2021

OPTIMAL CONSTRUCTION OF A LAYER-ORDERED HEAP AND ITS APPLICATIONS

Jake Pennington

OPTIMAL CONSTRUCTION OF A LAYER-ORDERED HEAP AND ITS APPLICATIONS

By

Jake Rooster Pennington

Bachelor of Arts, University of Montana, Missoula, MT, 2019

Bachelor of Science, University of Montana, Missoula, MT, 2020

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Arts
in Mathematics

The University of Montana
Missoula, MT

Autumn 2021

Approved by:

Ashby Kinch Ph.D., Dean
Graduate School

Oliver Serang Ph.D., Chair
Computer Science

Cory Palmer Ph.D.
Mathematical Sciences

Kelly McKinnie Ph.D.
Mathematical Sciences

Pennington, Jake, M.A., October 2021 Mathematics

Optimal Construction of a Layer-Ordered Heap and its Applications

Chairperson: Oliver Serang

The layer-ordered heap (LOH) is a simple data structure used in algorithms that perform optimal top-k on $X + Y$, algorithms with the best known runtime for top-k on $X_1 + X_2 + \cdots + X_m$, and the fastest method in practice for computing the most abundant isotopologue peaks in a chemical compound. In the analysis of these algorithms, the rank, α, has been treated as a constant and n, the size of the array, has been treated as the sole parameter. Here, we explore the algorithmic complexity of LOH construction with α as a parameter, introduce a few algorithms for constructing LOHs, analyze their complexity in both n and α, and demonstrate that one algorithm is optimal in both n and α for building a LOH of any rank. We then apply this to improve performance in applications where they are employed, find an estimate for the optimal α given an n and k for top-k on $X + Y$, and derive a novel algorithm for top-k on a multinomial distribution. Finally, we show that the results of our LOH analysis correspond with empirical experiments of runtimes when applying the LOH construction algorithms to both a common task in machine learning and top-k on $X_1 + X_2 + \cdots + X_m$ and that our estimate of the optimal α for top-k on $X + Y$ corresponds well with empirical data.

ACKNOWLEDGMENTS

I would like to express my gratitude and appreciation to my committee for taking the time to evaluate my defense. I thank my coauthors Patrick Kreitzberg and Kyle Lucke for the time we spent working together to make new discoveries. I especially thank my advisor Dr. Oliver Serang for all of his help with my academic and personal growth. Finally, I would like to thank my parents, Joe and Carmen Pennington, for cultivating my appreciation for learning, my brother, Pepper Pennington, for motivating me to continue my education, and my sister, Jessica Riggin, for always being there for me.

This work was supported by Grant No. 1845465 from the National Science Foundation. The LOHification algorithms presented in this paper, implemented in C++17, can be found freely at `https://figshare.com/articles/software/Lohify/16837213`. The Cartesian product algorithms benchmarked in this paper, implemented in C++17, can be found freely at `https://figshare.com/articles/software/cartesian_product_tree_code/16837198`.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 Introduction

Sorting is one of the oldest problems in computing. There are several types of sorting algorithms, but we will focus on comparison based sorting because it requires only that the objects to be sorted can be pairwise compared. It has been known for a long time that the lower bound for comparison sort is in $\Omega(n \cdot \log(n))$ where n is the size of the list we are sorting [1]. Algorithms that achieve this bound have been known since the 1940's [2].

Because efficient sorting algorithms exist that achieve the lower bound in the worst case, comparison based sorting is typically thought of as an area for little practical and no theoretical improvement. However; many of the problems traditionally solved using sorting do not actually require a total ordering. This gives rise to the use of partial orderings to solve these problems.

Partial orderings have been used to replace sorting in algorithms where sorting is the limiting factor. For example, soft heaps [3] made possible the algorithm with fastest known worst-case runtime for computing the minimum spanning tree [4]. Soft heaps work by bounding the number of "corrupt" elements in a tree, which will be moved forward from their true position in the sorted order. They are of theoretical interest because they can be constructed in linear time, can insert elements and find-min in constant time, and they can delete-min in amortized constant time [3].

Soft heaps have also been used to perform optimal top-k, the problem of generating

the k smallest items in a collection, on $X + Y$ [5]; however, that algorithm is slow in practice due to the fragmented nature of soft heaps leading to more overhead and poor cache performance [6].

The layer-ordered heap (LOH) was invented to perform practical and efficient top-k on $X + Y$ in a way that also achieves optimal theoretical performance [6]. The LOH is qualitatively similar to a soft heap in that they both produce partial orderings, but different in that LOHs are contiguous in memory and have a simple data structure. LOHs are powerful in that they are simple to implement and are fast in practice.

In addition to optimal top-k on $X+Y$, layer-ordered heaps (LOHs) are used in the algorithm with best known runtime for top-k on $X_1 + X_2 + \cdots + X_m$ [7], practically fast calculation of where a statistical threshold occurs on monotonic functions [8], the fastest known method for computing the most abundant isotopologues of a compound [9], and for approximating marginal distributions under sums and differences of categorically distributed discrete random variables [10].

More specifically, a LOH of size n and rank α is an array of length n partitioned into ℓ layers (denoted $L_0, L_1, \ldots, L_{\ell-1}$) where every element in L_i is less than or equal to every element in L_j for all $i < j$ and the size of each layer, $|L_i|$, is $p_i - p_{i-1}$ where p_i, the i^{th} pivot index, is calculated as $p_i = \left\lceil \sum_{j=0}^{i} \alpha^j \right\rceil$ (ensuring the ratio of the layer sizes, $\frac{|L_{i+1}|}{|L_i|}$, tends to α as the index, i, tends to infinity). We stop computing pivots when the next pivot would be greater than n and the size of the final layer is the difference between the size of the array and the last pivot. Specifically: the construction of a LOH takes an array of n pairwise comparable elements, partitions it (in-place) into layers (that grow exponentially) based on α, and produces an array of the pivot indices. Fig 1.1 depicts a LOH of rank $\alpha = 2$.

Throughout this paper, the process of constructing a LOH of rank α from an array of length n will be denoted "LOHification." While LOHify with $\alpha = 1$ is equivalent

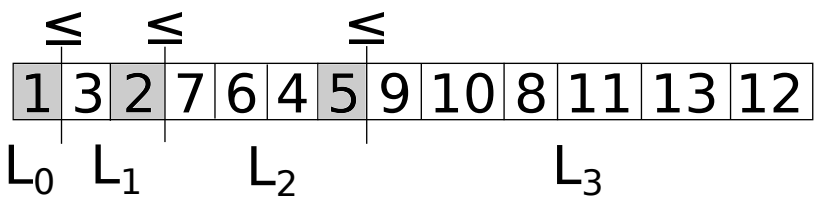

Figure 1.1 **A LOH of rank 2.** Pivot indices are shaded in gray. Notice that the last layer is not full.

to comparison sort and $\alpha \geq 2$ can be performed in $O(n)$ operations [7], the optimal runtime in terms of both n and α is unknown. Likewise, there is no known LOHify algorithm that is optimal in both n and α.

In this paper, we will derive the optimal construction of a layer-ordered heap for all of its parameters, use this to improve the performance of applications where it is employed, and present a novel algorithm for performing top-k on a multinomial distribution.

CHAPTER 2 Methods

2.1 Optimal construction of a LOH in terms of n and α

Here, we derive a lower bound runtime for LOHification, describe a few algorithms for LOHification, prove their runtimes, and demonstrate optimality of one method in both n and α. We later demonstrate the practical performance of these methods on both a non-parametric statistical test and a top-k on $X_1 + X_2 + \cdots + X_m$. Throughout this section, let $r(n, \alpha)$ denote the runtime of LOHification of rank α on an array of size n.

2.1.1 A lower bound on LOHification

Here we will prove an asymptotic lower bound on the complexity of constructing a LOH in terms of n and α by first proving bounds on variables and then using those to bound the process as a whole.

2.1.1.1 Bounds on variables

Lemma 2.1.0.1 (Upper bound on the number layers). *An upper bound on the number of layers, ℓ, in a LOH of n elements is* $\log_\alpha(n \cdot (\alpha - 1) + 1) + 1$.

Proof. Because the final pivot, $p_{\ell-2}$, can be no more than n, the size of our array, we

have the following inequality:

$$\begin{aligned}
\left\lceil \sum_{i=0}^{\ell-2} \alpha^i \right\rceil &\leq n \\
\sum_{i=0}^{\ell-2} \alpha^i &\leq n \\
\frac{\alpha^{\ell-1}-1}{\alpha-1} &\leq n \\
\alpha^{\ell-1}-1 &\leq n \cdot (\alpha-1) \\
\alpha^{\ell-1} &\leq n \cdot (\alpha-1)+1 \\
\ell-1 &\leq \log_\alpha(n \cdot (\alpha-1)+1) \\
\ell &\leq \log_\alpha(n \cdot (\alpha-1)+1)+1.
\end{aligned}$$

□

Lemma 2.1.0.2 (Lower bound on the number layers). *A lower bound on the number of layers, ℓ, in a LOH of n elements is* $\log_\alpha(n \cdot (\alpha-1)+1)$.

Proof. Because an additional pivot (after the final pivot) must be more than n, the size of our array, we have the following inequality:

$$\begin{aligned}
\left\lceil \sum_{i=0}^{\ell-1} \alpha^i \right\rceil &> n \\
\sum_{i=0}^{\ell-1} \alpha^i &\geq n, \text{ because } n \text{ is an integer;} \\
\frac{\alpha^{\ell}-1}{\alpha-1} &\geq n \\
\alpha^{\ell}-1 &\geq n \cdot (\alpha-1) \\
\alpha^{\ell} &\geq n \cdot (\alpha-1)+1 \\
\ell &\geq \log_\alpha(n \cdot (\alpha-1)+1)
\end{aligned}$$

$$\ell \;>\; \log_\alpha(n \cdot (\alpha - 1)).$$

□

Lemma 2.1.0.3 (Asypmtotic number of layers). *For $\alpha > 1$, the number of layers as n grows is asymptotic to* $\log_\alpha(n \cdot (\alpha - 1) + 1)$.

