CACHE-OBLIVIOUS SEARCHING AND SORTING IN MULTISETS

by

Arash Farzan

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We study three problems related to searching and sorting in multisets in the cache-oblivious model: Finding the most frequent element (the mode), duplicate elimination and finally multi-sorting. We are interested in minimizing the cache complexity (or number of cache misses) of algorithms for these problems in the context under which the cache size and block size are unknown.

We start by showing the lower bounds in the comparison model. Then we present the lower bounds in the cache-aware model, which are also the lower bounds in the cache-oblivious model. We consider the input multiset of size N with multiplicities $N_1, \ldots, N_k$. The lower bound for the cache complexity of determining the mode is $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right)$ where f is the frequency of the mode and M, B are the cache size and block size respectively. Cache complexities of duplicate removal and multi-sorting have lower bounds of $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}\frac{N_i}{B}\right)$.

We present two deterministic approaches to give algorithms: selection and distribution. The algorithms with these deterministic approaches differ from the lower bounds by at most an additive term of $\frac{N}{B}\log\log M$. However, since $\log\log M$ is very small in real applications, the gap is tiny. Nevertheless, the ideas of our deterministic algorithms can be used to design cache-aware algorithms for these problems. The algorithms turn out to be simpler than the previously-known cache-aware algorithms for these problems.

Another approach to design algorithms for these problems is the probabilistic approach. In contrast to the deterministic algorithms, our randomized cache-oblivious algorithms are all optimal and their cache complexities exactly match the lower bounds.

All of our algorithms are within a constant factor of optimal in terms of the number of comparisons they perform.

Acknowledgments

I would like to thank J. I. Munro, my thesis supervisor, for helping me all the way in the research towards this thesis. We spent a lot of time discussing different ideas and problems. His guidance was absolutely essential and his suggestions were invaluable in this work. I appreciate his willingness for discussions, even during his busiest times, whenever I sought help from him. I also appreciate the amount of time he put into turning my initial draft into a thesis.

I would like to also thank T.M. Chan and A. Lopez-Ortiz, my thesis readers, for fixing the thesis and making it more readable. Finally I would like to thank N. Nishimura, my former supervisor, for her guidance in choosing a research topic, and connecting me to faculty members with the same research interests.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Preliminaries

The memory in modern computers consists of multiple levels: registers, multiple levels of cache memories, main memory, disk, etc. The cache-oblivious model is a simple and elegant model that has proved to be successful in analyzing the algorithms in hierarchical memory models [6].

Traditionally, algorithms were analyzed in a random access memory model (RAM) in which the memory is assumed to be flat with a uniform access time. However, the ever-growing difference between access times of different levels of a memory hierarchy makes the RAM model ineffective (e.g., level-two cache is roughly 100 times faster than main memory and main memory is roughly $1,000,000$ times faster than disk [3].) Hierarchical memory models have been introduced to tackle this problem. These models usually suffer from the complexity of having too many parameters. Consequently, the algorithms in these models are too complicated and are tailored for a specific hardware configuration.

The cache-oblivious model is a simple hierarchical memory model that avoids any hardware configuration parametrization. It is known that if an algorithm in this model performs optimally on this two-level hierarchy, it will perform optimally on any level of a multiple-level hierarchy.

In this thesis, we will study various problems regarding the searching and sorting in multisets, and design cache-oblivious algorithms for them.

1.1 Hierarchical Memory Models

The *cache-aware (DAM) model* [1] is the simplest of hierarchical memory models. We have only two levels of memory in the model. On the first level, there is a cache memory of size M which is divided into $\frac{M}{B}$ blocks of size B; on the second level there is an arbitrarily large memory with the same block size. A word must be present in the cache to be accessed. If it is not in the cache, we say a *cache miss/fault* has occurred, and in this case the block containing the requested word must be brought in from the main memory. The block can be placed anywhere in the cache (i.e., it is fully associative.) If all blocks in the cache are occupied, a block is thrown out of the cache and replaced by the block containing the requested word. Algorithms in this model, have full control over the block replacement policy. In other words, algorithms choose where to place the blocks in the cache, and which block to evict from the cache.[1] Algorithms in this model are fully aware of the values of memory parameters M, B. We denote these algorithms as *cache-aware algorithms*.

The *cache-oblivious model* is the same as the DAM model except that algorithms have no knowledge of hardware configuration parameters and in particular they are not aware of the values M, B. We denote these latter algorithms as *cache-oblivious algorithms*. Cache-oblivious algorithms must work independently of the values M and B, so they can be run on any hardware configuration without any modification to the algorithm itself. The block replacement policy is assumed to be the off-line optimal one; however, using a more realistic replacement policy such as the least recently used policy (LRU) increases the number of cache misses by only a factor of two if the cache size is also doubled [11].

Cache-oblivious algorithms nicely model a multi-level memory hierarchy. A cache of size M and block size B at level i of the memory hierarchy behaves similarly to a cache with the same size and block size in a two-level memory hierarchy that serves

[1] Cache is a misnomer, though standard terminology, as the word "cache" suggests it should be hidden from the algorithm. Nevertheless as it makes the cache-aware and cache-oblivious model similar to each other, we will use the word "cache" to call the first-level memory even in the cache-aware model.

the same memory accesses [6]. This property of cache-oblivious algorithms makes them extremely useful; algorithm designers think only about two levels of memory and develop algorithms that perform efficiently in each level of the memory hierarchy as well.

Cache complexity of an algorithm is the number of cache misses the algorithm causes or equivalently the number of block transfers it incurs between these two levels. In this thesis, the size of cache is always denoted by M and the size of a block is always denoted by B. We are only interested in the asymptotic cache complexity of algorithms, our lower and upper bounds are all asymptotic. That is our concern is with the order of magnitude of the cache complexities and we ignore constant factors.

1.2 The Multiset Searching and Sorting Problems

In this thesis, we consider several problems related to multisets and design cache-oblivious algorithms for them. A multiset is a generalization of a set in which repetition is allowed, so we can have several items with the same key value. We denote the items in the multiset as *elements*.

In this thesis, elements are assumed to be objects each of which contains a key field showing the value of the element. Elements are considered to be atomic and unbreakable; the integrity of an element is always maintained throughout handling. Each element takes up one unit space for storage. The value of an element is the value of its key field. For the sake of brevity, we will use the terms "elements" and "values of elements" interchangeably, so the reader can think of elements as being their values, unless explicitly stated otherwise. Two elements are called distinct if their values are different. Similarly, an element is called a duplicate of (or equal to) another element if they have the same value. Likewise, an element is defined to be greater than (less than) another element, if the value of the former element is greater than (less than) the value of the latter one.

The first problem we study is finding the most frequent element (the mode) in a

multiset. The other two problems are duplicate elimination and multi-sorting. Suppose we are given a multiset of size N in which there are k pairwise distinct elements $i_1, \ldots, i_k$ whose multiplicities are $N_1, \ldots, N_k$ respectively. The problem of reducing the original multiset to the set $\{i_1, \ldots, i_k\}$ is called duplicate elimination. Finally multi-sorting is the problem of sorting the elements of the multiset and outputting the list of elements in the sorted order.

1.3 Background and Previous Results

All three of these problems have been extensively studied in the comparison model which relates to the random access memory model (RAM). In the comparison model, we are only interested in the number of comparisons the algorithm performs. Munro and Spira [10] proved tight lower and upper bounds for the problems of determining the mode and multi-sorting in the comparison model. Deriving a lower and upper bound for the problem of duplicate elimination is not hard once we have the bounds for multi-sorting. Later, Arge et al. [4] proved tight lower and upper bounds for these three problems in an I/O model that relates to the cache-aware model.

Sorting was among the first problems that were studied in the cache-oblivious model. Frigo et al. [6] proved that a set of size N can be sorted in $O\left(\frac{N}{B} \max\left\{1, \log_{\frac{M}{B}} \frac{N}{B}\right\}\right)$ [2]. Aggarwal and Vitter [2] had shown that this is also a lower bound even for a cache-aware algorithm. However, there is an assumption in the algorithm of Frigo et al. that is a common assumption in cache-oblivious algorithms: This assumption, known as the *tall-cache assumption*, is that $M = \Omega\left(B^2\right)$. Obviously their algorithm does not perform optimally in sorting the multiset, as they do not take advantage of the fact that there might be repetitions in the input list. We will use their results in our algorithms, so we will assume the tall-cache assumption as well.

[2] Throughout this thesis, the base of a logarithm is given only when required. The use of log is used to infer the base is an arbitrary constant greater than one. lg denotes the logarithm base 2. ln denotes the natural logarithm.

Under the tall cache assumption $\log_{\frac{M}{B}} \frac{N}{B}$ can be simplified as $\log_{\frac{M}{B}} N$. Although we assume the prsence of a tall cache throughout this thesis, we avoid using the shorter form of such logarithms to maintain the generality of the results.

The rest of this thesis is organized as follows: We will present the lower bounds for the problems in the comparison model in Chapter 2. We will then present the lower bounds in the cache-aware model in Chapter 3. Finally in Chapter 4, which is the contribution of the author, we will give cache-oblivious algorithms for each of the problems.

Chapter 2

Lower Bounds in the Comparison Model

In this chapter we will present the lower bounds for the three problems in the comparison model. Munro and Spira [10] first proved lower bounds for the two problems of sorting a multiset and determining the mode. The problem of duplicate elimination is quite similar to multisorting and once we have a lower bound for multisorting, proving a lower bound for duplicate elimination is not hard.

We are only interested in asymptotic lower bounds whereas Munro and Spira [10] proved exact lower bounds; therefore, our proofs are simpler than theirs although the ideas are the same.

The rest of this chapter is organized as the following: We will first show a lower bound on multisorting in Section 2.1. Then we will prove a lower bound for the duplicate elimination problem in Section 2.2. Finally we will present a lower bound for determining the mode in Section 2.3.

2.1 Lower Bound on Multisorting

We are given a multiset of size N with k distinct elements $i_1, \ldots, i_k$ with multiplicities $N_1, \ldots, N_k$ respectively ($\sum_{i=1}^{k} N_i = N$.) We next give a lower bound for the number

of three-branch comparisons (i.e. $\{<, =, >\}$) required to sort the multiset. Theorems 2.1.1 and 2.2.1 are proven in [10] with the constant term in the $\Omega()$ being 1, the logarithm base 2 and a negative lower order term. As we are concerned only with order of magnitude, we can give simpler proofs.

Theorem 2.1.1 ([10]). *The average number of comparisons required to sort a multiset of size N with multiplicities $N_1, \ldots, N_k$ is:*

$$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right).$$

Proof. We will give an information theoretic lower bound. Consider a decision tree that determines the total ordering of the multiset. There are $\binom{N}{N_1, N_2, \ldots, N_k}$ ways to construct the list of the elements of a multiset that has multiplicities $N_1, \ldots, N_k$. Since the decision tree must distinguish each of these configurations from any other, the tree has at least $\binom{N}{N_1, N_2, \ldots, N_k}$ leaves.

It is well-known that a tree with fixed out-degree d and l leaves has an average height of $\log_d l$. Hence, the decision tree must have an average height of:

$$\begin{aligned} \textit{average height} &= \Omega\left(\log_3 \binom{N}{N_1, N_2, \ldots, N_k}\right) \\ &= \Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right). \end{aligned}$$

Therefore, the average number of comparisons required to sort the multiset is:

$$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$$

. □

2.2 Lower Bound on Duplicate Elimination

We will present a lower bound for duplicate removal. Given a multiset of size N consisting of distinct elements $i_1, \ldots, i_k$ with multiplicities $N_1, \ldots, N_k$, the goal is to reduce the multiset to the set $\{i_1, \ldots, i_k\}$.

The lower bound for duplicate elimination follows immediately from the lower bound for multisorting in Theorem 2.1.1.

Theorem 2.2.1 ([10]). *Duplicate elimination of a multiset of N with multiplicities $N_1, \ldots, N_k$ requires an average number of comparisons of*

$$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right).$$

Proof. We will show that after duplicate removal from a multiset, the total ordering among all elements must be known. Hence, the lower bound of multisorting also lower bounds the number of comparisons of duplicate removal.

Suppose, using some algorithm, we have removed all duplicates and obtained the set $\{i_1, \ldots, i_k\}$. First of all, the total ordering among $i_1, \ldots, i_k$ must be known, since we are dealing with the comparison model and any two of them must be known not to be equal. Secondly, any other element that is not present in the final set must be equal to one of the elements of the set and we must know which one, since we have removed it as a duplicate.

Thus, the total ordering must be known after the duplicate removal, and we showed in Theorem 2.1.1 that, on average, we need at least

$$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$$

comparisons to determine the total ordering. □

2.3 Lower Bound on Determining the Mode

In this section, we consider the problem of determining the mode and give a lower bound on the number of comparisons required to find the mode in a multiset. The theorem and proof are from [10].

