sets \mathcal{A} and \mathcal{B} of cardinality $|\mathcal{A}| = |\mathcal{B}| = O(n^{\frac{k}{2}})$. By doing this we transformed our instance of k -LDT into an instance of 2SUM. 2SUM can be solved in $O(n^{\frac{k}{2}} \log n)$ with only $O(n^{\frac{k}{2}})$ comparisons using Lambert's algorithm [57].

Grønlund and Pettie [38] explain that one can generalize their new decision tree for 3SUM from the one-set version to a three-set version. This version of the decision tree has complexity

$$O(g(|\mathcal{A}| + |\mathcal{B}|) + g^{-1}|\mathcal{C}|(|\mathcal{A}||\mathcal{B}|) \log g).$$

Hence, if one chooses $g = \sqrt{n \log n}$ one obtains a $O(n^{\frac{k}{2}} \sqrt{\log n})$ decision tree for k -LDT with k odd which improves the best known lower bound.

Also, we can note that this way of doing makes use of $(2k - 2)$ -linear queries instead of the k -linear queries used in the standard 3SUM algorithm or in Lambert's algorithm. Indeed, since $|\mathcal{A}| = |\mathcal{B}| = O(n^{\frac{k-1}{2}})$, when we sort all $(\mathcal{A}_i - \mathcal{A}_i)$ and $(\mathcal{B}_i - \mathcal{B}_i)$ we are making $4\frac{k-1}{2} = (2k - 2)$ -linear queries.

Note that with both the classical algorithm and the algorithm of Grønlund and Pettie [38], the “time” required to build an instance of 2SUM or 3SUM from an instance of k -LDT is less than the depth of the decision trees used to

solve the problem. One must also note that it is possible to store or compute which elements of the original input set \mathcal{S} match a particular element of the sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$, allowing one to use the output of the 2SUM or 3SUM algorithm run on the constructed instance to find the output of the original k -LDT instance.

Chapter 8

Application of Meiser's Algorithm

As we have seen in [Section 6.4](#), it is possible to argue about an upper bound on the number of possible linear extensions of the poset $X + Y$ by regarding Sorting $X + Y$ as a point location problem in an arrangement of hyperplanes. We formally generalize this observation to the k -SUM problem. Since we saw earlier in [Section 5.5](#) that sorting $X + Y$ is a special case of 4LDT, this generalization feels natural.

8.1 Reduction to a Point Location Problem

We describe a problem¹ which can be used to model various kind of set sum problems, including subset sum, k -SUM, and k -LDT.

Given a set of hyperplanes \mathcal{H} and a point x in the n -dimensional space \mathbb{R}^n , the problem is to locate x relatively to the arrangement of hyperplanes $\mathcal{A}(\mathcal{H})$. This arrangement divides the space into cells of dimension $0, \dots, n-1$ and the goal is to determine which cell contains the point x . This amounts to determining for each hyperplane H_i whether the point x lies above, on, or below H_i . Formally, we define the point location problem in an arrangement of hyperplanes as follows

Problem 8.1 (Point location problem in an arrangement of hyperplanes). *Given a point $x \in \mathbb{R}^n$ and a set of hyperplanes $\mathcal{H} = \{H_i: 1 \leq i \leq m\}$ determine the position vector of x , $pv(x) \in \{-, 0, +\}^n$, where $pv_i(x) = \sigma, \sigma \in \{-, 0, +\}$ iff $x \in H_i^\sigma$. We define H_i^0, H_i^- and H_i^+ to be respectively H_i itself, the half-space below H_i and the half-space above H_i .*

¹Note that we already used some of the ideas of this section several times in the previous chapters.

In the k -SUM problem, we want to decide whether there exists a subset \mathcal{S} of the universe set $\mathcal{U} = \{u_1, \dots, u_n\} \subset \mathbb{R}$ such that \mathcal{S} contains exactly k elements of \mathcal{U} and the sum of those k elements is equal to 0. For a given such subset \mathcal{S} we name those elements s_1, \dots, s_k .