Proof. We know $\log_\alpha(n \cdot (\alpha - 1) + 1) \leq \ell \leq \log_\alpha(n \cdot (\alpha - 1) + 1) + 1$.

$$\begin{aligned}
& \lim_{n\to\infty} \frac{\log_\alpha(n \cdot (\alpha - 1) + 1) + 1}{\log_\alpha(n \cdot (\alpha - 1) + 1)} \\
= \; & \lim_{n\to\infty} \frac{\log_\alpha(n \cdot (\alpha - 1) + 1)}{\log_\alpha(n \cdot (\alpha - 1) + 1)} + \frac{1}{\log_\alpha(n \cdot (\alpha - 1) + 1)} \\
= \; & 1 + 0 \\
= \; & 1.
\end{aligned}$$

□

Lemma 2.1.0.4 (Upper bound on the size of a layer). *An upper bound on the size of layer i is $|L_i| \leq \lceil \alpha^i \rceil$.*

Proof. The i^{th} layer, $|L_i|$, as defined above, can be calculated by:

$$\begin{aligned}
|L_i| &= p_i - p_{i-1} \\
&= \left\lceil \sum_{j=0}^{i} \alpha^j \right\rceil - \left\lceil \sum_{j=0}^{i-1} \alpha^j \right\rceil \\
&\leq \lceil \alpha^i \rceil + \left\lceil \sum_{j=0}^{i-1} \alpha^j \right\rceil - \left\lceil \sum_{j=0}^{i-1} \alpha^j \right\rceil \\
&\leq \lceil \alpha^i \rceil.
\end{aligned}$$

□

2.1.1.2 Lower bound of LOHification

Here we will show that for $\alpha > 1$, the computational complexity of LOHification is in $\Omega\left(n\log(\frac{1}{\alpha-1}) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right)$. To do this, we will first prove a few lemmas that we will use in the proof of our theorem.

Lemma 2.1.0.5. *For $\alpha > 1, \lceil\alpha^i\rceil \cdot \log(\lceil\alpha^i\rceil) \sim \alpha^i \cdot \log(\alpha^i)$ as $n \to \infty$.*

Proof.

$$\begin{aligned}\alpha^i \cdot \log(\alpha^i) &\leq \lceil\alpha^i\rceil \cdot \log(\lceil\alpha^i\rceil) \\ &\leq (\alpha^i + 1) \cdot \log(\alpha^i + 1).\end{aligned}$$

We can combine this with the fact that i goes to infinity as n goes to infinity to show that the expressions are asymptotic as n goes to infinity.

$$\begin{aligned}\lim_{i\to\infty} \frac{(\alpha^i + 1) \cdot \log(\alpha^i + 1)}{\alpha^i \cdot \log(\alpha^i)} &= \lim_{i\to\infty} \frac{\alpha^i \log(\alpha^i + 1)}{\alpha^i \log(\alpha^i)} + \frac{\log(\alpha^i + 1)}{\alpha^i \log(\alpha^i)} \\ &= \lim_{i\to\infty} \frac{\log(\alpha^i + 1)}{\log(\alpha^i)} + \frac{\log(\alpha^i + 1)}{\alpha^i \log(\alpha^i)}\end{aligned}$$

which, by L'Hôpital's rule;

$$\begin{aligned}&= \lim_{i\to\infty} \frac{\frac{\alpha^i \log(\alpha)}{\alpha^i+1}}{\log(\alpha)} + \frac{\frac{\alpha^i\cdot\log(\alpha)}{\alpha^i+1}}{\alpha^i \cdot \log(\alpha) \cdot (\log(\alpha^i) + 1)} \\ &= \lim_{i\to\infty} \frac{\alpha^i}{\alpha^i + 1} + \frac{1}{(\alpha^i + 1) \cdot (\log(\alpha^i) + 1)} \\ &= \lim_{i\to\infty} 1 - \frac{1}{\alpha^i + 1} \\ &= 1.\end{aligned}$$

□

Lemma 2.1.0.6. *For $\alpha > 1$, $\frac{\log(n\cdot(\alpha-1)+1)}{\alpha-1} \in o(n)$.*

Proof.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{\log(n\cdot(\alpha-1)+1)}{\alpha-1}}{n} &= \lim_{n\to\infty} \frac{\log(n\cdot(\alpha-1)+1)}{n\cdot(\alpha-1)} \\
&= \lim_{n\to\infty} \frac{\alpha-1}{(\alpha-1)\cdot(n\cdot(\alpha-1)+1)} \text{ by L'Hôpital's rule} \\
&= \lim_{n\to\infty} \frac{1}{n\cdot(\alpha-1)+1} \\
&= 0.
\end{aligned}$$

□

Lemma 2.1.0.7. *For $\alpha > 1, n\log(\frac{n}{n\cdot(\alpha-1)+1}) \sim n\log(\frac{1}{\alpha-1})$*

Proof.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n\log\left(\frac{n}{n\cdot(\alpha-1)+1}\right)}{n\log\left(\frac{1}{\alpha-1}\right)} &= \lim_{n\to\infty} \frac{\log\left(\frac{n}{n\cdot(\alpha-1)+1}\right)}{\log\left(\frac{1}{\alpha-1}\right)} \\
&= \frac{1}{\log(\frac{1}{\alpha-1})} \cdot \left(\lim_{n\to\infty} \log\left(\frac{n}{n\cdot(\alpha-1)+1}\right)\right) \\
&= \frac{1}{\log(\frac{1}{\alpha-1})} \cdot \left(\log\left(\lim_{n\to\infty} \frac{n}{n\cdot(\alpha-1)+1}\right)\right) \\
&= \frac{1}{\log(\frac{1}{\alpha-1})} \cdot \log(\frac{1}{\alpha-1}) \text{ by L'Hôpital's rule} \\
&= 1.
\end{aligned}$$

□

Theorem 2.1.1. *For $\alpha > 1$, LOHification is in $\Omega\left(n\log(\frac{1}{\alpha-1}) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right)$ complexity.*

Proof. From $n!$ possible unsorted arrays, LOHification produces one of $|L_0|! \cdot |L_1|! \cdots |L_{\ell-1}|!$ possible valid results; and so, using an optimal decision tree, $r(n,\alpha) \in \Omega\left(\log_2\left(\binom{n}{|L_0|,|L_1|,\ldots,|L_{\ell-1}|}\right)\right)$; thus,

$$
\begin{aligned}
r(n,\alpha) &\in \Omega\left(\log\left(\frac{n!}{\prod_{i=0}^{\ell-1}(|L_i|!)}\right)\right) \\
&= \Omega\left(n\log(n) - \sum_{i=0}^{\ell-1}\log(|L_i|!)\right) \\
&= \Omega\left(n\log(n) - \sum_{i=0}^{\ell-1}\log(\lceil\alpha^i\rceil!)\right) \text{ Lemma 2.1.0.4} \\
&= \Omega\left(n\log(n) - \sum_{i=0}^{\ell-1}\lceil\alpha^i\rceil\cdot\log(\lceil\alpha^i\rceil)\right) (\text{since } \log(n!) \in \Theta(n\log(n))) \\
&= \Omega\left(n\log(n) - \sum_{i=0}^{\ell-1}\alpha^i\cdot\log(\alpha^i)\right) \text{ Lemma 2.1.0.5} \\
&= \Omega\left(n\log(n) - \sum_{i=0}^{\ell-1} i\cdot\alpha^i\cdot\log(\alpha)\right) \\
&= \Omega\left(n\log(n) - \log(\alpha)\cdot\sum_{i=0}^{\ell-1} i\cdot\alpha^i\right) \\
&= \Omega\left(n\log(n) - \log(\alpha)\cdot\left(\frac{\alpha^{\ell+1}\cdot(\ell-1)+\alpha-\alpha^\ell\cdot\ell}{(\alpha-1)^2}\right)\right) \\
&= \Omega\,(n\log(n) - \log(\alpha) \\
&\quad\cdot\left(\frac{(\alpha^{\log_\alpha(n\cdot(\alpha-1)+1)+1}\cdot\log_\alpha(n\cdot(\alpha-1)+1)-1)}{(\alpha-1)^2} + \frac{\alpha}{(\alpha-1)^2}\right. \\
&\quad\left.- \frac{\alpha^{\log_\alpha(n\cdot(\alpha-1)+1)}\cdot\log_\alpha(n\cdot(\alpha-1)+1)}{(\alpha-1)^2}\right)\Big) \text{ Lemma 2.1.0.3} \\
&= \Omega\,(n\log(n) - \log(\alpha) \\
&\quad\cdot\left(\frac{(n\cdot(\alpha-1)+1)\cdot\alpha\cdot(\log_\alpha(n\cdot(\alpha-1)+1)-1)}{(\alpha-1)^2} + \frac{\alpha}{(\alpha-1)^2}\right. \\
&\quad\left.- \frac{(n\cdot(\alpha-1)+1)\cdot\log_\alpha(n\cdot(\alpha-1)+1)}{(\alpha-1)^2}\right)\Big)
\end{aligned}
$$

$$
\begin{aligned}
&= \Omega\left(n\log(n) - \left(\frac{(n\cdot(\alpha-1)+1)\cdot\alpha\cdot(\log(n\cdot(\alpha-1)+1)-\log(\alpha))}{(\alpha-1)^2}\right.\right.\\
&\quad \left.\left.+\frac{\alpha\log(\alpha)}{(\alpha-1)^2} - \frac{(n\cdot(\alpha-1)+1)\log(n\cdot(\alpha-1)+1)}{(\alpha-1)^2}\right)\right)\\
&= \Omega\left(n\log(n) - \left(\frac{(n\cdot(\alpha-1)+1)\cdot\alpha\log(n\cdot(\alpha-1)+1)}{(\alpha-1)^2}\right.\right.\\
&\quad - \frac{(n\cdot(\alpha-1)+1)\cdot\alpha\log(\alpha)}{(\alpha-1)^2} + \frac{\alpha\log(\alpha)}{(\alpha-1)^2}\\
&\quad \left.\left.- \frac{(n\cdot(\alpha-1)+1)\cdot\log(n\cdot(\alpha-1)+1)}{(\alpha-1)^2}\right)\right)\\
&= \Omega\left(n\log(n) - \left(\frac{(n\cdot(\alpha-1)+1)\cdot\log(n\cdot(\alpha-1)+1)\cdot(\alpha-1)}{(\alpha-1)^2}\right.\right.\\
&\quad \left.\left.+\frac{\alpha\log(\alpha)}{(\alpha-1)^2} - \frac{(n\cdot(\alpha-1)+1)\cdot\alpha\log(\alpha)}{(\alpha-1)^2}\right)\right)\\
&= \Omega\left(n\log(n) - \frac{(n\cdot(\alpha-1)+1)\cdot\log(n\cdot(\alpha-1)+1)}{\alpha-1} - \frac{\alpha\log(\alpha)}{(\alpha-1)^2}\right.\\
&\quad \left.+\frac{(n\cdot(\alpha-1)+1)\cdot\alpha\log(\alpha)}{(\alpha-1)^2}\right)\\
&= \Omega\left(n\log(n) - n\cdot\log(n\cdot(\alpha-1)+1) - \frac{\log(n\cdot(\alpha-1)+1)}{\alpha-1}\right.\\
&\quad \left.-\frac{\alpha\log(\alpha)}{(\alpha-1)^2} + \frac{n\cdot\alpha\log(\alpha)}{\alpha-1} + \frac{\alpha\log(\alpha)}{(\alpha-1)^2}\right)\\
&= \Omega\left(n\log(n) - n\cdot\log(n\cdot(\alpha-1)+1) - \frac{\log(n\cdot(\alpha-1)+1)}{\alpha-1}\right.\\
&\quad \left.+\frac{n\cdot\alpha\log(\alpha)}{\alpha-1}\right)\\
&= \Omega\left(n\log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) - \frac{\log(n\cdot(\alpha-1)+1)}{\alpha-1} + \frac{n\cdot\alpha\log(\alpha)}{\alpha-1}\right)\\
&\subseteq \Omega\left(n\log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) + \frac{n\cdot\alpha\log(\alpha)}{\alpha-1}\right) \text{ Lemma 2.1.0.6}\\
&= \Omega\left(n\log\left(\frac{1}{\alpha-1}\right) + \frac{n\cdot\alpha\log(\alpha)}{\alpha-1}\right) \text{ Lemma 2.1.0.7.}
\end{aligned}
$$

□

It should be noted that the formulas with $\alpha = 1$ behave as expected. We have now established LOHification to be in $\Omega(n\log(\frac{1}{\alpha-1}) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1})$ for any $\alpha > 1$. Because

the $\log\left(\frac{1}{\alpha-1}\right)$ term becomes zero at $\alpha = 2$, we shall explore LOHification with $\alpha = 1$, $1 < \alpha < 2$, and $\alpha \geq 2$.