Theorem 2.3.1 ([10]). *Determining the mode of a multiset of size N with multiplicities $N_1, \ldots, N_k$ requires an average number of comparisons of $\Omega\left(N \log \frac{N}{f}\right)$ where $f = \max_{i=1}^{k} N_i$ is the frequency of the mode.*

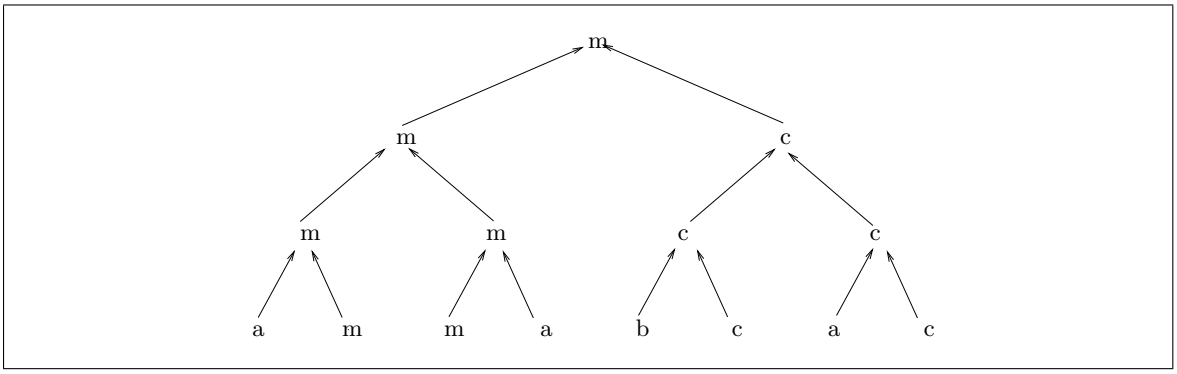

Figure 2.1: The tournament tree by which a multiset set is sorted after the mode is determined.

Proof. We will show that once the mode has been determined we can sort the multiset without too many comparisons. Since we proved a lower bound for sorting, it implies a lower bound for determining the mode.

Suppose the mode, with frequency f, has been found in the multiset. Consider those elements that have never lost in the comparisons so far (if $x < y$, we say x has lost to y.) There are at most f of these elements; otherwise they could be all equal and form a class of duplicates that has multiplicity greater than f.

We place these elements at the leaves of a balanced binary tree with f leaves and run a tournament to determine the maximum. We then remove the maximum elements and add any other elements that had lost only to the maxima. Again there can be at most f elements in the tree; otherwise they could form a class of more than f duplicates. We run the tournament again to select the next maxima and so on (see Figure 2.1 for an example.) Since the tree is always a balanced binary tree with f leaves, for each element we incur $\lceil \lg f \rceil$ comparisons, and therefore the total number of comparisons until the whole multiset gets sorted is $N \lceil \lg f \rceil$.

$N \lg f$ is an overestimate however. We can save some comparisons when equality between elements happen (i.e., $x = y$). When we remove off a maximum from the tree, we can remove all its duplicates along with it. To account for the number of comparisons we save, consider a class of duplicates of an element m with size N_m. Previously, we charged each of the elements for the height of the tree. Though, to

Table 2.1: The lower bounds in the comparison model

	Lower bound
Multisorting	$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$
Duplicate Elimination	$\Omega\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$
Determining the Mode	$\Omega\left(N \log \frac{N}{f}\right)$

draw them all off we require only $N_m - 1$ comparisons (more precisely equalities), plus those between an m and other (unequal) elements. The saving is minimized if all m's encounter each other at the highest possible point on the tree. Clearly, this is at depth $\lg N_m$. So the saving is $N_i \lg N_i - N_i$. Thus the number of comparisons we save in the total is:

$$\sum_{i=1}^{k} N_i \lg N_i - O(N).$$

Hence the total number of comparisons required to figure out the total ordering of the multiset when we have determined the mode is:

$$N \lg f - \sum_{i=1}^{k} N_i \lg N_i + O(N).$$

Hence, the lower bound on the average number of comparisons required to determine the mode is:

$$\Omega\left((N \log N - \sum_{i=1}^{k} N_i \log N_i) - (N \lg f - \sum_{i=1}^{k} N_i \lg N_i + O(N))\right) = \Omega\left(N \log \frac{N}{f}\right).$$

□

2.4 Summary

We proved lower bounds for the average number of comparisons required to solve the three problems. The lower bounds for the problems in the comparison model have been presented in Table 2.1.

Chapter 3

Lower Bounds in the Cache-Aware Model

In this chapter we will present lower bounds for the three problems in the cache-aware model. It is essentially a review of the work of Arge et al. [4].

The only difference between cache-aware and cache-oblivious model is that in the cache-aware model the sizes of cache and its pages (i.e., M, B) are known. Therefore, any lower bound in the cache-aware model for a specific problem is certainly also a lower bound in the cache-oblivious model for that problem.

On the other hand, any algorithm in the cache-oblivious model works in the cache-aware model by ignoring the knowledge about values of M and B. Therefore, any upper bound in the cache-oblivious model for a specific problem holds in the cache-aware model for that problem.

Arge et al. [4] proved cache-aware lower bounds for the three problems (namely, finding the mode, duplicate removal and multisorting). As mentioned, these lower bounds also hold in the cache-oblivious model. In this chapter, we will outline the proofs of the lower bounds. They also give algorithms that match the lower bounds. As our algorithms in Chapter 4 are simpler than theirs and at the same time, match the lower bounds, we will not mention their upper bounds.

The lower bounds can be duly translated from the lower bounds in the comparison

model. The notion of *I/O-trees* is defined as an adaptation of decision trees in the comparison model; we present a "main theorem" that relates the size of an I/O-tree and that of a decision tree. Since we have already proved lower bounds on the sizes of decision trees in chapter 2, by applying the main theorem, we can prove lower bounds on the sizes of I/O-trees.

The rest of this chapter is organized as follows: In Section 3.1, we will briefly describe the I/O-model in which the lower bounds are proved. In Section 3.2, the main theorem is presented. Then, each problem is studied in an individual section and specific lower bounds are presented. Finally, some extensions to the I/O-model is explained in the last section.

3.1 The I/O Model

First we need to specify the I/O model to prove lower bounds. We will assume the input elements $x_1, \ldots, x_N$ are atomic, and we also assume that the only operations allowed on the elements are comparisons. At any time two elements x_i, x_j are compared the two elements must be present in the cache.

Both the cache and the memory are divided into blocks of size B. An element can only be accessed if it is present in the cache. Each I/O operation swaps at most B contiguous elements that constitute a block between the cache and the main memory. The elements are removed from the main memory and are brought into the cache or they are flushed back from the cache to the main memory. Full associativity in the cache is assumed: Each block of the cache can be swapped with any block in the main memory. Furthermore, the algorithm can choose which block should be swapped with which (i.e., replacement policy is decided by the algorithm.) However the simple "least recently used" heuristic is known to achieve performance within a constant factor of the optimal (off-line) approach with half the number of pages of cache [11].

Note that our set of assumptions means that only one copy of an element can

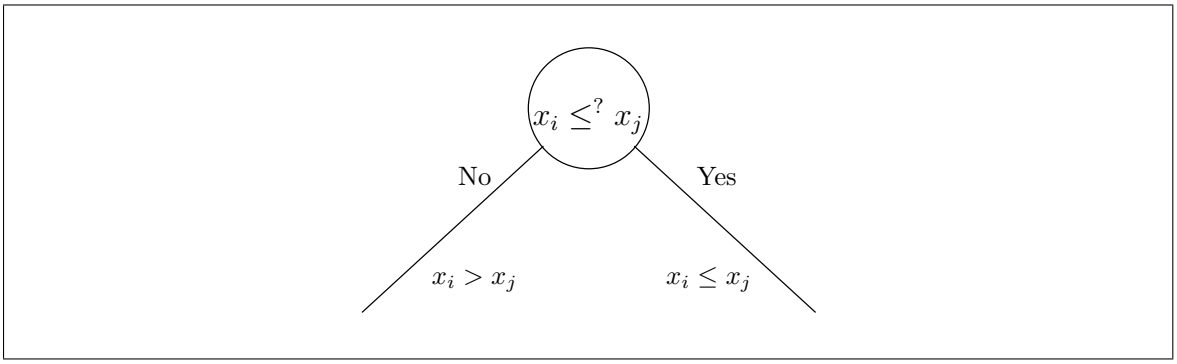

Figure 3.1: A comparison node and the labels of its edges

exist at any time. At the end of this chapter, we will show that this constraint as well as some other constraints can be relaxed to make the I/O model more realistic.

3.2 The Main Theorem

In this section, we explain the notion of I/O-trees which are counterparts to decision trees in the cache-aware model. We will also show, in the main theorem, how the size of an I/O-tree relates to that of a decision tree.

An I/O-tree has two types of nodes: comparison nodes and I/O nodes. In comparison nodes, two elements (say x_i, x_j) are compared; it is checked whether $x_i \leq x_j$. Depending on whether the answer is yes or no, two cases are possible. These correspond to the two children of the node, and we label the two edges to these children $x_i \leq x_j$ or $x_i > x_j$ accordingly (e.g., see Figure 3.1).

In an I/O-node a block of the cache is swapped by a block of the main memory. In other words, at most B contiguous elements that constitute to a block in the cache and a block in the main memory are swapped. An I/O-node can have many outgoing edges depending on which two elements in the memory are compared after the I/O operation. No label is attached to the outgoing edges of I/O-nodes.

The set of labels of edges on the path from the root of the tree to a node v of the tree is all the knowledge obtained up to the time node v is being invoked.

Definition 3.2.1. For a node v of the I/O-tree, $Path(v)$ is defined as the set of edges

of the path from the root of the tree down to the node v. The *predicate* of a node v, $P(v)$, is the logical "and" of all the labels on $Path(v)$.

At any leaf l of the tree, the knowledge, the predicate $P(l)$ provides, must be sufficient to solve the problem, since no further operations will be invoked after reaching a leaf.

Definition 3.2.2. An I/O-tree is called *valid* if for any leaf l, $P(l)$ is enough to reach a conclusion and solve the problem at hand.

If one changes internal nodes of a valid I/O-tree and their operations so that the predicate of each leaf l remains the same or is changed to a new $P(l)$ that implies the old $P(l)$, the new tree will be valid as well.

We are now ready to present the main theorem:

Theorem 3.2.1 ([4]). *If there is a valid I/O-tree T that solves a particular problem, there is a comparison tree T_c that solves the same problem and has the following property:*

$$height(T_c) \leq n \lg B + height_{I/O}(T) \times T_{merge}(M - B, B)$$

where $height(T_c)$ is the height of T_c, $height_{I/O}(T)$ is the maximum number of I/O-nodes on any path from the root a leaf, and $T_{merge}(M - B, B)$ is the number of cache misses that happens during merging two sorted lists of sizes $M - B$ and B.

Proof. The idea is to construct a comparison tree T_c from the I/O-tree T by a series of transformations; in each transformation a comparison subtree is considered and replaced by an optimized one.

Definition 3.2.3. A comparison subtree of an I/O-tree is a subtree whose root and leaves are all I/O-nodes and whose internal nodes are all comparison nodes (see Figure 3.2 for an illustration.)

At any time the uppermost comparison subtree is transformed; in other words, the comparison subtree whose root has the smallest depth (breaking ties arbitrarily) is considered and is replaced by another comparison subtree.

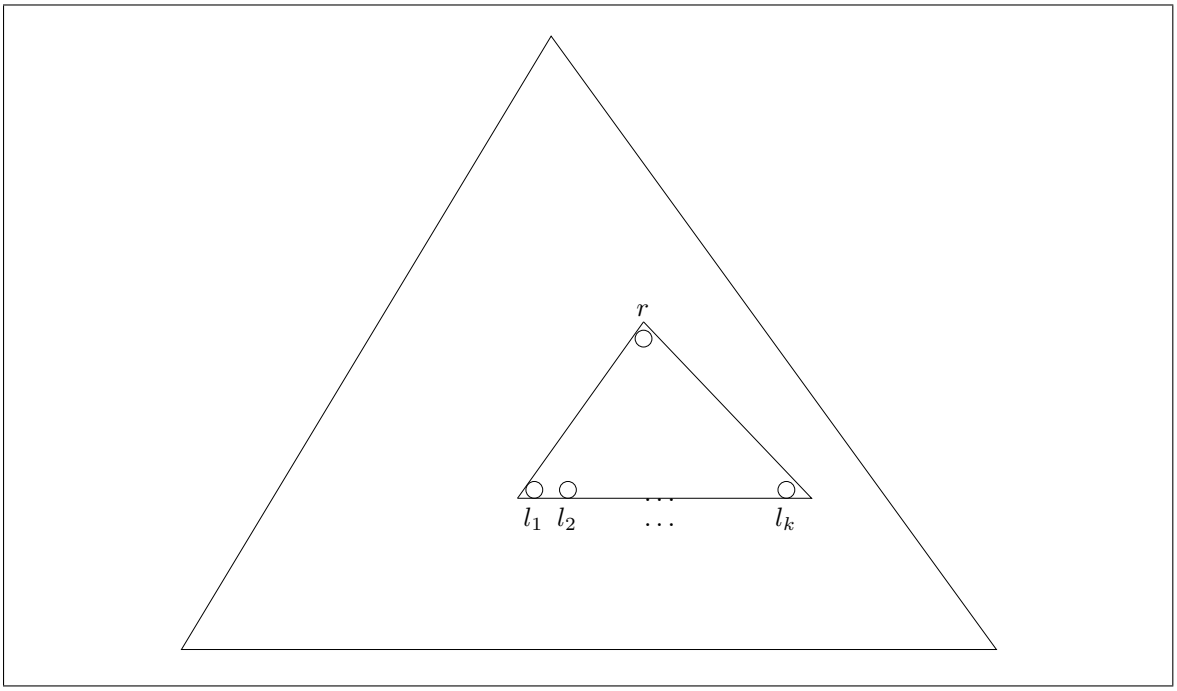

Figure 3.2: Comparison subtree of an I/O-tree: All the leaves $l_1, \ldots, l_k$ and the root r are I/O-nodes and internal nodes are of type comparison

During the transformation, we always maintain the invariant that at each I/O-node before the I/O-operations we know the total ordering of the M elements in the cache and the present elements in the cache are sorted based on this ordering. The invariant obviously holds at the beginning; at the root of the I/O-tree, there is no element in the memory and thus the invariant holds.