We give a procedure to convert any instance of the k -SUM problem into an instance of the point location problem in an arrangement of hyperplanes. By agreeing on an order of the elements of \mathcal{U} (for example, the total order on the real numbers) we can associate a unique n -tuple to each universe set \mathcal{U} , and thus a unique point x in \mathbb{R}^n for each \mathcal{U} . For each $\mathcal{S}_i = \{s_1, \dots, s_k\}$ we need to find whether the sum $s_1 + \dots + s_k$ is equal to zero. In our new representation of the problem this boils down to deciding whether our point x lies on the hyperplane of equation $x_{i_1} + \dots + x_{i_k} = 0$. There are less than n^k such hyperplanes, and to solve the problem we have to decide whether x lies on one of them.

We just showed how to reduce k -SUM to a point location problem in an arrangement of hyperplanes. In the next section, we detail a location algorithm due to Meiser [63] that can be used to solve k -SUM efficiently for any k . There exist several methods to solve this problem. We describe Meiser's Algorithm [63] since it is the first algorithm that solves this point location problem in time polynomial in both n and $\log m$. This particular algorithm gives us the opportunity to prove that the decision tree complexity of subset sum is polynomial.

8.2 Meiser's Algorithm

Since the beginning of this writing, we were only interested in the number of questions we ask to an oracle, and we will continue to take the same point of view here. We wonder how many times we need to ask an oracle whether a point lies above, on or under a certain hyperplane, to solve k -SUM using the reduction introduced in the previous section.

Given a set of hyperplanes \mathcal{H} and an input vertex x in \mathbb{R}^n , Meiser's algorithm [63] determines the position vector² $pv(x) \in \{-, 0, +\}^m$ of x inside the arrangement of hyperplanes $\mathcal{A}(\mathcal{H})$. We show how to solve k -SUM using $O(kn^3 \log^3 n)$ n -variate linear queries with this algorithm.

For k fixed, \mathcal{H} is the set of hyperplanes having equation involving exactly k coordinates of \mathbb{R}^n with coefficient 1, we have thus $|\mathcal{H}| = m = \binom{n}{k}$. We can do this without involving the input vertex at all, this costs us 0 queries.

²For this section, notation is strongly inspired by the notation used for the detailed explanation of the same problem in Bürgisser et al. [13]. The relevant section from this reference is numbered 3.4 and titled *Fast Point Location in Arrangement of Hyperplanes*.

We give each hyperplane an orientation so that when the point x is below the hyperplane H_i , it is in the lower half-space defined by H_i denoted H_i^- . When the point x is above the hyperplane H_i it is in the upper half-space defined by H_i denoted H_i^+ . We define H_i^0 to be the hyperplane H_i itself. The i^{th} component of the position vector is $pv_i(x) = \sigma$ if and only if $x \in H_i^\sigma$.

Our first idea to determine the components of $pv(x)$ is to iteratively select a random subset of hyperplanes $\tilde{\mathcal{H}}$ of size $O(\text{poly}(n))$ for which the relative position of x is not determined yet. Once we have computed the position vector of x for this subset $\tilde{\mathcal{H}}$ we have effectively determined in which cell³ of the arrangement $\mathcal{A}(\tilde{\mathcal{H}})$ x lies.

We are left with hyperplanes of the set $\mathcal{H} \setminus \tilde{\mathcal{H}}$. They can be partitioned in two disjoint sets: a set \mathcal{M} of hyperplanes that meet the cell of $\tilde{\mathcal{H}}$ containing x and a set \mathcal{O} for the others. The relative position of x to each hyperplane belonging to \mathcal{O} can be deduced without the need to query the oracle, the answer to the queries we would make can be computed by looking at the structure of the arrangement. For the first group of hyperplanes we call the procedure again with $\mathcal{H} \leftarrow \mathcal{M}$.

If we stop our explanation here we still have a worst-case $O(m)$ bound on the number of queries made. Hopefully, the next step of our explanation shows that it is possible to lower this bound.

The problem we have with our current algorithm is that there is no guarantee on the number of queries we avoid at each step. We use a result on ϵ -nets, attributed to Ken Clarkson by Bürgisser et al. [13], to make our algorithm achieve an upper bound of $O(n^3 \log^2 n \log m)$ queries.