Theorem 2.1.2. *For $\alpha \in (1, 2)$, LOHification is in $\Omega(n \log(\frac{\alpha}{\alpha-1}))$ complexity.*

Proof. From our proof in Theorem 2.1.1, we know that LOHification with $\alpha > 1$ is in $\Omega\left(n \log(\frac{1}{\alpha-1}) + \frac{n \cdot \alpha \cdot \log(\alpha)}{\alpha-1}\right)$. Thus, for our interval, we have:

$$\begin{aligned}
r(n,\alpha) &\in \Omega\left(n \log\left(\frac{1}{\alpha-1}\right) + \frac{n \cdot \alpha \log(\alpha)}{\alpha-1}\right) \\
&\in \Omega\left(n \log\left(\frac{1}{\alpha-1}\right) + n \log(\alpha) \cdot \frac{\alpha}{\alpha-1}\right) \\
&\quad \text{because } \frac{\alpha}{\alpha-1} \text{ is trivially bounded below by 1 on the interval } (1,2), \\
&\subseteq \Omega\left(n \log\left(\frac{1}{\alpha-1}\right) + n \log(\alpha)\right) \\
&\in \Omega\left(n \log\left(\frac{\alpha}{\alpha-1}\right)\right).
\end{aligned}$$

□

In the following sections, we will explore different algorithms for LOHification, their complexity, and for what values of α they are optimal.

2.1.2 Algorithms for LOHification

2.1.2.1 LOHification via sorting

Sorting, which can be done in $O(n \log(n))$, trivially LOHifies an array. Hence, LOHification is in $O(n \log(n))$ (we will later show a tighter bound).

When sorting is optimal

If $\alpha = 1$, each layer has $|L_i| = 1$, meaning an ordering over all elements; so sorting must be performed. Thus, for $\alpha = 1$, sorting is optimal. Furthermore, we can find an α^* where sorting is optimal for all $\alpha \leq \alpha^*$. Doing this, we find that, for any constant,

$C > 0$, sorting is optimal for $\alpha \leq 1 + \frac{C}{n}$. It should be noted that, as with the previous derivations, α is a free parameter. This section aims to highlight possible pitfalls that may arise from making α a function of n.

To prove this, we will use the following lemma.

Lemma 2.1.0.8. *For any constant, $C > 0$, $(n^2 + n) \cdot \log(1 + \frac{C}{n}) \in o(n \cdot \log(n))$.*

Proof.

$$
\begin{aligned}
& \lim_{n\to\infty} \frac{(n^2+n)\cdot\log(1+\frac{C}{n})}{n\cdot\log(n)} \\
= & \lim_{n\to\infty} \frac{(n+1)\cdot\log(1+\frac{C}{n})}{\log(n)} \\
= & \lim_{n\to\infty} \frac{n\cdot\log(1+\frac{C}{n})}{\log(n)} + \frac{\log(1+\frac{C}{n})}{\log(n)} \\
= & \lim_{n\to\infty} \frac{n\cdot\log(1+\frac{C}{n})}{\log(n)} \\
= & \lim_{n\to\infty} \frac{\log(1+\frac{C}{n}) - \frac{C}{n+C}}{(\frac{C}{n})} \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \frac{\log(1+\frac{C}{n})}{(\frac{C}{n})} - \frac{n}{n+C} \\
= & \lim_{n\to\infty} \frac{-(\frac{C}{n^2+C\cdot n})}{-(\frac{C}{n^2})} - 1 \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \frac{n^2}{n^2+C\cdot n} - 1 \\
= & \lim_{n\to\infty} \frac{n}{n+C} - 1 \text{ by L'Hôpital's rule} \\
= & \ 1 - 1 \text{ by L'Hôpital's rule} \\
= & \ 0.
\end{aligned}
$$

□

Theorem 2.1.3. *For any constant, $C > 0$, sorting is optimal for $\alpha \leq \left(1 + \frac{C}{n}\right) := \alpha^*$.*

Proof. Because decreasing α can only increase the number of layers, therefore the number of operations needed to LOHify, its suffices to show that sorting is optimal at $\alpha^* = \left(1 + \frac{C}{n}\right)$.

$$\begin{aligned}
\mathrm{r}(n, \alpha^*) &\in \Omega\left(n \log\left(\frac{1}{\alpha^* - 1}\right) + \frac{n \cdot \alpha^* \cdot \log(\alpha^*)}{\alpha^* - 1}\right) \\
&= \Omega\left(n \log\left(\frac{1}{\left(1 + \frac{C}{n}\right) - 1}\right) + \frac{n \cdot \left(1 + \frac{C}{n}\right) \cdot \log(\left(1 + \frac{C}{n}\right))}{\left(1 + \frac{C}{n}\right) - 1}\right) \\
&= \Omega\left(n \log\left(\frac{n}{C}\right) + \frac{(n + C) \cdot \log\left(1 + \frac{C}{n}\right)}{\frac{C}{n}}\right) \\
&= \Omega\left(n \log(n) - n \log(C) + \frac{(n^2 + C \cdot n) \cdot \log\left(1 + \frac{C}{n}\right)}{C}\right) \\
&= \Omega\left(n \cdot \log(n) + (n^2 + n) \cdot \log\left(1 + \frac{C}{n}\right)\right) \\
&= \Omega(n \cdot \log(n) + o(n \cdot \log(n))) \text{ Lemma 2.1.0.8} \\
&\subseteq \Omega(n \cdot \log(n)) \\
&\quad \text{Therefore,} \\
\text{LOHification} &\in \Theta(n \cdot \log(n)) \text{ for } \alpha \leq \left(1 + \frac{C}{n}\right).
\end{aligned}$$

□

Because sorting is optimal for these values of α, we know that, for all α at most $\alpha^* = \left(1 + \frac{C}{n}\right)$, LOHification is in $\Theta\left(n \log(\frac{n}{n \cdot (\alpha - 1) + 1}) + \frac{n \cdot \alpha \cdot \log(\alpha)}{\alpha - 1}\right)$. Next we will look at LOHification methods that are based on selection.

2.1.2.2 LOHification via iterative selection

LOHs can be constructed using one-dimensional selection, which can be done in linear time via median-of-medians [11]. The median-of-medians algorithm produces

the k^{th} smallest element in an array of length n in $O(n)$ time. From this we can partition the array on that element in linear time for a linear time top-k on an array.

In this section, we will describe LOHification algorithms that select away layers from the ends of the array, prove their complexity, and find the values of α for which they are optimal.

Selecting away the layer with the greatest index

This algorithm repeatedly performs a linear-time one-dimensional selection on the value at the first index (were the array in sorted order) in our last layer, $L_{\ell-1}$. Then, the array is partitioned about this value. This is repeated for $L_{\ell-2}$, $L_{\ell-3}$, and so on until the array has been partitioned about the minimum value in each layer thus LOHifying the array. We will prove that this algorithm is in $\Theta\left(\frac{\alpha \cdot n}{\alpha-1}\right)$ using the following lemmas.

Lemma 2.1.0.9. *For $\alpha > 1$, $\left(\frac{(\log_\alpha(n\cdot(\alpha-1)+1))^2-\log_\alpha(n\cdot(\alpha-1))}{2}\right) \in o\left(\frac{\alpha\cdot n}{\alpha-1}\right)$.*

Proof.

$$
\begin{aligned}
& \lim_{n\to\infty} \frac{\left(\frac{(\log_\alpha(n\cdot(\alpha-1)+1))^2-\log_\alpha(n\cdot(\alpha-1))}{2}\right)}{\left(\frac{\alpha\cdot n}{\alpha-1}\right)} \\
= & \lim_{n\to\infty} \frac{\frac{\alpha-1}{(\log(\alpha))^2}\cdot(\log(n\cdot(\alpha-1)+1))^2 - \frac{\alpha-1}{\log(\alpha)}\cdot\log(n\cdot(\alpha-1)+1)}{2\cdot\alpha\cdot n} \\
= & \lim_{n\to\infty} \frac{\frac{2\cdot(\alpha-1)^2}{(\log(\alpha))^2}\cdot\frac{\log(n\cdot(\alpha-1)+1)}{n\cdot(\alpha-1)+1} - \frac{(\alpha-1)^2}{\log(\alpha)}\cdot\frac{1}{n\cdot(\alpha-1)+1}}{2\cdot\alpha} \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \frac{\frac{2\cdot(\alpha-1)^2}{(\log(\alpha))^2}\cdot\frac{\log(n\cdot(\alpha-1)+1)}{n\cdot(\alpha-1)+1} - \cancel{\frac{(\alpha-1)^2}{\log(\alpha)}\cdot\frac{1}{n\cdot(\alpha-1)+1}}^{\to 0}}{2\cdot\alpha} \\
= & \lim_{n\to\infty} \frac{(\alpha-1)^2}{\alpha\cdot(\log(\alpha))^2}\cdot\frac{\log(n\cdot(\alpha-1)+1)}{n\cdot(\alpha-1)+1} \\
= & \frac{(\alpha-1)^2}{\alpha\cdot(\log(\alpha))^2}\cdot\left(\lim_{n\to\infty}\frac{\log(n\cdot(\alpha-1)+1)}{n\cdot(\alpha-1)+1}\right) \\
= & \frac{(\alpha-1)^2}{\alpha\cdot(\log(\alpha))^2}\cdot\left(\lim_{n\to\infty}\frac{1}{n\cdot(\alpha-1)+1}\right) \text{ by L'Hôpital's rule}
\end{aligned}
$$

$$= \quad 0.$$

□

Lemma 2.1.0.10. *For $\alpha > 1$, $\left(\frac{\log_\alpha(n\cdot(\alpha-1))}{\alpha-1}\right) \in o\left(\frac{\alpha\cdot n}{\alpha-1}\right)$.*

Proof.