We further assume that the B elements that are brought into the cache by any I/O-operations are already in sorted order. The first time a block is read into the cache, this assumption certainly does not hold; so we have to determine the ordering of a block when it is first brought in and sort it. This takes $B \lg B$ comparisons per a block, as there are N/B blocks, the total number of comparisons is $N \lg B$. When we are writing a block back into the main memory, we output it in sorted order, hence the next time it is brought in the cache, it is already sorted. Thus, it incurs no more comparisons.

Consider the uppermost comparison subtree T with root r and leaves $l_1, \ldots, l_k$.

At the root a block is swapped into the cache from the main memory. All internal nodes of T are comparison nodes. As we can only access the elements that are present in the cache, the comparisons at the internal nodes are all among those elements that are present in the cache.

We replace this subtree by a subtree that finds the total ordering among all the elements present in the cache. It is obvious that the set of predicates $P(l_i)$ at each leaf l_i in the new subtree includes the predicate $P(l_i)$ in the old subtree, since the total ordering is the most one can know about the elements (We know the relation between any two elements in the total ordering.)

The number of comparisons we incur to find the total ordering after reading a block into the cache can be computed as follows: According to our assumptions we know the ordering of the elements within the block, we have also maintained the invariant that we know the ordering among the $M - B$ elements that are present in the cache. Therefore, we just have to merge the B elements in the block with the $M - B$ elements present in the cache which is $T_{merge}(M - B, B)$.

We consider comparison subtrees one by one from the root of the I/O-tree down to the leaves, and replace each one with a subtree of height $T_{merge}(M-B, B)$. Therefore, the height of the resulting comparison tree T_c will have the following property:

$$height(T_c) \leq n \lg B + height_{I/O}(T) \times T_{merge}(M - B, B).$$

□

$T_{merge}(M-B, B)$ is easy to compute; we are merging two sorted lists of size $M-B$ and B. It can be done by looking up each of the B elements in the list of size $M - B$ by a binary search. The Hwang-Lin algorithm [7] can be used to merge two lists of sizes p and q $(p \leq q)$ using

$$p \left\lceil \lg \frac{q}{p} \right\rceil + p + \left\lfloor \frac{p}{2^{\lfloor \lg \frac{q}{p} \rfloor}} \right\rfloor - 1$$

comparisons which is less than $p \lg \frac{q}{p} + 3p$. Therefore,

$$T_{merge}(M - B, B) \leq B \lg \left(\frac{M - B}{B} \right) + 3B.$$

Corollary 3.2.2 ([4]). *If there is a valid I/O-tree T, there is a comparison tree T_c for which the following inequality holds:*

$$height(T_c) \leq N \lg B + height_{I/O}(T) \,.\, (B \lg(\frac{M-B}{B}) + 3B)$$

where $height(T_c)$ is the height of T_c and $height_{I/O}(T)$ is the maximum number of I/O-nodes on a path from the root to the leaves in T.

The inequality in Corollary 3.2.2 directly gives lower bounds for the number of I/O-operations in any valid I/O-tree T; the inequality combined with the lower bounds for the height of any valid comparison tree T_c in Chapter 2 imply a lower bound on $height_{I/O}(T)$ which is the number of necessary I/O-operations in the worst case.

3.3 Lower Bound on Determining the Mode

In Section 2.3, we showed that to find the mode in a multiset of size N where the frequency of the mode is f, any algorithm takes $N \lg(N/f)$ comparisons to within lower order terms. Here we combine this lower bound with the Corollary 3.2.2 in Section 3.2 to obtain a lower bound for the number of I/O operations involved.

Theorem 3.3.1 ([4]). *The number of I/O-operations required to determine the mode with frequency f in a multiset of size N is $\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB},\ \frac{N}{B}\right\}\right)$.*

Proof. According to Corollary 3.2.2, the following inequality holds:

$$height(T_c) \leq N \lg B + height_{I/O}(T) \,.\, (B \lg(\frac{M-B}{B}) + 3B).$$

In Section 2.3 we showed that:

$$height(T_c) \in \Omega(N \log(N/f)).$$

By combining these two, we obtain:

$$\begin{aligned} height_{I/O}(T) &\in \Omega\left(\frac{N\log(N/f) - N\log B}{B\log\frac{M-B}{B} + 3B}\right) \Rightarrow \\ height_{I/O}(T) &\in \Omega\left(\frac{N\log\frac{N}{fB}}{B\log\frac{M}{B}}\right) \Rightarrow \\ height_{I/O}(T) &\in \Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right). \end{aligned}$$

In case $\frac{N}{f} < M$, the above bound is negative. In this case, since we should read all the elements in there is an obvious lower bound of $O\left(\frac{N}{B}\right)$. □

3.4 Lower Bound on Sorting

In Section 2.1, we presented a lower bound on the number of comparisons required to sort a multiset. Here we combine the lower bound with Corollary 3.2.2 to obtain a lower bound for the number of I/O-operations required.

Theorem 3.4.1 ([4]). *To sort a multiset of size N with k distinct elements of multiplicities $N_1, N_2, \ldots, N_k$, the number of I/O-operations required is:*

$$\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \frac{N}{B}\right\}\right).$$

Proof. As it was shown in Section 2.1, the lower bound for the number of comparisons to sort a multiset is:

$$height(T_c) \in \Omega\left(N\log N - \sum_{i=1}^{k}N_i\log N_i\right).$$

According to Corollary 3.2.2:

$$height(T_c) \leq N\lg B + height_{I/O}(T)\ \times\ (B\lg(\frac{M-B}{B}) + 3B).$$

By combining these two together, we obtain:

$$\begin{aligned} height_{I/O}(T) &\in \Omega\left(\frac{N\log N - \sum_{i=1}^{k} N_i \log N_i - N\log B}{B\log\frac{M-B}{B} + 3B}\right) \Rightarrow \\ height_{I/O}(T) &\in \Omega\left(\frac{N\log\frac{N}{B} - \sum_{i=1}^{k} N_i \log N_i}{B\log\frac{M}{B}}\right) \Rightarrow \\ height_{I/O}(T) &\in \Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i\right). \end{aligned}$$

Again when $\frac{N}{f} < M$, the above bound is no use. In this case, since we should read all the elements in there is an obvious lower bound of $O\left(\frac{N}{B}\right)$. □

3.5 Lower Bound on Duplicate Elimination

In Section 2.2, it was proved that after duplicate removal, the total ordering among all elements must be known; in other words removing the duplicates of a multiset requires the same number of comparisons as sorting:

$$height(T_c) \in \Omega\left(N\log N - \sum_{i=1}^{k} N_i \log N_i\right).$$

One can follow the same steps as in Section 3.4 to obtain the same lower bound as in Theorem 3.4.1:

Theorem 3.5.1 ([4]). *To remove duplicates from a multiset of size N with k distinct elements of multiplicities $N_1, \ldots, N_k$, the number of I/O-operations required is:*

$$\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i,\ \frac{N}{B}\right\}\right).$$

3.6 Extensions to the I/O-Model

The I/O-model defined in Section 3.1 is limited; consequently, the lower bounds hold for algorithms that fit in the limited model. In this section, we will show some possible

extensions to the model. These extensions make the model more reasonable. We will also show that the lower bounds still hold in the new model.

The I/O-model in Section 3.1 is limited in three aspects: First, only one copy of any element can be present in the system at any time; we can relax this constraint so that elements can be copied. In this new model, I/O-operations do not have to be block swaps between the cache and the main memory; instead we have two kinds of I/O-operations: I-operations and O-operations. In I-operations, a block of the main memory is read into the cache, and in O-operations, a block of the cache is written back into the main memory (without it being removed from the cache). The new I/O-operations certainly allow copying of the elements.

It is not hard to see that the results of Section 3.2 still hold and consequently lower bounds remain valid. I/O-trees in the new model, have two types of I/O-nodes: I-nodes and O-nodes which correspond to I-operations and O-operations respectively. The steps in Section 3.2 can be followed in the same way to yield Corollary 3.2.2: A comparison subtree can be replaced by an optimized one in the same way. Once we obtain Corollary 3.2.2, the lower bounds can be obtained directly.

Secondly, in the model of Section 3.1, we assumed that no helper variable is used, and all branches are made by comparison between the elements. We can, however, permit the use of helper variables as long as their values are implied by the comparisons between the elements up to that point, and we can permit branching based on values of these variables. Since these variables cannot save any comparisons, our lower bounds remain intact.

Finally, we allowed only binary comparisons (i.e., $\leq$ and $>$). However, we can also permit ternary comparisons (i.e., $<$ or $=$ or $>$). Ternary comparisons do not affect the I/O-height of the I/O-tree and their effect on the comparison height is only a constant factor; as we are only interested in asymptotic lower bounds in this chapter, constant factors do not matter and the lower bounds remain valid.

3.7 Summary

In this chapter, we reported proofs of lower bounds for the cache complexity of our three problems. The lower bounds in the cache-aware model have been presented in Table 3.1.

Table 3.1: The lower bounds on the cache complexities in the cache-aware model

	Lower bound
Determining the Mode	$\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB},\ \frac{N}{B}\right\}\right)$
Multisorting	$\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}-\sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \frac{N}{B}\right\}\right)$
Duplicate Elimination	$\Omega\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}-\sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \frac{N}{B}\right\}\right)$

Chapter 4

Cache-Oblivious Upper Bounds

In this chapter, we will present cache-oblivious algorithms for each of the three problems. In Chapter 3, lower bounds were proved for the problems in the cache-aware model. Obviously, these lower bounds duly hold in the cache-oblivious model, since the cache-oblivious model is exactly the same as the cache-aware one except that there is no knowledge of values of parameters M, B.

Arge et al. [4] presented cache-aware algorithms that match the lower bounds proven in Chapter 3. We present cache-oblivious algorithms that are close to match the lower bounds. The cache-aware versions of our algorithms, however, match the lower bounds and are much simpler than the ones presented by Arge et al.

The rest of this chapter is organized as follows: We first explain the cache-oblivious model in detail in Section 4.1. We take three different approaches to give cache-oblivious algorithms for the three problems. Due to the similarity between our first approach and the selection sort, we call it *the selection approach*. It is explained in Section 4.2. Our second approach is called *the distribution approach* due to its similarity with distribution sort. We will describe this approach in Section 4.3. Finally, our last approach is randomized and will be discussed in Section 4.4.

4.1 The Cache-Oblivious Model

In this section the cache-oblivious model is described in detail. The memory hierarchy has two levels: On the first level, there is a cache of size M. On the second level, there is an infinite memory. The cache and the memory are divided into blocks of B words. The processor can only reference a word that is present in the cache. If the word has already been brought in the cache, a *cache hit* occurs. Otherwise, the block that contains the requested word must be fetched from memory into the cache, in this case a *cache miss* (or a cache fault) happens.

We suppose the block replacement policy is the off-line optimal one. At the time of a cache miss, a block is *automatically* evicted and replaced by the newly requested block. The choice of the block for eviction is automatic and is off-line optimal in the sense that it causes the least number of cache misses in the whole run of the algorithm. The assumption of an automatic off-line optimal is not unrealistic as a *least recently used (LRU)* policy is only a constant factor away from it if the cache size is doubled. We also assume that the cache is fully-associative; each block of the cache can be replaced by any block of the main memory.

The size of the cache (M) and the size of a block (B) are unknown to a cache-oblivious algorithm; in other words, the algorithm is *oblivious* of these values.

4.2 The Selection Approach

In this section we present our first approach to give upper bounds for solving the three problems cache-obliviously. Due to the resemblance of this approach to selection sort, this approach is called the selection approach.

We start by proving a key theorem in Section 4.2.1 which we will use as a building block in solving the three problems in the three following sections. The theorem deals with finding frequent elements in a multiset efficiently.

Although the upper bounds do not exactly match the lower bounds, they come very close. In Section 4.2.5, optimality of the algorithms are explained: Situations are

described where the algorithms are proven optimal and also situations are mentioned where the algorithms do not match the lower bounds.

4.2.1 Finding Frequent Elements

The main theorem for the selection approach is presented in this section. In the theorem, we efficiently find a set of "frequent" elements in a given multiset. Let us first define what exactly we mean by "frequent".

Definition 4.2.1. We call an element *C-frequent* if and only if it occurs more than $\frac{N}{C}$ times in a multiset of size N.

The main theorem is about "selecting" C-frequent elements quickly and is used as a building block in the algorithms of the selection approach.

Theorem 4.2.1. *In a multiset of size N, C-frequent elements (those with multiplicities greater than $\frac{N}{C}$) and their actual multiplicities can be determined with cache complexity $O\left(\frac{N}{B}\max\{1, \log_{\frac{M}{B}} \frac{C}{B}\}\right)$.*

Proof. The algorithm works in two phases. In the first phase, we try to find a set of at most C candidates that contains all the C-frequent elements. There may also be some other arbitrarily infrequent elements in our list of candidates. Note that, by definition, the number of C-frequent elements cannot exceed C. In the second phase, we check the C candidates to determine their exact frequencies. The algorithm is illustrated in Figure 4.1.