Theorem 8.1 (Clarkson). *Given a set \mathcal{H} of m hyperplanes in \mathbb{R}^n and ϵ with the constraint that $0 < \epsilon < 1$, there exists a subset $\mathcal{N} \subseteq \mathcal{H}$ of size $O(\frac{n^2}{\epsilon} \log^2 \frac{n}{\epsilon})$ such that for every simplex S in \mathbb{R}^n , if the number of hyperplanes of \mathcal{H} intersecting S is strictly greater than $\epsilon|\mathcal{H}|$ then at least one of them belongs to \mathcal{N} .*

By applying a theorem due to Haussler and Welzl [41], Bürgisser et al. [13] also prove (in addition to Theorem 8.1) that it is possible to construct \mathcal{N} with high probability using a random uniform sampling on \mathcal{H} , meaning the following corollary holds

Corollary 8.2. *If we choose $O(\frac{n^2}{\epsilon} \log^2 \frac{n}{\epsilon})$ hyperplanes uniformly at random from our m hyperplanes and denote this selection \mathcal{H}^* then for any simplex intersected by more than $\epsilon|\mathcal{H}|$ hyperplanes of \mathcal{H} there is a high probability that at least one of them is contained in \mathcal{H}^* .*

³Note that we always consider that x is inside a cell that is bounded. This can be implicitly or explicitly simulated.

If we take the contraposition of this corollary we obtain the following one

Corollary 8.3. *If there is no hyperplane in \mathcal{H}^* intersecting a given simplex, then, with high probability, the number of hyperplanes of \mathcal{H} intersecting the simplex is less or equal to $\epsilon|\mathcal{H}|$.*

We can use this corollary in the algorithm we were describing earlier. We make two changes to the previous algorithm, we explicitly define the cardinality of $\tilde{\mathcal{H}}$ and we include an additional step where we replace cell C with a simplex S .

Remember $\tilde{\mathcal{H}}$ is the set of hyperplanes for which we query the oracle in the current step of the algorithm. We choose $|\tilde{\mathcal{H}}| = O(\frac{n^2}{\epsilon} \log^2 \frac{n}{\epsilon})$ and instead of discarding hyperplanes not meeting the cell C of the arrangement $\mathcal{A}(\tilde{\mathcal{H}})$ containing x , we first refine the cell around point x . We do so by computing a simplex S inscribed in C and containing x . Then we discard hyperplanes that do not meet this simplex.

By proceeding this way we have the guarantee that we keep at most a constant fraction ϵ of the hyperplanes and thus the total number of queries made to determine the enclosing cell of all steps is $O(n^2 \log^2 n \log m)$. However, we still need to explicit how we find a simplex S containing x and inscribed in C .

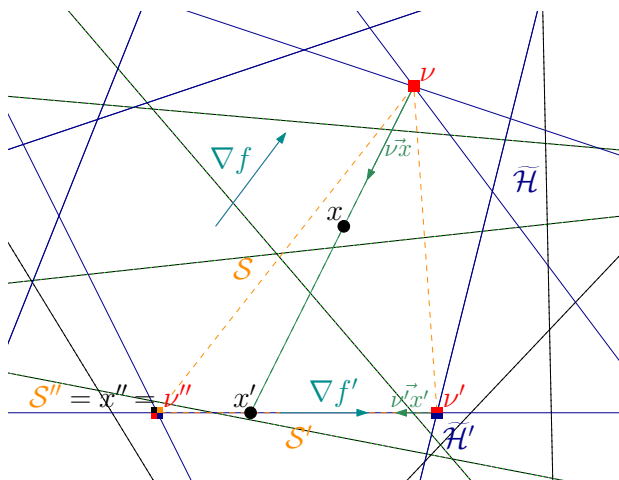
We explain how to build S . The algorithm can be sketched as follows

Algorithm 8.1 (Building S).

input A point x and a set of hyperplanes $\tilde{\mathcal{H}}$ in \mathbb{R}^n .