$$\begin{aligned}
& \lim_{n\to\infty} \frac{\left(\frac{\log_\alpha(n\cdot(\alpha-1))}{\alpha-1}\right)}{\left(\frac{\alpha\cdot n}{\alpha-1}\right)} \\
= & \lim_{n\to\infty} \frac{\log_\alpha(n\cdot(\alpha-1))}{\alpha\cdot n} \\
= & \lim_{n\to\infty} \frac{1}{\alpha\cdot\log(\alpha)} \cdot \frac{\log(n\cdot(\alpha-1))}{n} \\
= & \frac{1}{\alpha\cdot\log(\alpha)} \cdot \left(\lim_{n\to\infty} \frac{\log(n\cdot(\alpha-1))}{n}\right) \\
= & \frac{1}{\alpha\cdot\log(\alpha)} \cdot \left(\lim_{n\to\infty} \frac{\alpha}{n\cdot(\alpha-1)}\right) \text{ by L'Hôpital's rule} \\
= & \quad 0.
\end{aligned}$$

□

Lemma 2.1.0.11. *Selecting away the layer with the greatest index is in $\Omega\left(\frac{\alpha\cdot n}{\alpha-1}\right)$.*

Proof. By using a linear time one-dimensional selection, we can see that the runtime for selecting away the layer with the greatest index is:

$$\begin{aligned}
r(n,\alpha) \quad \in \quad & \Theta\left(n + (n - |L_{\ell-1}|) + \cdots + \left(n - \sum_{j<\ell-1} |L_{\ell-j-1}|\right)\right) \\
= \quad & \Theta\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=\ell-i}^{\ell-1} |L_j|\right)\right) \\
\subseteq \quad & \Omega\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=\ell-i}^{\ell-1} \lceil\alpha^j\rceil\right)\right)
\end{aligned}$$

$$
\begin{aligned}
&\subseteq \Omega\left(\sum_{i=0}^{\ell-1}\left(n-\sum_{j=\ell-i}^{\ell-1}(\alpha^j+1)\right)\right)\\
&= \Omega\left(\sum_{i=0}^{\ell-1}\left(n-i-\sum_{j=\ell-i}^{\ell-1}\alpha^j\right)\right)\\
&= \Omega\left(n\cdot\ell-\frac{\ell^2-\ell}{2}-\frac{1}{\alpha-1}\cdot\left(\sum_{i=0}^{\ell-1}(\alpha^\ell-\alpha^{\ell-i})\right)\right)\\
&= \Omega\left(n\cdot\ell-\frac{\ell^2-\ell}{2}-\frac{1}{\alpha-1}\cdot\frac{(\ell-1)\alpha^{\ell+1}-\ell\cdot\alpha^\ell+\alpha}{\alpha-1}\right)\\
&= \Omega\left(n\cdot\ell-\frac{\ell^2-\ell}{2}-\frac{1}{\alpha-1}\cdot\frac{(\alpha-1)\cdot\ell\cdot\alpha^\ell-\alpha\cdot(\alpha^\ell-1)}{\alpha-1}\right)\\
&= \Omega\left(n\cdot\ell-\frac{\ell^2-\ell}{2}-\frac{\ell\cdot\alpha^\ell}{\alpha-1}+\frac{\alpha\cdot(\alpha^\ell-1)}{(\alpha-1)^2}\right)\\
&= \Omega\left(\ell\cdot\left(n-\frac{\ell-1}{2}-\frac{\alpha^\ell}{\alpha-1}\right)+\frac{\alpha\cdot(\alpha^\ell-1)}{(\alpha-1)^2}\right)\\
&\subseteq \Omega\left(\ell\cdot\left(n-\frac{\ell-1}{2}-\frac{n\cdot(\alpha-1)+1}{\alpha-1}\right)+\frac{\alpha\cdot n\cdot(\alpha-1)}{(\alpha-1)^2}\right) \text{ Lemma 2.1.0.2}\\
&= \Omega\left(\ell\cdot\left(-\frac{\ell-1}{2}-\frac{1}{\alpha-1}\right)+\frac{\alpha\cdot n}{\alpha-1}\right)\\
&= \Omega\left(\frac{\alpha\cdot n}{\alpha-1}-\left(\frac{\ell^2-\ell}{2}+\frac{\ell}{\alpha-1}\right)\right)\\
&\subseteq \Omega\left(\frac{\alpha\cdot n}{\alpha-1}-\left(\frac{(\log_\alpha(n\cdot(\alpha-1)+1))^2-\log_\alpha(n\cdot(\alpha-1))}{2}\right)\right.\\
&\quad \left.-\left(\frac{\log_\alpha(n\cdot(\alpha-1))}{\alpha-1}\right)\right) \text{ Lemma 2.1.0.1 and Lemma 2.1.0.2}\\
&\subseteq \Omega\left(\frac{\alpha\cdot n}{\alpha-1}\right) \text{ Lemma 2.1.0.9 and Lemma 2.1.0.10}
\end{aligned}
$$

hence;

$$
r(n,\alpha) \in \Omega\left(\frac{\alpha\cdot n}{\alpha-1}\right).
$$

□

Theorem 2.1.4. *Selecting away the layer with the greatest index is in* $\Theta\left(\frac{\alpha\cdot n}{\alpha-1}\right)$.

Proof. Using a linear time one-dimensional selection, we can see that the runtime for

selecting away the layer with the greatest index is:

$$\begin{aligned}
r(n,\alpha) &\in \Theta\left(n + (n - |L_{\ell-1}|) + \cdots + \left(n - \sum_{j<\ell-1} |L_{\ell-j-1}|\right)\right) \\
&= \Theta\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=\ell-i}^{\ell-1} |L_j|\right)\right) \\
&\subseteq O\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=\ell-i}^{\ell-1} \alpha^j\right)\right) \\
&= O\left(n\cdot\ell - \frac{1}{\alpha-1}\cdot\left(\sum_{i=0}^{\ell-1}(\alpha^\ell - \alpha^{\ell-i})\right)\right) \\
&= O\left(n\cdot\ell - \frac{1}{\alpha-1}\cdot\frac{(\ell-1)\alpha^{\ell+1} - \ell\cdot\alpha^\ell + \alpha}{\alpha-1}\right) \\
&= O\left(n\cdot\ell - \frac{1}{\alpha-1}\cdot\frac{(\alpha-1)\cdot\ell\cdot\alpha^\ell - \alpha\cdot(\alpha^\ell-1)}{\alpha-1}\right) \\
&= O\left(n\cdot\ell - \frac{\ell\cdot\alpha^\ell}{\alpha-1} + \frac{\alpha\cdot(\alpha^\ell-1)}{(\alpha-1)^2}\right) \\
&= O\left(\ell\cdot\left(n - \frac{\alpha^\ell}{\alpha-1}\right) + \frac{\alpha\cdot(\alpha^\ell-1)}{(\alpha-1)^2}\right) \\
&\subseteq O\left(\ell\cdot\left(n - \frac{n\cdot(\alpha-1)+1}{\alpha-1}\right) + \frac{\alpha\cdot n\cdot(\alpha-1)}{(\alpha-1)^2}\right) \text{ Lemma 2.1.0.2} \\
&= O\left(\ell\cdot\left(-\frac{1}{\alpha-1}\right) + \frac{\alpha\cdot n}{\alpha-1}\right) \\
&= O\left(\frac{\alpha\cdot n}{\alpha-1} - \left(\frac{\ell}{\alpha-1}\right)\right) \\
&\subseteq O\left(\frac{\alpha\cdot n}{\alpha-1} - \left(\frac{\log_\alpha(n\cdot(\alpha-1))}{\alpha-1}\right)\right) \text{ Lemma 2.1.0.2} \\
&\subseteq O\left(\frac{\alpha\cdot n}{\alpha-1}\right) \text{ Lemma 2.1.0.10}
\end{aligned}$$

hence;

$$r(n,\alpha) \in O\left(\frac{\alpha\cdot n}{\alpha-1}\right);$$

therefore, by Lemma 2.1.0.11;

$$r(n,\alpha) \in \Theta\left(\frac{\alpha\cdot n}{\alpha-1}\right).$$

□

Selecting away the layer with the least index

We can also select from the other side. We perform this algorithm by performing a linear-time one-dimensional selection to select L_0, then, from the remaining layers, select L_1 and so forth until the array is LOHified. It is simple to see via the following proof that this method will never be better than selecting away the layer with the largest index.

Theorem 2.1.5. *Selecting away the layer with the least index is in* $\Omega\left(\frac{\alpha \cdot n}{\alpha - 1}\right)$.

Proof. Using a linear time one-dimensional selection, we can see that the runtime for selecting away the layer with the least index is:

$$\begin{aligned}
r(n,\alpha) &\in \Theta\left(n + (n - |L_0|) + \cdots + \left(n - \sum_{j<\ell-1} |L_j|\right)\right) \\
&= \Theta\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=0}^{i-1} |L_j|\right)\right) \\
&\subseteq \Omega\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=0}^{i-1} \lceil \alpha^j \rceil\right)\right) \\
&\subseteq \Omega\left(\sum_{i=0}^{\ell-1}\left(n - \sum_{j=0}^{i-1} (\alpha^j + 1)\right)\right) \\
&= \Omega\left(\sum_{i=0}^{\ell-1}\left(n - i - \sum_{j=0}^{i-1} \alpha^j\right)\right) \\
&\subseteq \Omega\left(\sum_{i=0}^{\ell-1}\left(n - i - \sum_{j=\ell-i}^{\ell-1} \alpha^j\right)\right) \\
&\subseteq \Omega\left(\frac{\alpha \cdot n}{\alpha - 1}\right), \text{ Theorem 2.1.4} \\
&\quad \text{hence;} \\
r(n,\alpha) &\in \Omega\left(\frac{\alpha \cdot n}{\alpha - 1}\right).
\end{aligned}$$

□

When iterative selection is optimal

Because selecting away the layer with the greatest index is never worse than selecting away the layer with the least index, we shall assume the iterative selection is selecting away the layer with the greatest index. Again, we shall also assume that $\alpha > 1$ as sorting is optimal for $\alpha = 1$. We will prove that this method is optimal in n for all values of $\alpha \geq C$ for any fixed constant $C > 1$, but not in α for $\alpha \in (1, 2)$.

This is similar to the notion that any terminating algorithm on a problem whose size is bounded by a constant is done in constant time. We show this to highlight some pitfalls that occur when α is treated like a constant instead of a parameter.