Phase 1 (Finding candidates). The key idea in this phase is essentially what Misra [8] used. We find and remove a set of t ($t \geq C$) distinct elements from the multiset. The resulting multiset has the property that those elements that were C-frequent in the original multiset are still C-frequent in the reduced multiset. This is because after deleting the t distinct elements, the frequency of each element can be reduced by at most one. Therefore those elements that had multiplicities greater

Procedure FREQUENT ELEMENT FINDER(C)

Variables:

Candidates: Holds as many as C distinct elements along with a counter for each element (Initially empty.)

G: Holds exactly C contiguous elements from the multiset (duplicates are possible.)

T: Holds as many as $2C$ distinct elements along with a counter for each element.

begin

// Phase 1: Finding Candidates.

while *there are more elements in the multiset* **do**

1 Read the next C elements from the multiset into G.

2 Sort G and Candidates.

3 Merge G and Candidates into T:

4 Reduce T by throwing out all but the $C-1$ most frequent elements. This is done as follows:

4a Sort T according to the value of the counters.

4b Throw out all but the elements with $C-1$ largest counters.

4c Decrease the counters of the remaining elements by the counter value of the most frequent element discarded.

5 Candidates $\leftarrow$ T.

// Phase 2: Confirmation.

while *there are more elements in the multiset* **do**

6 Read the next C elements from the multiset into G.

7 Sort G and Candidates.

8 Accumulatively increase the counters of Candidates by a merge of G and Candidates.

9 Retain elements in Candidates with counters larger than $\frac{N}{C}$.

end

Figure 4.1: Frequent Finding Algorithm

than N/C, now have now multiplicities greater than $N/C - 1 = \frac{N-C}{C} \geq \frac{N-t}{C}$. The claim follows by considering the fact that the size of the new multiset is $N - t$.

Thus, we can keep removing sets of at least C distinct elements from the multiset one at a time until the multiset has no longer more than C distinct elements; the C-frequent elements in the original multiset must be also present in the final multiset.

The algorithm illustrated in Figure 4.1 works as follows. We scan the multiset from the beginning to the end in groups of C elements (there are N/C such groups). We maintain an array of *"candidates"* of size C which is initially empty and eventually will hold as many as C distinct values; each element in this array also has a counter associated with it which shows the number of occurrences of the element so far. As soon as the number of elements in this array goes over C, it means there are C or more distinct elements in the array and we can reduce the size of the multiset by removing C distinct elements.

We read C elements into G (line 1). Then we sort the elements into G and also sort the elements in the $Candidates$ array (line 2.) This can be done by using any method of cache-oblivious sorting for sets which causes $O\left(\frac{C}{B} \max\left\{1, \log_{\frac{M}{B}} \frac{C}{B}\right\}\right)$ cache misses (e.g. see Frigo et al. [6]).

Once arrays G and $Candidates$ are sorted, we merge them into another array T (line 3.) The merging is done as in the merge sort. All elements in T are required to be distinct and hence there is a counter associated with each element to represent the number of its duplicates. The merging takes a pass from each list and may cause $O(\frac{C}{B})$ cache misses, which is negligible compared to that of sorting the C elements.

If $|T| \geq C$, we downsize T. Logically, we would like to repeatedly remove one copy of each distinct element until fewer than C distinct elements remain. The counters of these remaining elements are then adjusted to these new, lower, values. This logical procedure, however, must be performed in a cache-efficient manner, so we actually proceed as follows. We sort T according to the value of the *counters* (line 4a). We then find the C^{th} largest counter m_c. All elements with a counter value at most m_c are thrown away and the counter of the rest of the elements is decreased by m_c (lines

4b and 4c.) One can easily see that this is equivalent to repeatedly throwing away groups of at least C distinct elements from T one at a time.

Array *Candidates* is emptied and is set to T (line 5.) The next group of C elements are read in from the multiset and the process continues so on.

After the last group of C elements is considered, the *Candidates* array contains all possible C-frequent elements in the multiset; however, as mentioned, it may also contain some more arbitrary elements.

The cache complexity of this phase is obviously:

$$\frac{N}{C} \times O\left(\frac{C}{B} \max\left\{1, \log_{\frac{M}{B}} \frac{C}{B}\right\}\right) \quad = \quad O\left(\frac{N}{B} \max\left\{1, \log_{\frac{M}{B}} \frac{C}{B}\right\}\right).$$

Phase 2 (Confirmation). This phase is similar to the first phase, except that array *Candidates* remains intact throughout this phase. Given the candidates array, we first zero out all the counters of the elements. We then consider N/C groups of C elements from the multiset one at a time. We read next group of C elements into G (line 6.) We then sort the elements in G and *Candidates* using any cache-oblivious sorting algorithm for sets (line 7.) Then we count how many times each of the candidates occur in the group by a scan of G and *Candidates* as in the merge sort (line 8.) We accumulate these counts so that after considering the final group of C elements, we know exactly how many times each of the candidates has occurred in the whole multiset. We finally keep all elements whose counters are more than N/C and throw away the rest (line 9.) Cache complexity of this phase can be worked out similarly as the first phase to be:

$$O\left(\frac{N}{B} \max\left\{1, \log_{\frac{M}{B}} \frac{C}{B}\right\}\right).$$

□

The number of comparisons this algorithm makes or its time complexity can be worked out in a manner similar to its cache complexity. Sorting each group of $O(C)$ elements requires $O(C \lg C)$ comparisons. As we sort the entire multiset of size N in groups of C elements, and we sort array T which can hold as many as $2C$ elements,

it requires $N/C \times O(C \lg C) = O(N \lg C)$ comparisons. The number of comparisons during the merging is the same as the number of elements in the multiset which is negligible compared to the number of comparisons during the sorting. Thus, the total number of comparisons is $N \lg C$.

The key point in using the frequent finder algorithm on specific data is the wise choice of C. Indeed the method itself is not worded to take advantage of the data at hand. We will show in the following sections how this wise choice of C can be made.

4.2.2 Determining the Mode

In this section, we present an algorithm that determines the mode of a multiset. We apply Theorem 4.2.1 of the previous section repeatedly for a series of values for C. The upper bound will turn out to be the maximum of two terms: $\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{fB}$ and $\frac{N}{B} \log\log \frac{N}{f}$. The first term is the lower bound we would like to match. However, the second term is the extra term that does not let the algorithm always match the lower bound. We will show that the excess from the lower bound is small and in fact less than $\frac{N}{B} \log\log M$ [1].

Theorem 4.2.2. *Given a multiset of size N, with f the frequency of the mode unknown, The mode and f, can be found with cache complexity*

$$O\left(\max\left\{\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{fB}, \frac{N}{B} \log\log \frac{N}{f}\right\}\right).$$

Proof. We repeatedly apply Theorem 4.2.1 for a series of increasing values of C to determine whether there is any C-frequent element in the multiset. The first time some C-frequent elements are found, we halt the algorithm and declare the most frequent among them as the mode.

The algorithm in Theorem 4.2.1 is executed in rounds as C goes doubly exponentially for the following values of C in order: $C = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^i}, \ldots, 2^{2^{\lceil \lg\lg N+1 \rceil}}$. At the end of each round, we either end up empty-handed or we find some C-frequent

[1] For practical caches $\log\log M$ is small. For example, for a cache of size $M = 1000$ terabytes, $\log\log M$ is less than 6.

elements. In the former case, the algorithm continues with the next value of C. In the latter case, we declare the most frequent of the C-frequent elements to be the mode, and the algorithm halts. Note that the algorithm of Theorem 4.2.1 also produces the actual multiplicities of the C-frequent elements, thus finding the most frequent element among the C-frequent ones requires only a pass of the at most C elements to select the element with the maximum multiplicity.

The correctness of the algorithm is straightforward. First note that the algorithm will eventually halt and report an element as the mode; for a value of C as large as N, the algorithm will halt, since all elements are $N+1$-frequent. Secondly, the element that the algorithm reports is truly the mode. For a value of C, the algorithm finds all the C-frequent elements; the mode must be among these elements as it is the most frequent element. Since we report the most frequent element among the C-frequent elements, we output the mode.

We now analyze the cache complexity of the algorithm. We denote by f the frequency of the mode, and set $k = \left\lfloor \lg\lg \frac{N}{f} \right\rfloor$; we have the following inequality:

$$\frac{N}{2^{2^{k+1}}} < f \leq \frac{N}{2^{2^k}}.$$

Then the algorithm runs for values $2^{2^1}, 2^{2^2}, \ldots, 2^{2^{k+1}}$ of C. By summing up the cache complexity of the algorithm in Theorem 4.2.1 for the mentioned values of C, we obtain:

$$\begin{aligned}
\sum_{j=1}^{k+1} O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{2^{2^j}}{B}\right\}\right) &= O\left(\frac{N}{B}\max\left\{(k+1),\ \log_{\frac{M}{B}}\frac{2^{2^{k+2}}}{B}\right\}\right) \\
&= O\left(\max\left\{\frac{N}{B}k,\ \frac{N}{B}\log_{\frac{M}{B}}\frac{2^{2^k}}{B}\right\}\right) \\
&= O\left(\max\left\{\frac{N}{B}\log\log\frac{N}{f},\ \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right\}\right).
\end{aligned}$$

Hence the cache complexity of the algorithms consists of two terms: $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}$ and $\frac{N}{B}\log\log\frac{N}{f}$. The former term is the lower bound proven. The latter term produces a gap between the upper and the lower bound. It essentially arises from the first runs

Table 4.1: Optimality of the upper bound for determining the mode in Theorem 4.2.2

	Upper Bound	**Lower Bound**
$N/f > M$	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log M\right\}\right)$	$\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right)$
$N/f \leq M$	$O\left(\frac{N}{B}\log\log\frac{N}{f}\right)$	$\Omega\left(\frac{N}{B}\right)$

of the frequent finder algorithm for small values of C where $C < M$. In each such run of the algorithm we at least take a pass of the whole multiset which cost $\frac{N}{B}$ cache misses each. The issue of the gap between the lower and upper bound is discussed in detail later. □

As mentioned earlier, the upper bound given in Theorem 4.2.2 does not match the lower bound. To see how far it can be from the lower bound, two cases are considered (these two cases are summarized in Table 4.1):

- $N/f > M$: The upper bound in Theorem 4.2.2 is:

$$U = O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right).$$

The first term is the lower bound; so we focus on the extra term $\frac{N}{B}\log\log\frac{N}{f}$:

$$\begin{aligned}\frac{N}{B}\log\log\frac{N}{f} &= \frac{N}{B}\log\left(\log_{\frac{M}{B}}\frac{N}{f}\times\log\frac{M}{B}\right)\\ &= \frac{N}{B}\left(\log\log_{\frac{M}{B}}\frac{N}{f}+\log\log\frac{M}{B}\right)\\ &= \frac{N}{B}\log\log_{\frac{M}{B}}\frac{N}{f}+\frac{N}{B}\log\log\frac{M}{B}\end{aligned}$$

Thus, we can rewrite the upper bound U as:

$$\begin{aligned} U &= O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right) \\ &= O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \left(\frac{N}{B}\log\log_{\frac{M}{B}}\frac{N}{f} + \frac{N}{B}\log\log\frac{M}{B}\right)\right\}\right) \\ &= O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log M\right\}\right). \end{aligned}$$

- $N/f \leq M$: In this case the upper bound can be rewritten as:

$$\begin{aligned} U &= O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right) \\ &= O\left(\frac{N}{B}\log\log\frac{N}{f}\right). \end{aligned}$$

  However the lower bound in this case is $\Omega(N/B)$, because the algorithm has to look at each element at least once; otherwise it is easy to come up with an adversary algorithm that makes the algorithm make a wrong decision. Suppose there are exactly N/f different elements that each occur exactly f times. The algorithm cannot make any decision whatsoever regarding to which element is the mode until the last element is read, since by changing the value of the last element one can get a different element as the mode.

The difference between the lower bounds and upper bounds in the two cases can be clearly seen in Table 4.1. The cache complexity of the algorithm has an excess from the lower bound that is $O\left(\frac{N}{B}\log\log M\right)$ and when $N/f < M$, this excess reduces to $O\left(\frac{N}{B}\log\log\frac{N}{f}\right)$. For practical values of cache size M, $\lg\lg M$ can be considered as a small constant.

The number of comparisons this algorithm requires is easy to compute: At each round, for each value of C, the number of comparisons is $O\left(N\log C\right)$ as it was shown

in Section 4.2.1. Therefore, the total number of comparisons the algorithm for determining the mode is:

$$\begin{aligned}\sum_{i=1}^{\lceil \lg \lg \frac{N}{f} \rceil} O\left(N \log 2^{2^i}\right) &= O\left(N 2^{\lceil \log \log \frac{N}{f} \rceil}\right) \\ &= O\left(N \log \frac{N}{f}\right).\end{aligned}$$

According to the lower bound in Section 2.3, the algorithm performs the asymptotically optimal number of comparisons to determine the mode.

4.2.3 Duplicate Elimination

We are given a multiset of size N which consists of k pairwise distinct elements $i_1, \ldots, i_k$ whose multiplicities are $N_1, \ldots, N_k$ respectively. Without loss of generality, we can assume $N_1 \geq N_2 \geq \cdots \geq N_k$. The goal in this section is to reduce the multiset to the set $\{i_1, \ldots, i_k\}$.