1. Find a vertex ν of the cell containing x , ν is one of the vertices of our simplex.
2. Compute x' , the projection of x along $\vec{\nu x}$ on the boundary of C .
3. Let H_θ denote the hyperplane in $\tilde{\mathcal{H}}$ containing x' . Compute $\tilde{\mathcal{H}}'$ as the intersection of all hyperplanes of $\tilde{\mathcal{H}} \setminus \{H_\theta\}$ with H_θ .
4. Induction on x' and $\tilde{\mathcal{H}}'$ in \mathbb{R}^{n-1} , store result in S' .
5. Return S , the convex hull of $S' \cup \{\nu\}$.

We can solve step 1 by using linear programming with our $O(n^2 \log^2 n)$ hyperplanes and the answers to queries $x \in ? H_i^\sigma, \sigma \in \{-, 0, +\}$ as constraints of the linear program. We arbitrarily choose an objective function with a gradient non-orthogonal to all hyperplanes in $\tilde{\mathcal{H}}$ and look for the optimal solution. The optimal solution being the intersection of n hyperplanes from the arrangement, its coordinates are independent of the exact location of x inside its cell and thus this step involves no query at all.



In step **2** we find the projection of x on one of the faces of C by computing the closest hyperplane of $\tilde{\mathcal{H}}$ to x in direction $\nu\vec{x}$. This is done by projecting x on every hyperplane of $\tilde{\mathcal{H}}$ and then computing the distance between x and his projections, keeping the projection that is the closest in the correct direction as the projection on C . Each such projection and distance computation counts as one query since this computation could be used to forge a query. The number of projections to compute is $|\tilde{\mathcal{H}}|$ and hence the query complexity of this step is $O(\frac{n^2}{\epsilon} \log^2 \frac{n}{\epsilon})$.

In step 4 the base case is $n = 0$. In \mathbb{R}^0 we have only one point and this point is the last vertex of our simplex. The recursion depth is n , the dimension of R^n , and thus the total number of queries made to compute S is $O(n^3 \log^2 n)$.

Figure 8.1 shows a complete step of Meiser’s algorithm. Let us summarize

the algorithm

Algorithm 8.2 (Meiser's algorithm).

input $x \in \mathbb{R}^n$, the point to be located.

1. Compute the position of $O(n^2 \log^2 n)$ hyperplanes relatively to x , effectively computing cell C containing x .
2. Recursively build simplex S containing x and inscribed in C .
3. For any hyperplane H_i not meeting the simplex, deduce $pv_i(x)$.
4. Discard all hyperplanes that do not meet the inside of the simplex.
5. Recurse on hyperplanes that are left.

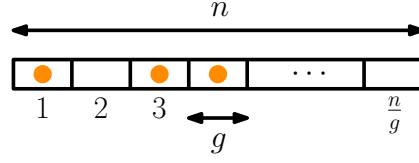
Since the complexity of step 1 is $O(n^2 \log^2 n)$, the complexity of step 2 is $O(n^3 \log^2 n)$, the complexity of steps 3 and 4 is 0, and the induction depth is $O(\log m)$, the total complexity of this algorithm is $O(n^3 \log^2 n \log m)$. For our k -SUM problem, $m = \binom{n}{k}$, and thus the complexity is $O(kn^3 \log^3 n)$. Note that the overall complexity in the Random Access Machine model is polynomial.

In the next section we apply the same algorithm to solve the subset sum problem and show that in our sufficiently powerful computing model the subset sum problem has polynomial complexity.

8.3 Strongly Polynomial Nonuniform Algorithm for Subset Sum

We apply Meiser's algorithm to the subset sum problem. In the subset sum problem with input set $\mathcal{S} \subset \mathbb{R}$ of cardinality $|\mathcal{S}| = n$ we want to decide whether there exists a subset $\mathcal{S}' \subseteq \mathcal{S}$ whose elements sum up to zero. Hence, we have to search through 2^n subsets of \mathcal{S} . Reducing to the point location problem in an arrangement of hyperplanes, we have to locate input point $x \in \mathbb{R}^n$ inside an arrangement of 2^n hyperplanes, that is, all hyperplanes with equation having combinations of 0 and 1 as coefficients. Meiser's algorithm solves this problem using only $O(n^4 \log^2 n)$ queries, that is, poly-time in our decision tree model.