Theorem 2.1.6. *Given any constant: $C > 1$, iterative selection is optimal (in n) for all $\alpha \geq C$.*

Proof. LOHification is trivially in $\Omega(n)$, as that is the cost to load the data. As α increases, the number of layers (and hence, the work) can only decrease, thus it suffices to show iterative selection is optimal at $\alpha = C$.

$$\begin{aligned} r(n, C) &\in O\left(\frac{C \cdot n}{C - 1}\right) \text{ Theorem 2.1.4} \\ &\in O(n) \\ &\text{therefore;} \\ \text{LOHification} &\in \Theta(n) \text{ for all } \alpha \geq \text{some fixed constant } C > 1. \end{aligned}$$

□

Lemma 2.1.0.12. *Iterative selection is sub-optimal (in n) for $\alpha = \alpha^* = 1 + \frac{C}{n}$ for any constant $C > 0$.*

Proof.

$$\begin{aligned} r(n,\alpha^*) &\in \Theta\left(\frac{\alpha^* \cdot n}{\alpha^* - 1}\right) \text{ Theorem 2.1.4} \\ &= \Theta\left(\frac{\left(1+\frac{C}{n}\right)\cdot n}{\left(1+\frac{C}{n}\right)-1}\right) \\ &= \Theta\left(\frac{n+C}{\left(\frac{C}{n}\right)}\right) \\ &= \Theta\left(\frac{n^2+C\cdot n}{C}\right) \\ &\subseteq \Theta(n^2). \end{aligned}$$

This does not achieve our lower bound. □

Theorem 2.1.7. *Iterative selection does not achieve the lower bound for* $\alpha \in (1,2)$ *that was demonstrated in Theorem* 2.1.2.

Proof.

$$\begin{aligned} r(n,\alpha) &\in \Theta\left(\frac{\alpha \cdot n}{\alpha - 1}\right), \text{ Theorem 2.1.4} \\ &\in \Theta\left(n \cdot \frac{\alpha}{\alpha - 1}\right) \\ &\in \omega\left(n \log\left(\frac{\alpha}{\alpha - 1}\right)\right). \end{aligned}$$

□

We will later show in Lemma 2.1.0.19 that the lower bound for $\alpha \in (1,2)$ demonstrated in Theorem 2.1.2 can be achieved, thus making iterative selection sub-optimal.

2.1.2.3 Selecting to divide remaining pivot indices in half

For this algorithm, we first calculate the pivot indices in $O(n)$. Then, we perform a linear-time one-dimensional selection for the median pivot, partition on that pivot, and then recurse on the sub-problems until the array is LOHified.

Runtime

One-dimensional selection is in $\Theta(n)$, thus the cost of every layer in the recursion is in $\Theta(n)$. Because splitting at the median pivot creates a balanced-binary recursion tree, the cost of the algorithm is in $\Theta(n \cdot d)$ where d is the depth of the recursion tree. The number of pivots in each recursive call is one less than half of the number of pivots in the parent call, so we have $d = \log_2(\ell)$. Hence:

$$\begin{aligned} r(n, \alpha) &\in \Theta(n \cdot \log(\ell)) \\ &= \Theta(n \cdot \log(\log_\alpha(n \cdot (\alpha - 1) + 1))) \\ &= \Theta\left(n \cdot \log\left(\frac{\log(n \cdot (\alpha - 1) + 1)}{\log(\alpha)}\right)\right). \end{aligned}$$

When selecting to divide remaining pivot indices in half is optimal

Here we will show that this method is optimal for the values of α where sorting is optimal: $1 \leq \alpha \leq \alpha^* = 1 + \frac{C}{n}$ for any constant, $C > 0$. We will then show that it is sub-optimal for $\alpha =$ some fixed constant $C > 1$.

Lemma 2.1.0.13. $n \cdot \log\left(\frac{\log(C)}{\log\left(1+\frac{C}{n}\right)}\right) \in \Theta(n \cdot \log(n))$ *for any constant,* $C > 0$.

Proof.

$$\lim_{n \to \infty} \frac{n \cdot \log\left(\frac{\log(C)}{\log\left(1+\frac{C}{n}\right)}\right)}{n \cdot \log(n)}$$

$$
\begin{aligned}
&= \lim_{n\to\infty} \frac{\log\left(\frac{\log(C)}{\log\left(1+\frac{C}{n}\right)}\right)}{\log(n)} \\
&= \lim_{n\to\infty} \frac{\left(\frac{C}{n\cdot(n+C)\cdot\log\left(1+\frac{C}{n}\right)}\right)}{\left(\frac{1}{n}\right)} \text{ by L'Hôpital's rule} \\
&= \lim_{n\to\infty} \frac{1}{(n+C)\cdot\log\left(1+\frac{C}{n}\right)} \\
&= \lim_{n\to\infty} \frac{\left(\frac{1}{n+C}\right)}{\log\left(1+\frac{C}{n}\right)} \\
&= \lim_{n\to\infty} \frac{\left(\frac{-1}{n^2+2\cdot C\cdot n+C^2}\right)}{\left(\frac{-C}{n^2+C\cdot n}\right)} \text{ by L'Hôpital's rule} \\
&= \lim_{n\to\infty} \frac{n^2+C\cdot n}{C\cdot n^2+2\cdot C^2\cdot n+C^3} \\
&= \frac{1}{C} \text{ by L'Hôpital's rule.}
\end{aligned}
$$

□

Lemma 2.1.0.14. *Selecting to divide remaining pivot indices in half is optimal for* $\alpha = \alpha^* = 1 + \frac{C}{n}$ *for any constant,* $C > 0$.

Proof.

$$
\begin{aligned}
r(n,\alpha^*) &\in \Theta\left(n\cdot\log\left(\frac{\log(n\cdot(\alpha^*-1)+1)}{\log(\alpha^*)}\right)\right) \\
&= \Theta\left(n\cdot\log\left(\frac{\log\left(n\cdot\left(\left(1+\frac{C}{n}\right)-1\right)+1\right)}{\log\left(1+\frac{C}{n}\right)}\right)\right) \\
&= \Theta\left(n\cdot\log\left(\frac{\log(C))}{\log\left(1+\frac{C}{n}\right)}\right)\right) \\
&= \Theta(n\cdot\log(n)) \text{ Lemma 2.1.0.13.}
\end{aligned}
$$

□

Lemma 2.1.0.15. *Selecting to divide remaining pivot indices in half is sub-optimal for $\alpha =$ some fixed constant $C > 1$.*

Proof. Because Theorem 2.1.6 shows that LOHification can be done in linear time under these conditions, it only remains to show that this method is $\in \omega(n)$.

$$\begin{aligned} r(n, C) &\in \Theta\left(n \cdot \log\left(\frac{\log(n \cdot (C-1)+1)}{\log(C)}\right)\right) \\ &= \Theta(n \cdot \log(\log(n))) \\ &\subseteq \omega(n). \end{aligned}$$

□

2.1.2.4 Partitioning on the pivot closest to the center of the array

For this algorithm, we start by computing the pivots and then performing a linear-time selection algorithm on the pivot closest to the true median of the array to partition the array into two parts. We then recurse on the parts until all layers are generated. In this section, we will describe the runtime recurrence in detail, and then prove that this method has optimal performance at any α.

The Runtime Recurrence

Let n_s be the starting index of our (sub)array. Let n_e be the ending index or our (sub)array. Let $m(n_s, n_e, \alpha)$ be the number of pivots between n_s and n_e (exclusive). Let $x(n_s, n_e, \alpha)$ be the index of the pivot closest to the middle of the (sub)array starting at n_s and ending at n_e. Then the runtime of our algorithm, $r(n, \alpha)$, is equal to $s(0, n, \alpha)$ where

$$s(n_s, n_e, \alpha) = \begin{cases} 0, & n_s \geq n_e \\ 0, & m(n_s, n_e, \alpha) = 0 \\ n_e - n_s + r(n_s, x(n_s, n_e, \alpha) - 1, \alpha) + r(x(n_s, n_e, \alpha) + 1, n_e, \alpha), & \text{else} \end{cases}$$

Describing the asymptotic bounds of a recurrence has been an area of interest in computer science for a long time and many methods have been found to calculate these bounds. Due to the fact that the recursive calls differ in both size and number of pivots, however, the recurrence for this algorithm does not fit the form of the 'Master Theorem' [12], nor can it be solved with the more general Akra-Bazzi method [13]. Instead, we will bound the recursion tree by bounding how far right we go in the recursion tree, t_{max}, and use this to find the deepest layer, d^* for which all branches have work. Because performing two selections is $\in O(n)$, we will bound the size of the recursions by half of the parent by selecting on the pivots on both sides of the true median (if the true median is a pivot we just pay for it twice). From there, the bound on the runtime can be computed as $O(d^* \cdot n) + O\left(\sum_{d=d^*}^{\log(n)} \sum_{t=1}^{t_{max}} \frac{n}{2^d}\right)$. This scheme is depicted in Fig 2.1.

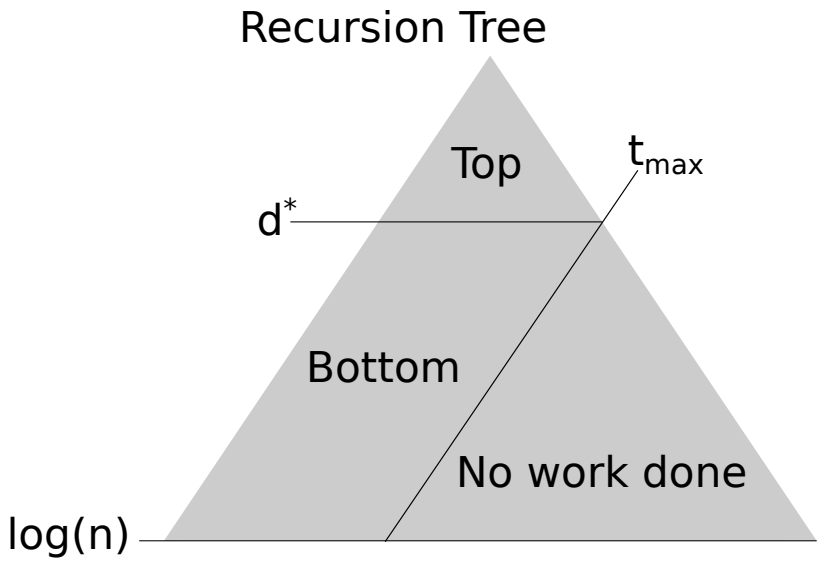

Figure 2.1 **The recursion tree for partitioning on the pivot closest to the center of the array.** The work at "Top" is $\in O(n \cdot d^*)$ and the work done at "Bottom" is $\in O\left(\sum_{d=d^*}^{\log(n)} \sum_{t=1}^{t_{max}} \frac{n}{2^d}\right)$. d^* is the greatest depth at which all branches have work and t_{max} is a bound on the "furthest right" we go in the recursion tree.

Bounds on variables

Lemma 2.1.0.16. *For $\alpha > 1$, the number of pivots between any two points is* $m(n_s, n_e, \alpha) \leq \log_\alpha \left(\frac{n_e \cdot (\alpha-1)+1}{(n_s-1)\cdot(\alpha-1)+1}\right)$.