Again, the cache complexity of our algorithm for duplicate removal does not quite match the proven lower bound in Section 3.5. As in Section 4.2.3, the excess from the lower bound is small and does not exceed from an additive term of $\frac{N}{B} \lg \lg M$. As $\lg \lg M$.

Theorem 4.2.3. *Eliminating the duplicates of a multiset of size N whose multiplicities are $N_1 \geq N_2 \geq \ldots \geq N_k$ can be done with cache complexity:*

$$O\left(\max\left\{\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{k} \frac{N_i}{B} \log_{\frac{M}{B}} N_i,\ \sum_{i=1}^{k} \frac{N_i}{B} \log \log \frac{N}{N_i}\right\}\right).$$

Proof. The algorithm is similar to that in Theorem 4.2.2 where we find the mode in a multiset.

We, again, repeatedly apply Theorem 4.2.1 in rounds for each value of $C = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^{\lceil \lg \lg N + 1 \rceil}}$ to find C-frequent elements. At the end of each round, we

output the discovered C-frequent elements as they belong to the final set and remove all occurrences of these elements from the multiset.

The deletion of the discovered elements from the multiset is done as in Theorem 4.2.1; the multiset is scanned in groups of C elements. Each group of C elements is already sorted; we compare the sorted list with the frequent elements (which are also already sorted in the "candidates array"). The comparison is done as in the merge sort and we delete the intersection of these two from the multiset. The algorithm continues in next rounds with the reduced multiset. The algorithm halts when the multiset is reduced to an empty set.

The correctness of this algorithm is clear; all element $i_1, \ldots, i_k$ are eventually discovered and reported in the output. Obviously the algorithm does not output repeated elements as we delete copies of discovered elements after we output them.

It remains to analyze the cache complexity. The cache complexity is a summation of the cache complexity of the algorithm in Theorem 4.2.1 that finds the frequent elements. In that algorithm to find C-frequent elements we had $O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{C}{B}\right\}\right)$ cache misses. To get rid of the max term in our computations, we sum the two terms separately and then take the maximum of the two sums.

We clearly discover the elements in their order of frequencies; element i_1 with frequency N_1 is discovered first, then i_2 with frequency N_2 and so on.

Definition 4.2.2. Define $C(r)$ as the contribution of the term $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{C}{B}\right)$ in the number of cache misses during the period of time after it has just discovered element i_{r-1} until it discovers i_r. Similarly define $D(r)$ as the contribution of $O\left(\frac{N}{B}\right)$ term in the number of cache misses during the same period of time.

We first focus on $C(r)$ for different values of r. In Section 4.2.2 to determine the mode, we implicitly computed $C(1)$ to be $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{N_1 B})$. For all values of $r > 1$, since all elements $i_1, \ldots, i_{r-1}$ are already removed from the multiset, the size of the multiset the algorithm deals with is $N' = N - (N_1 + \ldots + N_{r-1})$. Set $p = \left\lfloor \lg\lg\frac{N}{N_{r-1}} \right\rfloor$

and $q = \left\lfloor \lg \lg \frac{N}{N_r} \right\rfloor$, and we have:

$$\frac{N}{2^{2^{p+1}}} < N_{r-1} \leq \frac{N}{2^{2^p}}$$
$$\frac{N}{2^{2^{q+1}}} < N_r \leq \frac{N}{2^{2^q}}.$$

Then the value of $C(r)$ for all values of $r > 1$ can be computed as follows:

$$\begin{aligned}
C(r) &= \sum_{i=p+2}^{q+1} O\left(\frac{N'}{B} \log_{\frac{M}{B}} \frac{2^{2^i}}{B}\right) \\
&= O\left(\frac{N'}{B} \log_{\frac{M}{B}} 2^{2^q - 2^p}\right) \\
&= O\left(\frac{N'}{B} \log_{\frac{M}{B}} \frac{N_{r-1}}{N_r}\right) \\
&= O\left(\frac{N - \sum_{i=1}^{r-1} N_i}{B} \log_{\frac{M}{B}} \frac{N_{r-1}}{N_r}\right) \\
&= O\left(\frac{N - \sum_{i=1}^{r-1} N_i}{B} (\log_{\frac{M}{B}} N_{r-1} - \log_{\frac{M}{B}} N_r)\right).
\end{aligned}$$

The summation of $C(r)$ would yield the first part of the cache complexity of the algorithm:

$$\begin{aligned}
\sum_{r=1}^{k} C(r) &= O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{N_1 B}\right) + \sum_{r=2}^{k} O\left(\frac{N - \sum_{i=1}^{r-1} N_i}{B} (\log_{\frac{M}{B}} N_{r-1} - \log_{\frac{M}{B}} N_r)\right) \\
&= O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{k} \frac{N_i}{B} \log_{\frac{M}{B}} N_i\right). \qquad (4.2.1)
\end{aligned}$$

We now focus on $D(r)$ for different values of r; the computation is similar to that of $C(r)$. We have implicitly computed $D(1)$ in Section 4.2.2 to be $\frac{N}{B} \log \log \frac{N}{N_1}$, where we found the mode. For $r > 1$, the algorithm deals only with $N' = N - (N_1 + \ldots, N_{r-1})$ elements. The number of passes during the time after i_{r-1} is discovered and before i_r

is discovered is $O\left(\log\log\frac{N}{N_r} - \log\log\frac{N}{N_{r-1}}\right)$. Thus,

$$\begin{aligned} D(r) &= O\left(\frac{N'}{B}(\log\log\frac{N}{N_r} - \log\log\frac{N}{N_{r-1}})\right) \\ &= O\left(\frac{(N - \sum_{i=1}^{r-1} N_i)}{B}(\log\log\frac{N}{N_r} - \log\log\frac{N}{N_{r-1}})\right). \end{aligned}$$

The sum of $D(r)$ for all values of r will yield the second part of the cache complexity:

$$\begin{aligned} \sum_{r=1}^{k} D(r) &= O\left(\frac{N}{B}\log\log\frac{N}{N_1}\right) + \sum_{r=2}^{k} O\left(\frac{(N - \sum_{i=1}^{r-1} N_i)}{B}(\log\log\frac{N}{N_r} - \log\log\frac{N}{N_{r-1}})\right) \\ &= O\left(\sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right). \end{aligned} \tag{4.2.2}$$

By taking the maximum of the two formulas in equations 4.2.1 and 4.2.2, the theorem is obtained. □

As in determining the mode, the cache complexity of the algorithm for duplicate elimination does not match the lower bound. The upper bound can be rewritten as:

$$\begin{aligned} U &= O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right) \\ &= O\left(\max\left\{\sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}\frac{N}{N_i B},\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right). \end{aligned}$$

The first term is the lower bound we should match. The same argument as in Theorem 4.2.3 will show that the excess from the lower bound is:

$$\sum_{i=1}^{k}\frac{N_i}{B}\log\log\min\left\{M, \frac{N}{N_i}\right\}.$$

The following excess cannot be more than $\frac{N}{B}\log\log M$, and since $\log\log M$ is small, we are again not far from the optimal.

Analysis of the time complexity or the number of comparisons of the algorithm is quite similar to the computation of $C(r)$ in the proof of Theorem 4.2.3: Define $comp(r)$ as the number of comparisons the algorithm makes after discovering i_{r-1} until it discovers i_r. Also set $p = \left\lfloor \lg\lg \frac{N}{N_{r-1}} \right\rfloor$, $q = \left\lfloor \lg\lg \frac{N}{N_r} \right\rfloor$, and $N' = N - \sum_{i=1}^{r-1} N_i$. We showed in Section 4.2.2 that $Comp(1) = O\left(N \log \frac{N}{N_1}\right)$. For $r > 1$ we have:

$$\begin{aligned} Comp(r) &= \sum_{i=p+2}^{q+1} O\left(N' \log 2^{2^i}\right) \\ &= O\left(N' \log 2^{2^q - 2^p}\right) \\ &= O\left(N' \log \frac{N_{r-1}}{N_r}\right) \\ &= O\left((N - \sum_{i=1}^{r-1} N_i) \log \frac{N_{r-1}}{N_r}\right) \\ &= O\left((N - \sum_{i=1}^{r-1} N_i)(\log N_{r-1} - \log N_r)\right). \end{aligned}$$

By summing up the values of $Comp(r)$, we obtain:

$$\begin{aligned} \sum_{r=1}^{k} Comp(r) &= O\left(N \log \frac{N}{N_1}\right) + \sum_{r=2}^{k} O\left((N - \sum_{i=1}^{r-1} N_i)(\log N_{r-1} - \log N_r)\right) \\ &= O\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right). \end{aligned}$$

Thus, according to the lower bound in Section 2.2 the algorithm performs the optimal number of comparisons asymptotically.

4.2.4 Multi-sorting

We are given a multiset of size N consisting of k distinct elements $i_1, i_2, \ldots, i_k$ with multiplicities $N_1, N_2, \ldots, N_k$ respectively. We are to sort the multiset and output the N elements in the sorted order.

Sorting can be studied in two models: In the first model, elements are such that one of the two equal elements can be removed and then later on, it can be copied

from the other one to be regenerated again. In this model, we can keep only one copy of an element and throw away its duplicates at the time of first encounter.

However, in the second model, elements cannot be deleted and regenerated by copying them back. In this model, there is more to an element than just its key. For example, elements can be objects consisting of multiple fields; when two objects are compared, a certain field, say their keys, are compared to each other. Obviously, an entire object cannot be deleted just because it has a key equal to the key of another object.

Sorting of multisets is basically the same as duplicate removal; however, some modifications are required. Sorting in the first model mentioned above is straightforward. In the algorithm for duplicate elimination in Section 4.2.3, we output not only the elements $i_1, \ldots, i_k$ but their multiplicities as well. We then sort the set $i_1, \ldots, i_k$, and then, since we know their multiplicities, we copy them in the final output as many times as it had been before in the multiset. Sorting k elements has cache complexity of $O\left(\frac{k}{B}\log_{\frac{M}{B}}\frac{k}{B}\right)$ which is less than $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i\right)$.

Sorting in the second model needs more modifications though:

Theorem 4.2.4. *Sorting a multiset of size N whose multiplicities are $N_1, N_2, \ldots, N_k$ can be done with cache complexity of:*

$$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right).$$

Proof. We again run the algorithm in Section 4.2.3 for duplicate elimination. In this new algorithm, though, when we have found a frequent element, instead of just removing its duplicates from the multiset, we move them in a separate list. Therefore, when algorithm finishes, we have a list for each item in the output set. The lists are maintained in an array in a back to back fashion; where the list of duplicates of an elements ends in the array, the list of duplicates of another one begins. A *head* and a *tail* pointer is kept for each distinct element that shows the beginning and the end of its duplicates in the array. If multiple frequent elements are discovered in a round, their lists are kept in the sorted order among themselves.

Definition 4.2.3. At each run of the frequent finder algorithm (Algorithm 4.1) for a particular value of C, the C-frequent elements are discovered and are kept in lists that are stored in sorted order in a back to back fashion. We denote the collection of lists produced in a round of the frequent element finder algorithm by a *super-list*. We also denote by $L_1, \ldots, L_p$, the super-lists produced in rounds $1, \ldots, p$ respectively.

When all elements are discovered, we start from the last super-list L_p and merge super-lists to the final list one by one in the reverse order (i.e., $L_p, L_{p-1}, \ldots, L_1$). The merge is exactly the same as the merge in the merge sort.

The correctness of the algorithm is obvious and follows from the correctness of the duplicate removal algorithm in Theorem 4.2.3. It remains to analyze the cache complexity of the algorithm. We focus on the extra work to the duplicate elimination algorithm and show that the extra work does not change the asymptotic cache complexity of the duplicate elimination algorithm.

The extra work can be categorized into two parts:

1. Moving duplicates of an element i_j into bucket $B[i_j]$ where we keep all duplicates of i_j.

2. Merging the super-lists into a final output list.

First we show that moving duplicates into buckets does not change the cache complexity. Remember that the algorithm works in rounds, and in each round, we look for C-frequent elements for a value of C. If only one element (say i_j) is found at each round, the cost of moving duplicates into a list instead of removing them is $\left\lceil \frac{N_{i_j}}{B} \right\rceil$. This cost is certainly less than $\lceil \frac{N}{B} \rceil$ which is the cache complexity of scanning the multiset once. Since in each round, we scan the multiset once, this cost cannot increase the cache complexity of the whole algorithm by more than a factor of two.

However, multiple C-frequent elements might be discovered at a round. Bucketing multiple elements is not as easy. Suppose, in a round, we have discovered r elements

that are C-frequent (and so $r \leq C$), we want to distribute these elements into r associated buckets as we scan the multiset. The technique is complicated and will be explained in detail in the distribution approach in Section 4.3.1. Though the general idea is to sort each consecutive group of r elements in the multiset and then distribute each group separately. According to Theorem 4.3.1, this takes $O\left(\frac{N'}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{r}{B}\right\}\right)$ number of cache misses where N' is the size of the multiset at the round. Remember that we have already spent $O\left(\frac{N'}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{C}{B}\right\}\right)$ at the round for duplicate removal, and since $r \leq C$, it means the cache complexity of this extra work does not change the asymptotic complexity of the duplicate removal algorithm.