The subset sum problem is of interest here because it is an NP-complete problem [52], while our model of computation gives a polynomial time algorithm. When the algorithm is described in [13] it is not assumed that the analysis is made in a different model than the classical RAM model but rather that the dimension of \mathbb{R}^n , n , is not part of the input but known in advance, allowing one to construct an efficient data structure for $\mathcal{A}(\mathcal{H})$ for any

Figure 8.2: Solving k -SUM for blocks 1, 3, and 4 only.

instance of the problem with fixed n . This is a consequence of the nonuniformity of the model we use as we explained in [Section 1.2](#). In our case, the algorithm is also *strongly polynomial* since the computations involved produce only real numbers whose sizes are bounded by the size of the input.

Meiser [63], Bürgisser et al. [13] give other examples of NP-complete problems that can be solved with Meiser's algorithm in $\text{poly}(n)$ queries in the linear decision tree model.

8.4 Solve k -SUM using only $o(n)$ -linear Queries

We have seen that using Meiser's algorithm one can solve the k -SUM problem with $O(n^3 \log^3 n)$ queries. During the construction of the simplex we need at each step of this algorithm, we use projection operations on the input point and the query hyperplanes. It is thus possible that some query we perform involves up to n linear terms of the input. We show that we can restrict ourselves to $o(n)$ -linear queries with the addition of a factor n to the complexity of the algorithm.

We look at [Figure 8.2](#). We represent input set \mathcal{S} as a vector of size n . The trick is the following. Instead of using Meiser's algorithm once to solve the problem of size n , we divide the input vector in blocks of size g and solve a k -SUM problem for all combinations of k blocks. There are n/g such blocks hence the number of problems to solve is $\binom{n/g}{k} = O((\frac{n}{g})^k)$.

The complexity of solving all the problems is thus

$$O\left(\left(\frac{n}{g}\right)^k (kg)^3 \log^3(kg)\right)$$

Since a query can only involve elements contained in the union of k disjoint subsets of size g , the queries made are (kg) -linear. Below is a pseudo-code description of the algorithm.

Algorithm 8.3 (Parameterizable k -SUM algorithm).

input Set \mathcal{S} .

1. Partition \mathcal{S} into disjoint subsets $\mathcal{S}_1, \dots, \mathcal{S}_{\lceil \frac{n}{g} \rceil}$ of size at most g .
2. For each of the $\binom{\lceil \frac{n}{g} \rceil}{k}$ possible ways to choose k subsets $\mathcal{S}_{i_1}, \dots, \mathcal{S}_{i_k}$ of \mathcal{S} without replacement:
 - 2.1. Solve a k -SUM instance with input $\mathcal{S}_{i_1} \cup \dots \cup \mathcal{S}_{i_k}$ using Meiser's algorithm.
 - 2.2. If the run of Meiser's algorithm yields a solution, output that solution and stop.

We want to parameterize g so that the queries are $o(n)$ -linear. A possible solution is to choose $g = n^{\frac{k-1}{k}} = n^{1-\frac{1}{k}}$. With this parameterization, the queries become $(kn^{1-\frac{1}{k}})$ -linear and we obtain a complexity of

$$O(n(kn^{1-\frac{1}{k}})^3 \log^3(kn^{1-\frac{1}{k}})).$$

Note that for any $\alpha < k$ if we choose $g = n^{\frac{k-\alpha}{k}} = n^{1-\frac{\alpha}{k}}$ we end up with $(kn^{1-\frac{\alpha}{k}})$ -linear queries and a total complexity of $O(n^\alpha(kn^{1-\frac{\alpha}{k}})^3 \log^3(kn^{1-\frac{\alpha}{k}}))$. Like the lower bound given by Ailon and Chazelle [1], it gives us a “parameterizable trade-off” between time complexity and query complexity.