Proof. By our definition, the i^{th} pivot, p_i, occurs at $p_i = \left\lceil \sum_{j=0}^{i} \alpha^j \right\rceil = \left\lceil \frac{\alpha^{i+1}-1}{\alpha-1} \right\rceil$. Let n_s be the start of our (sub)array and n_e be the end of our (sub)array. Then the number of pivots, p_e, occurring before n_e is bound by the inequality:

$$n_e \geq \left\lceil \frac{\alpha^{p_e+1}-1}{\alpha-1} \right\rceil$$

$$\begin{aligned} n_e &\geq \frac{\alpha^{p_e+1}-1}{\alpha-1} \\ n_e \cdot (\alpha-1) &\geq \alpha^{p_e+1}-1 \\ n_e \cdot (\alpha-1)+1 &\geq \alpha^{p_e+1} \\ \log_\alpha(n_e \cdot (\alpha-1)+1) &\geq p_e+1 \\ \log_\alpha(n_e \cdot (\alpha-1)+1)-1 &\geq p_e. \end{aligned}$$

Similarly, the number of pivots, p_s, occurring before n_s is bound by the inequality:

$$\begin{aligned} n_s &\leq \left\lceil \frac{\alpha^{p_s+1}-1}{\alpha-1} \right\rceil \\ n_s &\leq \frac{\alpha^{p_s+1}-1}{\alpha-1}+1 \\ n_s-1 &\leq \frac{\alpha^{p_s+1}-1}{\alpha-1} \\ (n_s-1) \cdot (\alpha-1) &\leq \alpha^{p_s+1}-1 \\ (n_s-1) \cdot (\alpha-1)+1 &\leq \alpha^{p_s+1} \\ \log_\alpha((n_s-1) \cdot (\alpha-1)+1) &\leq p_s+1 \\ \log_\alpha((n_s-1) \cdot (\alpha-1)+1)-1 &\leq p_s. \end{aligned}$$

By combining these two inequalities, we can find an upper bound on the number of pivots in the (sub)array, $m(n_s, n_e, \alpha)$:

$$\begin{aligned} m(n_s, n_e, \alpha) &\leq (\log_\alpha(n_e \cdot (\alpha-1)+1)-1)-(\log_\alpha((n_s-1) \cdot (\alpha-1)+1)-1) \\ &\leq \log_\alpha(n_e \cdot (\alpha-1)+1)-\log_\alpha((n_s-1) \cdot (\alpha-1)+1) \\ &\leq \log_\alpha\left(\frac{n_e \cdot (\alpha-1)+1}{(n_s-1) \cdot (\alpha-1)+1}\right). \end{aligned}$$

□

A bound on the runtime recurrence

For the following bounds, we will assume that $\alpha > 1$. Let d be the depth of our current recursion (indexed at 0) and t be how far right in the tree we are at our current recursion (indexed at 1). To get an upper bound on the recurrence, we will compute the cost of selecting for both the first index before the true middle and the first index after the true median. We will then treat the true middle as $x(n_s, n_e, \alpha)$ for our recursive calls. Under these restrictions, $n_s = \frac{n\cdot(t-1)}{2^d}$ and $n_e = \frac{n\cdot t}{2^d}$ for a given t and d. Knowing this, we can calculate bounds for $m(n_s, n_e, \alpha)$ in terms of t and d.

Lemma 2.1.0.17. *For $\alpha > 1$, $m(n_s, n_e, \alpha) \leq \log_\alpha \left(\frac{n\cdot t\cdot(\alpha-1)+2^d}{(n\cdot(t-1)-2^d)\cdot(\alpha-1)+2^d}\right)$.*

Proof.

$$\begin{aligned} m(n_s, n_e, \alpha) &\leq \log_\alpha \left(\frac{n_e\cdot(\alpha-1)+1}{(n_s-1)\cdot(\alpha-1)+1}\right) \text{ Lemma 2.1.0.16} \\ &\leq \log_\alpha \left(\frac{\frac{n\cdot t}{2^d}\cdot(\alpha-1)+1}{(\frac{n\cdot(t-1)}{2^d}-1)\cdot(\alpha-1)+1}\right) \\ &\leq \log_\alpha \left(\frac{n\cdot t\cdot(\alpha-1)+2^d}{(n\cdot(t-1)-2^d)\cdot(\alpha-1)+2^d}\right). \end{aligned}$$

□

We can then use this to calculate t, in terms of α, n and d for which $m(n_s, n_e, \alpha) < 1$. This will give us a bound on how far right we go in the recursion tree.

Lemma 2.1.0.18. $t_{max} = \frac{\alpha}{\alpha-1} + 1$.

Proof. By Lemma 2.1.0.17, we have $m(n_s, n_e, \alpha) \leq \log_\alpha \left(\frac{n\cdot t\cdot(\alpha-1)+2^d}{(n\cdot(t-1)-2^d)\cdot(\alpha-1)+2^d}\right)$. Thus, we need only find the least value of t for which this expression is less than 1.

$$\log_\alpha \left(\frac{n\cdot t\cdot(\alpha-1)+2^d}{(n\cdot(t-1)-2^d)\cdot(\alpha-1)+2^d}\right) \quad < \quad 1$$

$$\begin{aligned}
\left(\frac{n\cdot t\cdot(\alpha-1)+2^d}{(n\cdot(t-1)-2^d)\cdot(\alpha-1)+2^d}\right) &< \alpha \\
n\cdot t\cdot(\alpha-1)+2^d &< \alpha\cdot(n\cdot(t-1)-2^d)\cdot(\alpha-1)+\alpha\cdot 2^d \\
n\cdot t\cdot(\alpha-1) &< \alpha\cdot(n\cdot(t-1)-2^d)\cdot(\alpha-1)+(\alpha-1)\cdot 2^d \\
n\cdot t &< \alpha\cdot(n\cdot(t-1)-2^d)+2^d \\
n\cdot t &< \alpha\cdot n\cdot t-\alpha\cdot n-\alpha\cdot 2^d+2^d \\
t &< \alpha\cdot t-\alpha-\frac{(\alpha-1)\cdot 2^d}{n} \\
t-\alpha\cdot t &< -\alpha-\frac{(\alpha-1)\cdot 2^d}{n} \\
t\cdot(\alpha-1) &> \alpha+\frac{(\alpha-1)\cdot 2^d}{n} \\
t &> \frac{\alpha}{\alpha-1}+\frac{2^d}{n}.
\end{aligned}$$

Because $2^d \leq n$ at any layer of the recursion, $t_{max} = \frac{\alpha}{\alpha-1}+1$.

□

Using this, we can define s^*, an upper bound on our runtime recurrence where $s(0,n,\alpha) \leq s^*(1,0,n,\alpha)$ and

$$s^*(t,d,n,\alpha) = \begin{cases} 0 & t > t_{\max} \\ 0 & 2^d > n \\ \frac{n}{2^d} + s^*(2\cdot t-1, d+1, n, \alpha) + s^*(2\cdot t, d+1, n, \alpha) & \text{otherwise.} \end{cases}$$

The runtime of partitioning on the pivot closest to the center of the array

Theorem 2.1.8. *For $\alpha > 1$, partitioning on the pivot closest to the center of the array is $\in O\left(n\log\left(\frac{\alpha}{\alpha-1}\right)\right)$.*

Proof. Let d^* be the largest d for which all branches at layer d have work. Because $t_{max} = \frac{\alpha}{\alpha-1} + 1$ (Lemma 2.1.0.18), $d^* = \log_2(\frac{\alpha}{\alpha-1} + 1)$. This yields:

$$\begin{aligned}
r(n,\alpha) &= s(0,n,\alpha) \\
&\leq s^*(1,0,n,\alpha) \\
&\in O\left(\sum_{d=d^*}^{\log(n)} \sum_{t=1}^{t_{max}} \frac{n}{2^d}\right) + O(n \cdot d^*) \\
&\in O\left(\sum_{d=d^*}^{\log(n)} \frac{\alpha}{\alpha-1} \cdot \frac{n}{2^d}\right) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1} + 1\right)\right) \\
&\in O\left(\frac{n \cdot \alpha}{\alpha-1} \cdot \sum_{d=d^*}^{\log(n)} \frac{1}{2^d}\right) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right) \\
&\in O\left(\frac{n \cdot \alpha}{\alpha-1} \cdot \left(2^{1-d^*} - 2^{-\log(n)}\right)\right) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right) \\
&\in O\left(\frac{n \cdot \alpha}{\alpha-1} \cdot \left(2 \cdot \frac{\alpha-1}{2 \cdot \alpha - 1} - \frac{1}{n}\right)\right) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right) \\
&\in O\left(\frac{2 \cdot n \cdot \alpha}{2 \cdot \alpha - 1} - \frac{\alpha}{\alpha-1}\right) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right) \\
&\in O(n) + O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right) \\
&\in O\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right).
\end{aligned}$$

□

Theorem 2.1.9. *For $\alpha = 1$, partitioning on the pivot closest to the center of the array is optimal.*

Proof. Because we are sorting in this case, it suffices to show that this method is $\in O(n \log(n))$. Let d^* be the largest d for which all branches at that layer have work.

Because $\alpha = 1$, all branches have work. Thus $d^* = \log_2(n)$. This yields:

$$\begin{aligned} r(n,1) &\in O(n \cdot d^*) \\ &\in O(n \log(n)). \end{aligned}$$

□

Lemma 2.1.0.19. *Partitioning on the pivot closest to the center of the array is optimal for $\alpha \in (1,2)$.*

Proof.

$$r(n,\alpha) \in O\left(n \cdot \log\left(\frac{\alpha}{\alpha - 1}\right)\right) \text{ by Theorem 2.1.8.}$$

Because Theorem 2.1.2 puts LOHification in $\Omega\left(n \cdot \log\left(\frac{\alpha}{\alpha-1}\right)\right)$, this algorithm is optimal on these bounds. □

Lemma 2.1.0.20. *Partitioning on the pivot closest to the center of the array is optimal for $\alpha \geq 2$.*

Proof. Because increasing α decreases the number of layers, and thus the work, it suffices to show that the algorithm is optimal at $\alpha = 2$.

$$\begin{aligned} r(n,2) &\in O\left(n \cdot \log\left(\frac{2}{2-1}\right)\right) \\ &= O(n) \\ &\quad \text{Because loading the data puts } r(n) \in \Omega(n), \\ r(n) &\in \Theta(n). \end{aligned}$$

□

Theorem 2.1.10. *Partitioning on the pivot closest to the center of the array is optimal for all $\alpha \geq 1$.*

Proof.

By Theorem 2.1.9, Lemma 2.1.0.19, and Lemma 2.1.0.20, partitioning on the pivot closest to the center of the array is optimal for $\alpha \geq 1$

□

2.1.3 The optimal runtime for the construction of a layer-ordered heap of any rank

Partitioning on the pivot closest to the center of the array is optimal for all $\alpha \geq 1$ by Theorem 2.1.10. We can combine this with Theorem 2.1.1 to determine that LOHification (for $\alpha > 1$) is in:

$$\Theta\left(n \log\left(\frac{n}{n \cdot (\alpha - 1) + 1}\right) + \frac{n \cdot \alpha \cdot \log(\alpha)}{\alpha - 1}\right).$$

2.1.4 Quick LOHify

For this implementation of the algorithm, we partition on a random element, record the index of this element in an auxiliary array and then recurse on the left side until the minimum element is selected. While this method is probabilistic with a worst case construction in $O(n^2)$, it performs well in practice and has a linear expected construction time.