Secondly, we show that merging the super-lists one by one, does not change the cache complexity. When it is time to merge L_i, the merged list has $|L_{i+1}| + \ldots + |L_p|$ elements; therefore, the number of cache misses required to merged these two lists is $O\left(\frac{\sum_{j=i}^{p}|L_j|}{B}\right)$ but this is not more than the cost we spent at round i to discover L_i: The size of the multiset at round i is $\sum_{j=i}^{p}|L_j|$ and to find some frequent elements in the multiset, we take a pass of the whole multiset once. The cache complexity of a pass through all elements is $O\left(\frac{\sum_{j=i}^{p}|L_j|}{B}\right)$. Therefore, the extra work of this part lower bounds the cache complexity of the duplicate elimination algorithm. Hence, this part cannot increase the cache complexity of the whole algorithm by more than a factor of two. □

It is clear to see that the asymptotic time complexity of the sorting algorithm in Theorem 4.2.4 is the same as that of duplicate removal algorithm in Theorem 4.2.3 and the number of comparison required is

$$O\left(N\log N - \sum_{i=1}^{k} N_i \log N_i\right)$$

which according to the lower bound in Section 2.1 is optimal.

4.2.5 Optimality of the Upper Bounds

In this section, we discuss the upper bounds for the three problems and explain the situations where the upper bounds match the lower bounds and where they do not.

In all of our three algorithms (namely determining the mode, duplicate removal and multisorting) we differ from the lower bound by an additive factor of at most $\frac{N}{B}\log\log M$. As previously mentioned, the size of a memory component M in present memory hierarchies is such that $\log\log M$ does not exceed from six[2]. Thus $\log\log M$ can be considered as a small constant and since $\frac{N}{B}$ is the cost of a scan of the multiset, $\frac{N}{B}\log\log M$ costs no more than a small number of scans of the multiset. Hence, our upper bounds are close to the lower bounds.

The algorithms can exactly match the lower bounds if some extra knowledge is given to them. This knowledge can be of two kinds: Knowledge of the multiset and knowledge of the memory hierarchy. Knowledge about the multiset can be in form of knowing the multiplicities of some elements. Knowledge of the memory hierarchy is information about the size of cache or block size. Note that the latter kind of knowledge makes the algorithms cache-aware as opposed to cache-oblivious.

4.2.5.1 Knowledge of the Multiset

Here we suppose some extra information about the multiplicities of some elements is provided to the algorithms. We make use of this extra knowledge and adapt the algorithms so that they match the lower bounds.

Determining the mode: Suppose we are given the frequency of the mode f. We can change the algorithm in Theorem 4.2.2 to match the lower bound. We run the algorithm for determining the N/f-frequent elements in Theorem 4.2.1 for just one round. The algorithm certainly finds the mode as it is N/f-frequent and by Theorem 4.2.1 causes $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}$ cache misses. Thus, the whole algorithm has complexity of $\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}$ which is the lower bound as it was proven in Theorem 3.5.1.

In fact, the value of $\lg\lg\frac{N}{f}$ with a constant additive error is all we need to determine the mode optimally. We run the same algorithm of determining the mode in Theorem 4.2.2, that is we again find C-frequent elements using Theorem 4.2.1 in

[2] $\log\log M > 6$ implies $M > 10^{19}$.

rounds. However we do not start from $C = 2^{2^1}$; the first value for C is $2^{2^{\lg\lg \frac{N}{f}}}$ and as before, we square the value of C each time. Since we have the value of $\lg\lg \frac{N}{f}$ but for a constant additive error, we only run the algorithm for finding the frequent elements for a constant number of runs and therefore, the cache complexity of the algorithm will match the lower bound. On the other hand, if our estimate of $\lg\lg \frac{N}{f}$ is high, we catch the mode in the first run of the algorithm. In this run of the algorithm, the value we assign to C is the right value of $\frac{N}{f}$ raised to some constant power. Consequently, the cache complexity of the algorithm is the lower bound times the constant number. Therefore, asymptotic cache complexity matches the lower bound.

Duplicate removal and multisorting: If we are given the multiplicities $N_1, \ldots, N_k$ of the elements, we can again apply the frequent element finder algorithm in Theorem 4.2.1 in rounds; this time, however, we run the algorithm in rounds for values of $C = \frac{N}{N_1}, \ldots, \frac{N}{N_k}$. It is not hard to follow the changes in the cache complexity computations in Theorems 4.2.3 and 4.2.4, the new cache complexity consists only of term

$$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i\right),$$

which is the lower bound.

Similar to finding the mode, knowing only the value of $\lg\lg \frac{N}{N_i}$ for $i = 1, \ldots, k$ with a constant additive error is enough for the algorithm to match the lower bound. For each element with multiplicity N_i, we will have to perform the C-frequent element finder algorithm in a constant number of rounds for the neighborhood values of $C = 2^{2^{\lg\lg \frac{N}{N_i}}}$.

4.2.5.2 Knowledge of the Memory Hierarchy

Here we present the "partially cache-aware" versions of our algorithms. We will show that knowing only the value of M can make our algorithms match the lower bounds. Our algorithms are simpler than the cache-aware algorithm of Arge et al. [4].

In all three algorithms we run the C-frequent element finder in Theorem 4.2.1 for a series of rounds for $C = 2^{2^1}, 2^{2^2}, \ldots$. The new algorithms work as the old ones with the only difference that since we now know the value of M, we start off with value $C = M = 2^{2^{\lg \lg M}}$ and continue similarly.

Correctness of the new algorithms is obvious: It follows from the correctness of the old algorithms by noting that all P-frequent elements for $P < M$ will be discovered at round $C = M$.

The cache complexity of the new algorithms can be analyzed as follows: Since $C \geq M$, the C-frequent element finder algorithm in Theorem 4.2.1 will have cache complexity:

$$O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{C}{B}\right\}\right) \;=\; O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{C}{B}\right).$$

Consequently, the cache complexity of determining the mode in Theorem 4.2.2 will reduce to:

$$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right).$$

As well, the cache complexity of duplicate removal and multisorting will reduce to:

$$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i\right).$$

Thus, knowing only the value of M (without knowing the value of B), makes all three algorithms match the lower bound. In fact, we only used the value of $\lg \lg M$, so knowing the value of $\log \log M$ with a constant additive error makes our algorithms optimal; this shows how little information our algorithms need to work optimally.

4.3 The Distribution Approach

In this section, we will take a different approach to solve the three problems. Due to the similarity between the new approach and the distribution sort, we will call this approach as the distribution approach.

As in Section 4.2, we will start with an algorithm which we will call the *distribution* algorithm and we will use the algorithm as a building block to give algorithms for the three problems. The cache complexities of the algorithms are, interestingly, exactly the same as those in the selection approach in Section 4.2.

The algorithm, that we will use as a building block, distributes elements of the multiset into C buckets of approximately equal sizes such that elements in any bucket are smaller than those in the following bucket.

The rest of this section is organized as follows: We will present the distribution algorithm in Section 4.3.1. Then we will present algorithms for the three problems (determining the mode, duplicate removal, and multi-sorting) in the three following sections.

4.3.1 The Distribution Algorithm

Here we present the distribution algorithm on which the algorithms are based. The algorithm is to distribute the elements into C buckets for a fixed C.

Definition 4.3.1. A distribution of elements of the multiset is called a *C-distribution* if the elements are partitioned into at least C buckets $B_1, \ldots, B_k$ $(k \geq C)$ such that for any two buckets B_i, B_j $(i < j)$, the elements in B_i are smaller than those in B_j. Furthermore, we require the size of the buckets to be not much larger than $\frac{N}{C}$, unless a bucket consists entirely of duplicates of the same element. In such a case, we do not impose any restriction on its size. More precisely, each bucket B_i satisfies at least one of the following two conditions:

- The size of B_i is not greater than $\frac{2N}{C}$ (i.e., $|B_i| \leq \frac{2N}{C}$),
- B_i consists entirely of elements with the same value.

In the following theorem we present the algorithm and show how C-distribution can be done cache-efficiently:

Theorem 4.3.1. *In a multiset of size N, C-distribution (as defined in Definition 4.3.1) can be performed with cache complexity of $O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{C}{B}\right\}\right)$.*

Proof. We first divide the elements of the multiset to $\frac{N}{C}$ groups of size C and then sort each group internally using any optimal cache-oblivious sort. As cache-oblivious sorting of C elements has cache complexity of $O\left(\frac{C}{B}\log_{\frac{M}{B}}\frac{C}{B}\right)$, the cache complexity of this task is

$$\frac{N}{C} \times O\left(\frac{C}{B}\log_{\frac{M}{B}}\frac{C}{B}\right) \quad = \quad O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{C}{B}\right).$$

Frigo et al. [6] proposed a cache-oblivious distribution sort in which there is a step called the *distribution step*. In this step, they show how k contiguous sorted subarrays, each of size k, can be k-distributed to k buckets with cache complexity of $O\left(\frac{k^2}{B}\right)$ which is the cache complexity of a scan over the subarrays. Their algorithm can also output the least element of each bucket. We call these representatives *pivots*.

Their distribution sort uses a procedure DISTRIBUTE(i, j, m) which distributes elements of m subarrays starting from subarray i into buckets starting from bucket j. The procedure works recursively and it turns out that the cache complexity of distributing all the subarrays to all buckets (i.e. DISTRIBUTE(1, 1, k)) is $O\left(\frac{k^2}{B}\right)$.

The goal is to distribute the N elements of the multiset among C buckets. In order to fulfill this task, we apply the distribution step of Frigo et al. [6] repeatedly for $k = C$: The multiset is already divided up into sorted subarrays of size C. we take C of these subarrays at a time (i.e. C^2 contiguous elements are considered at a time). Using their algorithm, we distribute these C^2 elements into C buckets. We then take the next C^2 elements (more precisely next C sorted subarrays each of size C) and distribute them among the C partially filled buckets. We continue until all elements of the multiset are distributed among the buckets. As we apply their algorithms $\frac{N}{C^2}$ times, the cache complexity of this task is:

$$\frac{N}{C^2} \times O\left(\frac{C^2}{B}\right) \quad = \quad O\left(\frac{N}{B}\right).$$

There is a subtle point in repeatedly applying the algorithm in [6]: When we apply their algorithm for the second time on, the buckets are no longer empty, but in their algorithm, they assume they start off with empty buffers. One can easily see that their result still holds even if the buckets are initially non-empty. Their algorithm always maintains a set of pivots for the buckets by which the elements are dispensed to the right bucket. Before the first application, they are all initialized to $+\infty$, and during the run of the algorithm, these pivots are updated on an on-going basis. We do not initialize the pivots to $+\infty$ in the second and later runs of the algorithm; we use the pivots from the last run of the algorithm as the pivots for the next run. Finally the pivots of the last run of the algorithm is the final pivots of the buckets.

We are almost done; elements are evenly distributed among the buckets as in a C-distribution. However, since we could have multiple copies of the same value in the multiset, duplicates of an element can be spread among two or more adjacent buckets. We will create a bucket for each of these elements and copy all copies of these elements into these buckets. We will show how this can be done cache-efficiently.

If copies of a value are spread among several buckets, that value must show up as the pivots of all those buckets (except possibly for the first.) We scan the buckets twice in opposite orders and generate a bucket for any pivot that occurs more than once and move all copies of the pivot into the corresponding bucket. The first scan is from left to right (more precisely, from B_1 to B_C). We always remember the pivot of the preceeding bucket during the scan and move copies of that pivot in the current bucket to a new bucket. Then we scan the buckets from right to left (i.e. from B_C to B_1) and again we remember the preceeding pivot during the scan and move copies of the pivot in the current bucket to the associated bucket. This step takes two scans of the buckets and causes $O\left(\frac{N}{B}\right)$ cache misses.

Correctness of the algorithm is obvious and follows from the correctness of the algorithm of Frigo et al. [6]. The cache complexity of the algorithm can be obtained by summing the cache misses at each step and works out to be

$$O\left(\frac{N}{B}\max\left\{1, \log_{\frac{M}{B}}\frac{C}{B}\right\}\right).$$

□

Time complexity of the algorithm or its number of comparisons can be computed easily. To sort elements in groups of size C, we have $\frac{N}{C} \times O(C \log C) = O(N \log C)$ number of comparisons. The time complexity of each run of the distribution algorithm by Frigo et al. [6] is $O(C^2)$; as we run their algorithm N/C^2 times, the total time is $O(N)$. The final two scans have obviously $O(N)$ time complexities. Thus, the total time complexity of the algorithm in Theorem 4.3.1 is $O(N \log C)$.

4.3.2 Determining the Mode

In this section, we use the distribution algorithm in Theorem 4.3.1 to find the mode in a multiset. The algorithm is essentially the same as that for determining the mode under the selection approach (Section 4.2.2); the only difference is that instead of applying the main theorem of the selection approach repeatedly, we apply the main theorem of the distribution approach (i.e. Theorem 4.3.1.)

It turns out that the cache complexity of the algorithm is exactly the same as in the selection approach:

Theorem 4.3.2. *The mode, with frequency f, of a multiset of size N can be found with cache complexity $O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right)$.*

Proof. We repeatedly run the distribution algorithm in Theorem 4.3.1 for the following values of C in order: $C = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^i}, \ldots, 2^{2^{\lceil \lg\lg N+1\rceil}}$. In the first step, the multiset is 2^{2^1}-distributed into subarrays to obtain level-one subarrays. In the next step, each level-one subarray is 2^{2^2}-distributed to get level-two subarrays, and so on; at step i, each subarray of level $i-1$ is 2^{2^i}-distributed to obtain subarrays of level i.

Definition 4.3.2. A subarray in which all elements have the same key value is called a *homogeneous* subarray. Subarrays that have at least two elements with different key values are called *heterogeneous* subarrays.