Bibliography

- [1] Ailon, N. and Chazelle, B. (2005). Lower bounds for linear degeneracy testing. *J. ACM*, 52(2):157–171.
- [2] Bareiss, E. H. (1968). Sylvester’s identity and multistep integer-preserving gaussian elimination. *Mathematics of computation*, 22(103):565–578.
- [3] Barequet, G. and Har-Peled, S. (2001). Polygon-containment and translational min-Hausdorff-distance between segments sets are 3SUM-hard. *International Journal of Computational Geometry & Applications*, 11(04):465–474.
- [4] Barrera, A. H. (1996). Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 289–294. Carleton University Press.
- [5] Berge, C. and Chvátal, V. (1984). *Topics on perfect graphs*. North-Holland Amsterdam, The Netherlands.
- [6] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. (1973). Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461.
- [7] Bremner, D., Chan, T. M., Demaine, E. D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Pătraşcu, M., and Taslakian, P. (2012). Necklaces, convolutions, and $X + Y$. *Algorithmica*, pages 1–21.
- [8] Brightwell, G. (1999). Balanced pairs in partial orders. *Discrete Mathematics*, 201(1):25–52.
- [9] Brightwell, G. and Winkler, P. (1991). Counting linear extensions. *Order*, 8(3):225–242.
- [10] Brightwell, G. R., Felsner, S., and Trotter, W. T. (1995). Balancing pairs and the cross product conjecture. *Order*, 12(4):327–349.

- [11] Brodnik, A., López-Ortiz, A., Raman, V., and Viola, A., editors (2013). *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*. Springer.
- [12] Buck, R. (1943). Partition of space. *American Mathematical Monthly*, pages 541–544.
- [13] Bürgisser, P., Clausen, M., and Shokrollahi, M. A. (1997). *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer.
- [14] Cardinal, J. and Fiorini, S. (2013). On generalized comparison-based sorting problems. In [11], pages 164–175.
- [15] Cardinal, J., Fiorini, S., Joret, G., Jungers, R. M., and Munro, J. I. (2010). An efficient algorithm for partial order production. *SIAM journal on computing*, 39(7):2927–2940.
- [16] Cardinal, J., Fiorini, S., Joret, G., Jungers, R. M., and Munro, J. I. (2013). Sorting under partial information (without the ellipsoid algorithm). *Combinatorica*, 33(6):655–697.
- [17] Chazelle, B. (1993). Cutting hyperplanes for divide-and-conquer. *Discrete & Computational Geometry*, 9(1):145–158.
- [18] Cibulka, J. (2011). On average and highest number of flips in pancake sorting. *Theoretical Computer Science*, 412(8):822–834.
- [19] Ciura, M. (2001). Best increments for the average case of shellsort. In *Fundamentals of Computation Theory*, pages 106–117. Springer.
- [20] Clarkson, K. L. (1987). New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, 2(1):195–222.
- [21] Cohen, D. S. and Blum, M. (1995). On the problem of sorting burnt pancakes. *Discrete Applied Mathematics*, 61(2):105–120.
- [22] Cover, T. M. and Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.
- [23] Csiszár, I., Koerner, J., Lovasz, L., Marton, K., and Simonyi, G. (1990). Entropy splitting for antiblocking corners and perfect graphs. *Combinatorica*, 10(1):27–40.