Expected Runtime of Quick LOHify

Quick LOHify can be thought of as a Quick-Selection with $k = 1$ and a constant number of operations per recursion for the auxiliary array. From this, we know the expected runtime to be in $\Theta(n)$. A direct proof is also provided.

Theorem 2.1.11. *The expected runtime for Quick LOHify is in* $\Theta(n)$.

Proof. The runtime is proportional to the number of comparisons. Suppose x_i is the i^{th} element in the sorted array and assume without loss of generality that $i < j$. We compare x_i and x_j only when one of these values is the pivot element.

Let $W_{i,j}$ be the probability that x_i is compared to x_j. Because we only recurse on the left, the smallest possible window that contains both x_i and x_j has j elements (both elements must be on the correct side of the previous pivot). Hence $W_{i,j} \leq \frac{2}{j}$. Since loading the data is in $\Omega(n)$, it only remains to show that our expected number of comparisons, $\mathbb{E}[c]$, is in $O(n)$.

The expected number of comparisons can be found by summing the probability over all pairs of elements. This yields:

$$
\begin{aligned}
\mathbb{E}[c] &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} W_{i,j} \\
\mathbb{E}[c] &\leq \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j} \\
&= 2 \cdot \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{1}{j} \\
&= 2 \cdot \begin{pmatrix}
1 & + \frac{1}{2} & + \frac{1}{3} & + \cdots & + \frac{1}{n-1} \\
 & + \frac{1}{2} & + \frac{1}{3} & + \cdots & + \frac{1}{n-1} \\
 & & + \frac{1}{3} & + \cdots & + \frac{1}{n-1} \\
 & & & + \ddots & + \frac{1}{n-1} \\
 & & & & + \frac{1}{n-1}
\end{pmatrix} \\
&= 2 \cdot (n-1) \\
&= 2 \cdot n - 2 \\
&\in O(n).
\end{aligned}
$$

□

Expected α of Quick LOHify

Unlike other constructions of a LOH, an α is not specified when performing Quick LOHify nor is it guaranteed to be the same across different runs. We can, however, determine that the expected value of α is in $\Theta(\log(n))$.

Theorem 2.1.12. *The expected α for Quick LOHify is $\in \Theta(\log(n))$.*

Proof.

The average α, $\mathbb{E}[\alpha]$, can be computed as the average ratio of the last two layers. This can be found by dividing the sum of all ratios by the number of ways to choose the pivots. This yields:

$$\begin{aligned}
\mathbb{E}[\alpha] &= \frac{1}{\binom{n}{2}} \cdot \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{n-j}{j-i} \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{n-j}{j-i} \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{n-i-k}{k} \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} \left(\sum_{k=1}^{n-i-1} \frac{n-i}{k} - 1 \right) \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} \left(\left(\sum_{k=1}^{n-i-1} \frac{n-i}{k} \right) - (n-i-2) \right) \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} \left((n-i) \cdot \left(\sum_{k=1}^{n-i-1} \frac{1}{k} \right) - n + i + 2 \right) \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} ((n-i) \cdot H_{n-i-1} - n + i + 2) \\
&= \frac{2}{n^2-n} \cdot \sum_{i=0}^{n-2} (n \cdot H_{n-i-1} - i \cdot H_{n-i-1} - n + i + 2)
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{n^2-n}\cdot\left(\left(n\cdot\sum_{i=0}^{n-2}(H_{n-i-1})\right)-\left(\sum_{i=0}^{n-2}(i\cdot H_{n-i-1})\right)\right.\\
&\quad \left.-(n^2-2\cdot n)+\left(\sum_{i=0}^{n-2} i\right)+(2\cdot n-4)\right)\\
&= \frac{2}{n^2-n}\cdot\left(\left(n\cdot\sum_{k=1}^{n-1}H_k\right)-\left(\sum_{k=1}^{n-1}(n-k-1)\cdot H_k\right)\right.\\
&\quad \left.-(n^2-2\cdot n)+\frac{n^2-3\cdot n+2}{2}+(2\cdot n-4)\right)\\
&= \frac{2}{n^2-n}\cdot\left(\left(\sum_{k=1}^{n-1}(k+1)\cdot H_k\right)+\frac{-n^2+5n-6}{2}\right)\\
&\quad \text{which we simplify with Wolfram Mathematica to:}\\
&= \frac{2}{n^2-n}\cdot\left(\frac{(n^2+n)\cdot H_n}{2}-\frac{n^2}{4}-\frac{3\cdot n}{4}+\frac{-n^2+5\cdot n-6}{2}\right)\\
&= \frac{2}{n^2-n}\cdot\frac{2\cdot n^2\cdot H_n+2\cdot n\cdot H_n-3\cdot n^2+7\cdot n-12}{4}\\
&= \frac{2\cdot n^2\cdot H_n+2\cdot n\cdot H_n-3\cdot n^2+7\cdot n-12}{2\cdot n^2-2\cdot n}\\
&\in \Theta(\log(n)).
\end{aligned}
$$

□

2.2 Optimal α for top-k on $X+Y$

Now that we have an optimal LOHification algorithm for any α, we can estimate the optimal α for performing a top-k on $X+Y$ via Serang's method [6] where X and Y are arrays of length n. Because LOHification with $\alpha \geq 2$ is in $\Theta(n)$ (Lemma 2.1.0.19), and increasing α only improves the speed of LOHification itself; we will focus on the interval $\alpha \in (1, 2]$.

2.2.1 Overview of the algorithm

Serang's algorithm is divided into four phases.

Phase 0

X and Y are both LOHified using the optimal LOHification algorithm.

Phase 1

Layer products of the form $X^{(u)} + Y^{(v)}$ (where $X^{(u)}$ and $Y^{(v)}$ are layers in their respective LOHs) are considered. For this phase and the next, only the minimum and maximum values in the layer products are generated. We also note whether the value is a minimum or maximum. We represent these as tuples of the form (r, b, u, v) where r represents the extreme value in $X^{(u)} + Y^{(v)}$, b is 0 if the extreme value is a minimum and 1 if the extreme value is a maximum (we have two tuples for every layer product), and u and v represent the indices in the X-LOH and Y-LOH respectively. The tuples associated with minimal values in layer products will be referred to as min-corners and the tuples associated with maximal values in layer products will be referred to as max-corners.

These tuples are then put into a priority queue and popped in sorted order until we are guaranteed the popped max-corners are associated with layer products that have a total area at least k. The contents of layer products associated with popped max-corners are now generated and put into a vector.

Phase 2

The values associated with all max values that remain in our queue are now generated and put into the vector. This is because their min corners have been popped,

and therefore those layer products may contain some values at most the maximum max corner currently popped from the priority queue. Now the vector contains all values that may be less than the worst max corner that has been popped from the priority queue. Since the k^{th} best value is the minimal value with at least k values less than or equal to it, then we have at least k values, which represent all values at most the worst max corner that has been popped from the priority queue; therefore, that worst max corner popped is an upper bound for the k^{th} smallest value.

Phase 3

A 1-dimensional k-selection is done on the vector. This runs in linear time using median-of-medians [11]. Serang shows that the vector contains $\Theta(k)$ values because of a geometric series that appears in the sizes of the layer products, which until the final max corner was popped from the priority queue, must have had less than k values in the vector [6].

2.2.2 Deriving the runtime in terms of n, k, and α

When deriving the runtime for this algorithm, the important things to consider are the cost of LOHification, the cost of generating the layer products, the cost of the partial sorting of the layer products, and the size of the final vector that we perform the top-k on.

We know from Lemma 2.1.0.19 that the cost of LOHification (using the optimal method on the bounds we are considering) is in $\Theta(n \log(\frac{\alpha}{\alpha-1}))$. Next, we will derive a bound on the total number of layer products to help analyze the runtime of generating and performing a partial sort on the layer products.

Lemma 2.2.0.1 (There are $o(n^{1/2})$ layer products as n grows for $\alpha > 1$).

Proof. By Lemma 2.1.0.3, the asymptotic number of layers in a LOH is $\log_\alpha(n \cdot (\alpha - 1) + 1)$. Because X and Y both have length n and are LOHified using the same α, the total number of layer products in $X + Y$ is $(\log_\alpha(n \cdot (\alpha - 1) + 1))^2$.

$$
\begin{aligned}
& \lim_{n\to\infty} \frac{(\log_\alpha(n \cdot (\alpha - 1) + 1))^2}{n^{1/2}} \\
= & \lim_{n\to\infty} \frac{\left(\frac{\log(n \cdot (\alpha - 1) + 1)}{\log(\alpha)}\right)^2}{n^{1/2}} \\
= & \lim_{n\to\infty} \frac{(\log(n \cdot (\alpha - 1) + 1))^2}{n^{1/2} \cdot (\log(\alpha))^2} \\
= & \lim_{n\to\infty} \frac{\left(\frac{2 \cdot (\alpha-1) \cdot \log(n \cdot (\alpha-1)+1)}{n \cdot (\alpha-1)+1}\right)}{\left(\frac{(\log(\alpha))^2}{2n^{1/2}}\right)} \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \frac{4n^{1/1} \cdot (\alpha - 1) \cdot \log(n \cdot (\alpha - 1) + 1)}{n \cdot (\alpha - 1) \cdot (\log(\alpha))^2 + (\log(\alpha))^2} \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{(\alpha - 1) \cdot \log(n \cdot (\alpha - 1) + 1)}{n^{1/2} \cdot (\alpha - 1) + \frac{1}{n^{1/2}}}\right) \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{\log(n \cdot (\alpha - 1) + 1}{n^{1/2}}\right) \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{\left(\frac{\alpha-1}{n \cdot (\alpha-1)+1}\right)}{\left(\frac{1}{2n^{1/2}}\right)}\right) \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{2n^{1/2} \cdot (\alpha - 1)}{n \cdot (\alpha - 1) + 1}\right) \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{\left(\frac{\alpha-1}{n^{1/2}}\right)}{\alpha - 1}\right) \text{ by L'Hôpital's rule} \\
= & \lim_{n\to\infty} \left(\frac{4}{(\log(\alpha))^2}\right) \cdot \left(\frac{1}{n^{1/2}}\right) \\
= & \ 0
\end{aligned}
$$

□

Generating a tuple associated with layer product $X^{(u)} + Y^{(v)}$ only requires the min

and max elements of $X^{(u)}$ and $Y^{(v)}$. Getting the min and max values of an array is trivially done in $O(n)$. Because the layers of a LOH partition the original array, the total time to get all mins and maxes in a LOH is in $O(n)$ which is in $O(Phase0)$. Once we have these values, every tuple can be generated in $O(1)$ time. Thus the time it takes to generate all tuples needed is in $o(n^{1/2})$ by Lemma 2.2.0.1; which, is in dwarfed by the cost of Phase 0. By using a binary heap as our priority queue, the partial sort on the tuples can take no longer than a heap-sort on all the tuples, which puts the partial sort in $O(n^{1/2}\log(n))$ which is in $O(Phase0)$.