After all subarrays of level $i-1$ have been 2^{2^i}-distributed to subarrays of level i using the algorithm in Theorem 4.3.1, we check all subarrays to see if they are homogeneous, and also count the number of elements in each subarray. Obviously, this task can be done in a scan. We denote by s_{ho}, s_{he} the maximum size of homogenous and heterogeneous individual subarrays respectively. If $s_{ho} \geq s_{he}$, we have found the mode: We declare s_{ho} to be the frequency of the mode, report the corresponding element as the mode, and then the algorithm halts. If $s_{ho} < s_{he}$, the algorithm continues to the next level.

We now prove the correctness of the algorithm. There must be a heterogenous subarray in each level for the algorithm to continue running. Furthermore, a heterogeneous subarray is broken into at least two subarrays in the next level. Since this process cannot continue forever, the algorithm must halt. Clearly the element that is reported as the mode is indeed the mode, since the biggest homogeneous subarray is reported and all the heterogeneous subarrays are smaller than that.

The analysis of the cache complexity of this algorithm is the same as the mode-finding algorithm in the selection approach in Theorem 4.2.2. The cache complexity of the main algorithms in the selection approach (Theorem 4.2.1) and the distribution approach (Theorem 4.3.1) are exactly the same. If we prove that the algorithm will finish in level $\left\lceil \lg \lg \frac{N}{f} \right\rceil$ in the worst case as in the selection approach, since we run the algorithms for the same set of values of C, the overall cache complexities of both algorithms will be the same.

It remains only to show that in the worst case the algorithm will halt at level $\left\lceil \lg \lg \frac{N}{f} \right\rceil$ as in the corresponding theorem in the selection approach (i.e. Theorem 4.2.2.) One can show that the size of any heterogenous subarray in level i is not greater than $\frac{N}{2^{2^i}}$ (i.e. $s_{he} \leq \frac{N}{2^{2^i}}$.): Each heterogenous subarray of level i is produced by 2^{2^i}-distribution of a subarray in level $i-1$ and thus its size cannot be greater than $\frac{N}{2^{2^i}}$. Thus, in level $\left\lceil \lg \lg \frac{N}{f} \right\rceil$, the size of heterogenous subarrays are not greater than f (i.e. $s_{he} \leq f$.) Since the frequency of the mode is f, a homogeneous subarray must contain all copies of the mode and since $s_{ho} = f \geq s_{he}$, the algorithm halts in this

level. It is not hard to produce an example for each approach that the algorithms finish in level/round $\Omega\left(\log\log\frac{N}{f}\right)$. □

The number of comparisons in the algorithm can be computed as follows: The number of comparisons for C-distribution is $O(N \log C)$ as it was shown in Section 4.3.1. Therefore, the number of comparisons in the algorithm is:

$$\sum_{i=1}^{\lceil \lg\lg\frac{N}{f} \rceil} O\left(N \log 2^{2^i}\right) \;=\; O\left(N \log \frac{N}{f}\right).$$

According to Theorem 2.3.1 the number of comparisons is optimal.

4.3.3 Duplicate Elimination

In this section, we present an algorithm for duplicate elimination using the distribution approach.

We adopt the same terminology as in duplicate removal in the selection approach in Section 4.2.3. We are given a multiset of size N consisting of k elements $i_1, \ldots, i_k$ with multiplicities $N_1 \geq \ldots \geq N_k$ respectively. We will reduce the multiset to set $\{i_1, \ldots, i_k\}$. The results are exactly the same as in the selection approach.

Theorem 4.3.3. *Duplicate removal of a multiset of size N with multiplicities $N_1 \geq N_2 \geq \ldots \geq N_k$ can be done with cache complexity of:*

$$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right).$$

Proof. We again apply the C-distribution algorithm in Theorem 4.3.1 repeatedly for the following values of C: $C = 2^{2^1}, 2^{2^2}, \ldots, 2^{2^i}, \ldots, 2^{2^{\lceil \lg\lg N+1 \rceil}}$. After each level i, where we apply the 2^{2^i}-distribution algorithm, we check all resulting subarrays. If a subarray is homogeneous (i.e. consists totally of equal elements), we output the element as an element of the final set and remove the subarray. The remaining heterogeneous subarrays are compressed so that they are contiguous. The distribution algorithm is again applied to obtain the next level and so on.

The correctness and the cache complexity of the algorithm follows from the correctness and the cache complexity of the duplicate removal algorithm under the selection approach in Theorem 4.2.3. In the duplicate removal algorithm of the selection approach, the frequent finding algorithm in Theorem 4.2.1 is executed in different rounds and in each round some frequent elements are "discovered" and removed from the multiset. These elements are C-frequent elements (i.e. have multiplicities greater than $\frac{N}{C}$ as in Definition 4.2.1) for a particular value of C of the round. We just need to show that if an element is discovered and removed in round i of the duplicate removal in the selection approach, it is also discovered and removed by level i under the distribution approach (i.e. after i runs of the distribution algorithm).

We now prove the fact. We showed in the proof of Theorem 4.3.2 that after i runs of the distribution algorithm there cannot be any heterogeneous subarray of size greater than $\frac{N}{2^{2^i}}$. Hence, all 2^{2^i}-frequent elements must have been discovered and removed by level i. Note that in the duplicate elimination algorithm under the selection approach, in run number i of the frequent finder algorithm, we discover and remove 2^{2^i}-frequent elements. Therefore, those elements that are removed in round i of the algorithm under the selection approach are also discovered and removed by level i in the algorithm under the distribution approach.

The fact, we just showed, implies that the values of C for which we run the distribution algorithm of Theorem 4.3.1 is a subset of the values of C for which we run the frequent finder element in Theorem 4.2.1. By observing that the cache complexity of the frequent finding algorithm and the distribution algorithm are exactly the same, one can see that the cache complexity of the duplicate removal under the selection approach upper bounds that of the distribution approach.

It is not hard to come up with an example where an element of multiplicity $\frac{N}{2^{2^i}}$ is discovered at level $\Theta(i)$ in the distribution approach. This means the cache complexities of the two algorithms are asymptotically the same. □

Since the number of comparisons is also the same in the frequent finding algorithm in selection approach and distribution algorithm in distribution approach, the

number of comparisons required by the duplicate removal algorithm in Theorem 4.3.3 is the same as that in the selection approach which is $O\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$. According to Section 2.2 this is the optimal number of comparisons.

4.3.4 Multi-sorting

In this section, we will show how a multiset can be sorted using the distribution approach. Suppose we are given a multiset of size N consisting of k elements $i_1, \ldots, i_k$ with multiplicities $N_1 \geq \ldots \geq N_k$ respectively. We are to sort the multiset.

As was discussed in Section 4.2.4, sorting can be considered in two models. In the first model, copies of an element can be deleted and then regenerated by copying them back. In the second model, however, this is not possible; as there may be more fields in an element than just the key field.

Sorting under the first model is essentially the same as duplicate elimination; we just have to remember how many times an element happens and then copy it that number of times in the final output. Thus, the duplicate removal algorithm in the previous section will do this job.

Sorting in the second model is more challenging:

Theorem 4.3.4. *Sorting a multiset of size N with multiplicities $N_1, N_2, \ldots, N_k$ can be done with cache complexity of:*

$$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}} N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right).$$

Proof. The algorithm is basically the same as duplicate removal algorithm in Theorem 4.3.3; we apply the C-distribution algorithm in Theorem 4.3.1 repeatedly for a set of values for C. In the duplicate removal algorithm, after each level i, homogeneous subarrays are deleted. In the sorting algorithm, instead of deleting these subarrays we copy them in order to a separate list L_i.

When all elements have been removed from the multiset, we have a collection of lists $L_1, \ldots, L_p$, where p is the number of levels the algorithm runs. Note than each

list L_i is sorted, since the homogeneous subarrays have been copied in order. These lists are exactly the "super-lists" we formed in the sorting algorithm in the selection approach in Theorem 4.2.4. These lists are merged together one at a time in the reverse order as the super-lists were merged in Theorem 4.2.4.

Obviously the cache complexity of the algorithm is the same as that in the selection approach. □

Similarly, the time complexity or the number of comparisons of the algorithm is the same as the sort in the selection approach which is $O\left(N \log N - \sum_{i=1}^{k} N_i \log N_i\right)$. Hence, according to Section 2.1, the number of comparisons is optimal.

4.4 The Randomized Approach

The two previous approaches were deterministic and do not quite match the lower bounds. In this section, we will present a randomized approach that does match the lower bounds.

In Section 4.4.1, we show that the same lower bounds as for deterministic algorithms hold for randomized algorithms. Then we match these lower bounds using randomized algorithms in Section 4.4.2.

4.4.1 Randomized Lower Bounds

We will use Yao's minimax principle [9] to obtain randomized lower bounds for our three problems. Yao's minimax principle, in short, says that the *average-case* complexity of an optimal deterministic algorithm lower bounds the expected running time of any randomized algorithm. Thus we will first show average-case cache-oblivious lower bounds.

We obtained our cache-oblivious lower bounds by first showing lower bounds in the comparison model in Chapter 2. We then defined the notion of an I/O-tree and used the main theorem by Arge et al. [4] that relates the size of an I/O-tree in the cache-aware model and a decision-tree in the comparison model to obtain lower bounds in

the cache-aware model that also hold for the cache-oblivious models. We will follow the same path to get average-case lower bounds in the cache-oblivious model.

The results in Chapter 2 are all average-case lower bounds in the comparison model; therefore, the lower bounds hold for average height of the corresponding decision trees. The main theorem in Section 3.2 works for average heights as well: The theorem proves an inequality that relates the I/O-height of an I/O tree and height of a decision tree. With the same proof as presented in Section 3.2, one can prove the same inequality holds if we replace the I/O-height of an I/O-tree by *average I/O-height* (i.e., average number of I/O-nodes on the paths from the root to the leaves) of the tree and the height of the tree by the *average height* of the tree. Consequently, the lower bounds will hold for average-case cache complexities of our three problems as well. Thus, the average-case lower bounds are the same as the worst-case ones in the cache-oblivious model.

By using the Yao's minimax principle, since we know that the lower bounds remain in tact for the average-case cache complexities of the three problems, the same lower bounds will also hold for the expected running time of randomized algorithms.

4.4.2 Randomized Upper Bounds

In this section, we present our randomized algorithms. Thus far, we have used two deterministic approaches to solve the problems: The selection approach and the distribution approach. Both can be randomized. We will only show how the selection approach can be randomized; as the distribution approach can be randomized in a similar manner.

In the selection approach we use the C-frequent finding algorithm of Section 4.2.1 for a series of values C. In the randomized approach, we will use randomization to estimate the right value for C so that we do not have to try all values for C and then confirm the value. We then use this method to address the problem of finding the mode. Finally we give algorithms for the other two problems. The other two problems are similar and so we can describe their algorithms together.

4.4.2.1 Determining the Mode

We still use the C-frequent finder algorithm in Section 4.2.1, although instead of starting from small values of C and squaring it at each step, we will estimate a good value for C using the randomized techniques and jump to that value of C.

Theorem 4.4.1. *The mode, with frequency f, of a multiset of size N can be found with expected cache complexity $O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\right\}\right)$.*

Proof. We require a random sample of the elements to estimate the appropriate value of C for our C-frequent finder algorithm. The sample must be large enough to produce a good estimate, with high confidence. It must also be small enough so that working with the sample does not dominate our cost.

We have a high degree of latitude in choosing the sample size. Something around $\sqrt{N}$ is a reasonable choice. Making it $\sqrt{N}\ln N$ simplifies some of the calculations in the proof. The proof is also simplified if we sample with replacement (i.e. the same element can be chosen more than once.)

We can afford to sort them as sorting $\sqrt{N}\ln N$ elements has cache complexity of

$$O\left(\frac{\sqrt{N}\ln N}{B}\log_{\frac{M}{B}}\frac{\sqrt{N}\ln N}{B}\right),$$

which is less than the $\frac{N}{B}$ required for a scan of the entire multiset. After sorting the sample, we scan and find its mode. We denote the frequency of the sample mode by p.

The estimate of the frequency of the mode in the multiset is $f' = \frac{p\sqrt{N}}{\ln N}$; thus we start by finding C-frequent elements for $C = \frac{N}{f'}$. If there is no C-frequent element for this value of C, we square C (i.e. $C \leftarrow C^2$) and try to find the C-frequent elements for the new value of C and so on.

The correctness of the algorithm is obvious: Sooner or later the right value is assigned to C and the mode is discovered. It remains to show the expected cache complexity of the algorithm. Depending whether the estimated value for the frequency

of the mode f' is less than the actual frequency of the mode or is greater than it, we have two cases:

1. We have underestimated the value of f (i.e. $f' < f$): In this case, we find the mode on the first run of the frequent element finder algorithm, but as the value of C is greater than what it should be, the cache complexity of the algorithm is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{f'B}\right)$.

2. We have overestimated the value of f (i.e. $f' > f$): In this case, we would first run the frequent element finder algorithm for $C = \frac{N}{f'}$ and the algorithm would fail to find a value, then we would run the algorithm for C^2 and so on. Since we square the value of C each time, it is easy to see that the number of times we have to repeat the algorithm to get to the mode is at most $O\left(\log\log_{\frac{N}{f'}}\frac{N}{f}\right)$. The cache complexity of finding the C-frequent elements, as we proved in Theorem 4.2.1, is $O\left(\frac{N}{B}\max\{1, \log_{\frac{M}{B}}\frac{C}{B}\}\right)$: There are two terms $\frac{N}{B}$ and $\log_{\frac{M}{B}}\frac{C}{B}$. The term that keeps us away from the lower bounds is the first. The $\log_{\frac{M}{B}}\frac{C}{B}$ does not cause overflow from the lower bound even when we try all values for C. In this case, the contribution of the term $\frac{N}{B}$ is $O\left(\frac{N}{B}\log\log_{\frac{N}{f'}}\frac{N}{f}\right)$.