- [24] de la Vega, W. F., Frieze, A. M., and Santha, M. (1998). Average-case analysis of the merging algorithm of Hwang and Lin. *Algorithmica*, 22(4):483–489.
- [25] Dietzfelbinger, M. (1989). Lower bounds for sorting of sums. *Theoretical Computer Science*, 66(2):137–155.
- [26] Dijkstra, E. W. (1982). Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3):223–233.
- [27] Dobkin, D. and Lipton, R. J. (1976). Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186.
- [28] Erickson, J. (1999). Lower bounds for linear satisfiability problems. *Chicago J. Theor. Comput. Sci.*, 1999.
- [29] Feller, W. (1967). A direct proof of stirling’s formula. *American Mathematical Monthly*, pages 1223–1225.
- [30] Ford, L. R. and Johnson, S. M. (1959). A tournament problem. *American Mathematical Monthly*, pages 387–389.
- [31] Frame, J., Robinson, G. d. B., Thrall, R., et al. (1954). The hook graphs of the symmetric group. *Canad. J. Math*, 6(316):C324.
- [32] Frazer, W. D. and Bennett, B. (1972). Bounds on optimal merge performance, and a strategy for optimality. *Journal of the ACM (JACM)*, 19(4):641–648.
- [33] Fredman, M. L. (1976). How good is the information theory bound in sorting? *Theoretical Computer Science*, 1(4):355–361.
- [34] Gajentaan, A. and Overmars, M. H. (2012). On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 45(4):140–152.
- [35] Goldstine, H. H. and Von Neumann, J. (1948a). *Coding of some Combinatorial (Sorting) Problems*, pages 49–68. In [36].
- [36] Goldstine, H. H. and Von Neumann, J. (1948b). *Planning and coding of problems for an electronic computing instrument*. Institute for Advanced Study.
- [37] Greene, C., Nijenhuis, A., and Wilf, H. S. (1979). A probabilistic proof of a formula for the number of Young tableaux of a given shape. *Advances in Mathematics*, 31(1):104–109.

- [38] Grønlund, A. and Pettie, S. (2014). Threesomes, degenerates, and love triangles. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 621–630. IEEE.
- [39] Hardin, R. H. (2001). The On-Line Encyclopedia of Integer Sequences. [A060854](#). Array $T(m, n)$ read by antidiagonals: $T(m, n)$ ($m \geq 1, n \geq 1$) = number of ways to arrange the numbers $1, 2, \dots, mn$ in an $m \times n$ matrix so that each row and each column is increasing.
- [40] Harper, L., Payne, T. H., Savage, J. E., and Straus, E. (1975). Sorting $X + Y$. *Communications of the ACM*, 18(6):347–349.
- [41] Haussler, D. and Welzl, E. (1987). ϵ -nets and simplex range queries. *Discrete & Computational Geometry*, 2(1):127–151.
- [42] Hoare, C. A. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.
- [43] Huber, M. (2006). Fast perfect sampling from linear extensions. *Discrete Mathematics*, 306(4):420–428.
- [44] Huffman, D. A. et al. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.
- [45] Hwang, F. and Lin, S. (1969). An analysis of ford and johnson’s sorting algorithm. In *Proc. Third Annual Princeton Conf. on Inform. Sci. and Systems*, pages 292–296.
- [46] Hwang, F. K. and Lin, S. (1972). A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput.*, 1(1):31–39.
- [47] Jiang, T., Li, M., and Vitányi, P. (2000). A lower bound on the average-case complexity of shellsort. *Journal of the ACM (JACM)*, 47(5):905–911.
- [48] Kahn, J. and Kim, J. H. (1995). Entropy and sorting. *Journal of Computer and System Sciences*, 51(3):390–399.
- [49] Kahn, J. and Linial, N. (1991). Balancing extensions via Brunn-Minkowski. *Combinatorica*, 11(4):363–368.
- [50] Kahn, J. and Saks, M. (1984). Balancing poset extensions. *Order*, 1(2):113–126.
- [51] Kaligosi, K., Mehlhorn, K., Munro, J. I., and Sanders, P. (2005). Towards optimal multiple selection. In *Automata, Languages and Programming*, pages 103–114. Springer.

- [52] Karp, R. M. (1972). Reducibility among combinatorial problems. In [65], pages 85–103.
- [53] King, J. (2004). A survey of 3SUM-hard problems. Not published.
- [54] Kislitsyn, S. (1968). A finite partially ordered set and its corresponding set of permutations. *Mathematical Notes of the Academy of Sciences of the USSR*, 4(5):798–801.
- [55] Knuth, D. (1998). *The art of computer programming: Sorting and Searching, 2nd edn., vol. 3*. Addison-Wesley, Reading.
- [56] Körner, J. (1973). Coding of an information source having ambiguous alphabet and the entropy of graphs. In *6th Prague conference on information theory*, pages 411–425.
- [57] Lambert, J.-L. (1990). Sorting the sums $(x_i + y_j)$ in $O(n^2)$ comparisons. In *STACS 90*, pages 195–206. Springer.
- [58] Leiserson, C. E., Rivest, R. L., Stein, C., and Cormen, T. H. (2001). *Introduction to algorithms*. MIT press.
- [59] Linial, N. (1984). The information-theoretic bound is good for merging. *SIAM Journal on Computing*, 13(4):795–801.
- [60] Manacher, G. K. (1979). The ford-johnson sorting algorithm is not optimal. *Journal of the ACM (JACM)*, 26(3):441–456.
- [61] Martelli, A. (2006). *Python in a Nutshell*. O'Reilly Media, Inc.
- [62] Matoušek, J. (2002). Lectures on discrete geometry, volume 212 of. *Graduate Texts in Mathematics*, pages 323–362.
- [63] Meiser, S. (1993). Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303.
- [64] Meyer auf der Heide, F. (1984). A polynomial linear search algorithm for the n -dimensional knapsack problem. *Journal of the ACM (JACM)*, 31(3):668–676.
- [65] Miller, R. E. and Thatcher, J. W., editors (1972). *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*, The IBM Research Symposia Series. Plenum Press, New York.

- [66] Mohanty, G. (1979). *Lattice path counting and applications*. Academic Press.
- [67] Moran, S. and Yehudayoff, A. (2015). A note on average-case sorting. *Order*, pages 1–6.
- [68] Mowshowitz, A. and Dehmer, M. (2012). Entropy and the complexity of graphs revisited. *Entropy*, 14(3):559–570.
- [69] Musser, D. R. (1997). Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993.
- [70] O’Rourke, J. (2012). Problem 41: Sorting $X + Y$ (pairwise sums). <http://cs.smith.edu/~orourke/TOPP/P41.html#Problem.41>.
- [71] Peczarski, M. (2006). The gold partition conjecture. *Order*, 23(1):89–95.
- [72] Peczarski, M. (2008). The gold partition conjecture for 6-thin posets. *Order*, 25(2):91–103.
- [73] Sedgewick, R. (1980). *Quicksort*, volume 492. Dissertations-G.
- [74] Sedgewick, R. (1996). Analysis of shellsort and related algorithms. In *Algorithms—ESA ’96*, pages 1–11. Springer.
- [75] Sedgewick, R. and Wayne, K. (2011). *Algorithms, 4th Edition*. Addison-Wesley.
- [76] Shell, D. L. (1959). A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32.
- [77] Simonyi, G. (1995). Graph entropy: a survey. *Combinatorial Optimization*, 20:399–441.
- [78] Simonyi, G. (2001). Perfect graphs and graph entropy. an updated survey. *Perfect graphs*, pages 293–328.
- [79] Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686.
- [80] Sloane, N. J. A. (no date). The On-Line Encyclopedia of Integer Sequences. [A001855](#). Sorting numbers: maximal number of comparisons for sorting n elements by binary insertion.
- [81] Stanley, R. P. (2011). *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition.

- [82] Steiger, W. and Streinu, I. (1995). A pseudo-algorithmic separation of lines from pseudo-lines. *Information processing letters*, 53(5):295–299.
- [83] van Emde Boas, P. (1975). Preserving order in a forest in less than logarithmic time. In *FOCS*, pages 75–84.
- [84] van Lamoen, F. (no date). The On-Line Encyclopedia of Integer Sequences. [A039622](#). Number of $n \times n$ Young tableaux.
- [85] Wegener, I. (1993). BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98.
- [86] Wild, S. and Nebel, M. E. (2012). Average case analysis of java 7’s dual pivot quicksort. In *Algorithms–ESA 2012*, pages 825–836. Springer.
- [87] Wild, S., Nebel, M. E., and Neininger, R. (2013). Average case and distributional analysis of java 7’s dual pivot quicksort. *CoRR*, abs/1304.0988.
- [88] Williams, J. W. J. (1964). Algorithm-232-heapsort.
- [89] Yaroslavskiy, V. (2009). Dual-pivot quicksort. *Research Disclosure RD539015 (Sept., 2009)*, <http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf>.
- [90] Zaguia, I. (2012). The $1/3$ - $2/3$ conjecture for N -free ordered sets. *Electr. J. Comb.*, 19(2):29.