We now compute the probability of occurrence of each of these cases. We use the Chernoff lower tail and upper tail bounds [9] to show that the probability of f' being far from f is small. Let us first explain what we mean by f' being far from f, since if all elements are distinct, $f = 1$, however one can see that our estimate $f' = \Theta\left(\frac{\sqrt{N}}{\ln N}\right)$. In fact, we are interested in the values of $\lg\lg\frac{N}{f}$ and $\lg\lg\frac{N}{f'}$, and we implicitly prove that the probability of these values being far from each other is tiny.

We first analyze the probability of occurrence of the first case. Without loss of generality we can assume that $f > \sqrt{N}$, since otherwise the cache complexity of $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{f'B}\right)$ is optimal. We took $\sqrt{N}\ln N$ elements $s_1, \ldots, s_{\sqrt{N}\ln N}$ from the multiset. For $i = 1, \ldots, \sqrt{N}\ln N$, set $x_i = 1$ if s_i is a copy of the mode, and set $x_i = 0$ otherwise. The probability of each element being a copy of the mode is $\frac{f}{N}$ (i.e. $P[x_i = 1] = \frac{f}{N}$.) Hence, the expected value μ of the sum $S = \sum_{i=1}^{\sqrt{N}\ln N} x_i$ is

$\mu = \frac{f}{N} \times \sqrt{N} \ln N = \frac{f \ln N}{\sqrt{N}}$.

The actual multiset mode occurs S times in the sample. The mode of the sample occurs p times. Hence $S \leq p$, and therefore, $S \leq \frac{f' \ln N}{\sqrt{N}}$. We can now apply the Chernoff bound to bound the lower tail of the sum S. According to the Chernoff bound on the lower tail:

$$\begin{aligned} P[f' < (1-\delta)f] &\leq P[S < (1-\delta)f \times \frac{\ln N}{\sqrt{N}}] \\ &= P[S < (1-\delta)\mu] \\ &< e^{\frac{-\mu\delta^2}{2}} \\ &= e^{\frac{-f \ln N \delta^2}{2\sqrt{N}}} \\ &\leq e^{\frac{-\ln N \delta^2}{2}}. \qquad (as \;\; f > \sqrt{N}) \\ &= N^{-\frac{\delta^2}{2}}. \end{aligned}$$

The expected cache complexity of the algorithm in the first case can be computed as follows:

$$\begin{aligned} E &= \int_0^1 P[f' = (1-\delta)f] \times \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{f'B} \, d\delta \\ &\leq \int_0^1 N^{-\frac{\delta^2}{2}} \times \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{(1-\delta)fB} \, d\delta \\ &= O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{fB}\right). \end{aligned}$$

Thus the expected cache complexity is the optimal cache complexity.

Now we analyze the probability of the occurrence of the second case where $f' > f$. Without loss of generality, we can assume that $f' > \sqrt{N}$; since otherwise $f < f' < \sqrt{N}$ and consequently $\frac{N}{f} > \sqrt{N}$ and $\frac{N}{f'} > \sqrt{N}$ which means the mode is discovered after at most two runs of the algorithm. We will again use the Chernoff bound to bound the probability $P[f' = (1+\delta)f]$.

$$f' = (1+\delta)f \;\; \Rightarrow \;\; \frac{\sqrt{N}p}{\ln N} = (1+\delta)f \;\; \Rightarrow \;\; p = \frac{f \ln N}{\sqrt{N}}(1+\delta)$$

We will now show that the probability of getting p copies of an element t is small. Consider the sample set $s_1, \ldots, s_{\sqrt{N} \ln N}$. Define $x_i = 1$ if $s_i = t$ and $x_i = 0$ otherwise.

Obviously $P[x_i = 1] = \frac{f_t}{N}$ where f_t is the frequency of t in the multiset. The mean value for the sum $S = \sum_{i=1}^{\sqrt{N}\ln N} x_i$ is $\mu = \frac{f_t\sqrt{N}\ln N}{N} = \frac{f_t \ln N}{\sqrt{N}}$. Since $f > f_t$, the probability can be rewritten as:

$$\begin{aligned} P[S = (1+\delta)\frac{f\ln N}{\sqrt{N}}] &\leq P[S > p = (1+\delta)\frac{f_t l N}{\sqrt{N}}] \\ &= P[S > (1+\delta)\mu] \end{aligned}$$

Using the Chernoff bound on the upper tail, we obtain:

$$\begin{aligned} P[S = (1+\delta)\frac{f\ln N}{\sqrt{N}}] &\leq P[S > (1+\delta)\mu] \\ &< 2^{-((1+\delta)\frac{f}{f_t})\mu} \qquad (for\ \delta > 2e-1) \\ &= 2^{-(1+\delta)\frac{f\ln N}{\sqrt{N}}} \\ &\leq 2^{-p} = 2^{-\frac{f'\ln N}{\sqrt{N}}} = N^{-\frac{f'}{\sqrt{N}}}. \end{aligned}$$

Thus far, we have bounded the probability $P[S = (1+\delta)\frac{f\ln N}{\sqrt{N}}]$ for a particular element. The probability that $P[S = (1+\delta)\frac{f\ln N}{\sqrt{N}}]$ is true for at least one element is at most N times the probability for a particular element, thus:

$$\begin{aligned} P[f' = f(1+\delta)] &\leq NP[S = (1+\delta)\frac{f\ln N}{\sqrt{N}}] \\ &\leq N \times N^{-\frac{f'}{\sqrt{N}}} \\ &= N^{-\left(\frac{f'}{\sqrt{N}}-1\right)}. \end{aligned}$$

Note that we assumed that $f' > \sqrt{N}$ and hence $\frac{f'}{\sqrt{N}} - 1 > 0$. We can now compute the expected cache complexity of the algorithm in this case. As we mentioned earlier, the overflow from the lower bound is $\frac{N}{B}\log\log_{\frac{N}{f'}}\frac{N}{f}$, hence:

$$\begin{aligned} E &= \int_f^{+\infty} P[f' = (1+\delta)f] \times \frac{N}{B}\log\log_{\frac{N}{f'}}\frac{N}{f}\, df' \\ &\leq \int_f^{+\infty} N^{-\left(\frac{f'}{\sqrt{N}}-1\right)} \times \frac{N}{B}\log\log_{\frac{N}{f'}}\frac{N}{f}\, df' \\ &= O\left(\frac{N}{B}\right). \end{aligned}$$

We have shown that the overflow of the cache complexity of the lower bound is of $O\left(\frac{N}{B}\right)$ and thus negligible. Thus the expected cache complexity is asymptotically optimal.

In case $\frac{N}{f} < M$, since we scan the multiset at least once, the correct upper bound is $\frac{N}{B}$. □

4.4.2.2 Duplicate Elimination and Sorting

Having explained how we determine the mode optimally, the algorithms for duplicate elimination and sorting follows quickly. We run our sampling algorithm repeatedly to help us find the appropriate values for C. For each value of C, we find C-frequent elements as before.

Theorem 4.4.2. *Sorting or eliminating the repeated values of a multiset of size N whose multiplicities are $N_1 \geq N_2 \geq \ldots \geq N_k$ can be done with expected cache complexity of:*

$$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i, \frac{N}{B}\right\}\right).$$

Proof. The algorithms are the same as those of duplicate removal and sorting under the selection approach in Sections 4.2.3 and 4.2.4; the only difference is that we use our sampling algorithm to help us jump from a value of C where elements of multiplicity N_i are discovered to the next appropriate value of C where elements of N_{i+1} are discovered. Thus we do not have to try other values of C in between the two values.

The correctness of the algorithms follows from the correctness of the corresponding algorithms in the selection approach. The expected cache complexity of the algorithms can be computed as follows. The cache complexity of C-frequent element finder in Section 4.2.1 consists of two terms $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{C}{B}\right)$, and $O\left(\frac{N}{B}\right)$. The latter term causes overflow from the lower bounds in all three algorithms of the selection approach. We showed in the randomized mode finding that by using the sampling

algorithm and jumping over unnecessary steps we can save some scans and thus save this extra cost.

The algorithms for duplicate removal and sorting consist of at most k runs of the sampling algorithm and each step is similar to the mode-finding algorithm. As in the case of the mode finding algorithm, one can show that the expected extra cost at each step is negligible compared to the overall cache complexity of each step. Since the expected cache complexity of the each algorithm is the sum of the expected cache complexities of the steps, the total extra cost is negligible and the expected cache complexity of each algorithm is $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i\right)$. □

4.5 Summary

In this section, we summarize the upper bounds we achieved by our algorithms. The bounds of the deterministic algorithms are mentioned in Table 4.2. The bounds for the randomized algorithms, which are optimal, are shown in Table 4.3.

Table 4.2: Upper bounds of the deterministic algorithms

Algorithm	**Upper Bound**
Determining the mode	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\log\log\frac{N}{f}\right\}\right)$
Duplicate Elimination	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right)$
Multisorting	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \sum_{i=1}^{k}\frac{N_i}{B}\log\log\frac{N}{N_i}\right\}\right)$

Table 4.3: Upper bounds of the randomized algorithms

Algorithm	**Upper Bound**
Determining the mode	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}, \frac{N}{B}\right\}\right)$
Duplicate Elimination	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \frac{N}{B}\right\}\right)$
Multisorting	$O\left(\max\left\{\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}N_i,\ \frac{N}{B}\right\}\right)$

Chapter 5

Conclusion

In this thesis, we have studied three problems related to multisets in the cache-oblivious model: determining the mode, duplicate removal, and multi-sorting. Suppose we are given a multiset of size N with k distinct elements $i_1, i_2, \ldots, i_k$ with multiplicities $N_1, N_2, \ldots, N_k$. The problem of determining the mode is finding the element with the greatest multiplicity. Duplicate elimination is reducing the original multiset to the set $\{i_1, \ldots, i_k\}$. Multi-sorting is the problem of sorting the input list and outputting it in the sorted order. The last problem has the additional problem that each element may have extra information associated with it , and that data must be retained.

We have presented the known lower bounds for the cache complexity of each of these problems. Determining the mode has the lower bound of $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{fB}\right)$ where f is the multiplicity of the most frequent element and M is the size of the cache and B is size of a block in cache. The lower bound for the cache complexity of duplicate removal and multi-sorting is $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{k}\frac{N_i}{B}\log_{\frac{M}{B}}\frac{N_i}{B}\right)$.

We have considered both deterministic and randomized algorithms for these problems. In terms of the deterministic algorithms, we have followed two approaches: The selection and the distribution approach. In both of these approaches, the cache complexity of algorithms can differ from the lower bounds by an additive term of $O\left(\frac{N}{B}\log\log M\right)$ away from the lower bounds. Our randomized algorithms have costs

within a constant factor of the lower bounds.

Even our deterministic algorithms that may not be optimal in the cache-oblivious model can easily be patched so they work optimally in the cache-aware model. These algorithms are simpler than previously-existing cache-aware algorithms for the problems.

As for the future work, there is certainly room for improvement on the deterministic cache-oblivious upper bounds for the three problems. Since the deterministic lower and upper bounds do no match. One other interesting related area of future work is to design optimal cache-oblivious algorithms for adaptive sorting. Adaptive sorting algorithms take advantage of the existing order in the input to improve their time complexities (refer to [5] for a survey). Their performance is measured as a function of both the input size and the amount of disorder in the input. Designing cache-oblivious adaptive sorting algorithms is a new field to explore.

Bibliography

[1] A. Aggarwal and J. S. Vitter, *The I/O complexity of sorting and related problems*, Proceedings of the 14th International Colloquium on Automata, Languages, and Programming, LNCS, vol. 267, Springer-Verlag, July 1987, pp. 467–478.

[2] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM **31(9)** (1988), 1116–1127.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, *Cache-oblivious priority queue and graph algorithm applications*, Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, Montréal, Québec, Canada, May 19–21, 2002 (New York, NY, USA) (ACM, ed.), ACM Press, 2002, pp. 268–276.

[4] L. Arge, M. Knudsen, and K. Larsen, *A general lower bound on the I/O-complexity of comparison-based algorithms*, In Proceedings of Workshop on Algorithms and Data Structures (WADS'93), Springer-Verlag, 1993.

[5] V. Estivill-Castro and D. Wood, *A survey of adaptive sorting algorithms*, ACM Computing Surveys **24** (1992), no. 4, 441–476.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1999, pp. 285–297.

[7] F. K. Hwang and S. Lin, *A simple algorithm for merging two disjoint linearly ordered sets*, SIAM Journal on Computing **1** (1972), no. 1, 31–39.

[8] J. Misra and D. Gries, *Finding repeated elements*, Science of Computer Programming **2** (1982), 143–152.

[9] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.

[10] I. Munro and P. Spira, *Sorting and searching in multisets*, SIAM Journal on Computing **5** (1976), 1–8.

[11] D. D. Sleator and R.E. Tarjan, *Amortized efficiency of list update and paging rules*, Commun. ACM **28(2)** (1985), 202–208.