Because top-k on an array of length n can be done in $O(n)$ [11], we now need only a bound on the size of the final vector.

Lemma 2.2.0.2 (The size of the final vector is at most $k \cdot (\alpha^2 + \alpha + 1)$).

Proof. Let I be the set of all layer products associated with max-corners that have been popped from our priority queue immediately before the final max-corner popped in Phase 1. Let u^* be the greatest integer, z, such that $X^{(z)} + Y^{(0)} \in I$. Let v^* be the greatest integer, z, such that $X^{(0)} + Y^{(z)} \in I$. Let v_x for any $v \leq v^*$ be the greatest integer, z, such that $X^{(z)} + Y^{(v)} \in I$. Let u_v for any $u \leq u^*$ be the greatest integer, z, such that $X^{(u)} + Y^{(z)} \in I$. It should be noted that $X^{(u)} + Y^{(v)} \in I$ implies $X^{(i)} + Y^{(j)} \in I$ for all $0 \leq i \leq u$ and $0 \leq j \leq v$. This is because a minimal element in $X^{(i)}$ cannot be greater than a minimal element in $X^{(i+1)}$ by our layer ordering.

Let $|I|$ represent the number of elements in $X + Y$ that are in the union of all layer products in I.

By this construction,

$$|I| = \sum_{i=0}^{u^*} |X^{(i)}| \cdot \left(\sum_{j=0}^{u_i} |Y^{(j)}| \right) = \sum_{j=0}^{v^*} |Y^{(j)}| \cdot \left(\sum_{i=0}^{v_j} |X^{(i)}| \right)$$

Let V be the set of all layer products associated with max-corners that have ever been pushed into our priority queue. Because a max-corner in the priority queue implies that the associated min-corner has been popped, V is bounded by the min-corners that have ever been in our priority queue. Because $(r_{0,u,v}, 0, u, v)$ proposes $(r_{1,u,v}, 1, u, v)$, $(r_{0,u+1,v}, 0, u+1, v)$, and $(r_{0,u,v+1}, 0, u, v+1)$; and $(r_{1,u,v}, 1, u, v) < (r_{0,u+1,v+1}, 0, u+1, v+1)$ the the number of elements in $X + Y$ that are in the union of all layer products in V, $|V|$ (the size of the final vector) can be bounded by:

$$|V| \leq \sum_{i=0}^{u^*+1} |X^{(i)}| \cdot \left(\sum_{j=0}^{u_i+1} |Y^{(j)}| \right)$$

Because layers in both X and Y grow by a factor of α,

$$\begin{aligned} |V| &\leq \sum_{i=0}^{u^*} |X^{(i)}| \cdot \left(\sum_{j=0}^{u_i} |Y^{(j)}| \right) + \alpha \cdot \left(\sum_{i=0}^{u^*} |X^{(i)}| \cdot \left(\sum_{j=0}^{u_i} |Y^{(j)}| \right) \right) \\ &\quad + \alpha^2 \cdot \left(\sum_{i=0}^{u^*} |X^{(i)}| \cdot \left(\sum_{j=0}^{u_i} |Y^{(j)}| \right) \right) \\ &\leq |I| + \alpha \cdot |I| + \alpha^2 \cdot |I| \\ &\leq |I| \cdot (\alpha^2 + \alpha + 1) \end{aligned}$$

By definition, $|I| < k$, so the size of the final vector is bounded by $k \cdot (\alpha^2 + \alpha + 1)$ in the worst case. □

Fig 2.2 and Fig 2.3 provide a visual aid to the previous theorem.

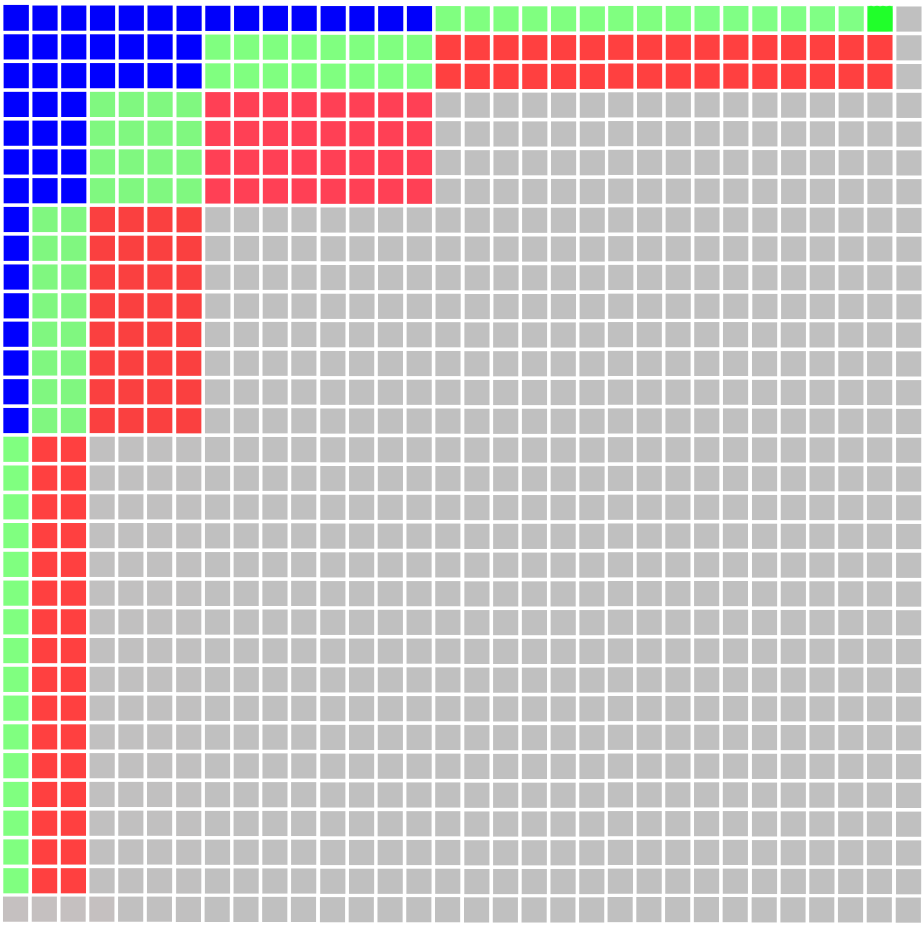

Figure 2.2 **A color coded depiction of $X + Y$.** In this depiction, the blue squares represent the elements in layer products associated that are in I. The light green squares represent elements in layer products that are adjacent to I, and the red squares represent elements in layer products that are diagonal to I. All non-gray squares represent elements in layer products that are in V. The layer products $X^{(0)} + Y^{(u_0+2)}$ and $X^{(v_0+2)} + Y^{(0)}$ are not in V because $|X^{(0)}| = |Y^{(0)}| = 1$ hence $(r_{1,u,0}, 1, u, 0) < (r_{0,u+1,0}, 0, u+1, 0)$ and $(r_{1,0,v}, 1, 0, v) < (r_{0,0,v+1}, 0, 0, v+1)$.

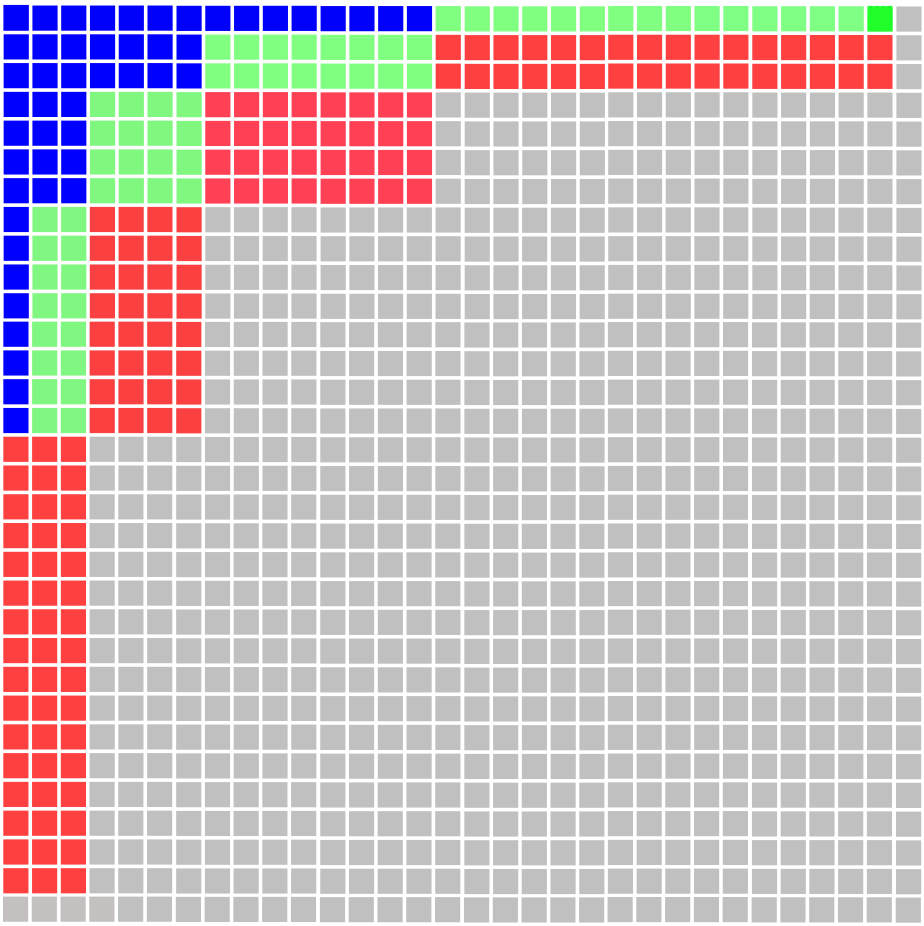

Figure 2.3 **An alternate coloring of $X + Y$.** With this recoloring (the bottom left layer product is now red), it is easy to see that the area of the green part is bounded by α times the area of the blue part and the area of the red part is bounded by α times the area of the blue and green parts combined.

Theorem 2.2.1 (An estimate of the optimal α for top-k on $X + Y$). *Using the bounds found earlier, we can determine that the optimal α for top-k on $X + Y$ is approximately*

$$\frac{7k}{3\cdot 2^{2/3}\cdot \sqrt[3]{20k^3+\frac{432k^2\cdot n}{\ln(2)}+\sqrt{\left(20k^3+\frac{432k^2\cdot n}{\ln(2)}\right)^2-1372k^6}}} + \frac{\sqrt[3]{20k^3+\frac{432k^2\cdot n}{\ln(2)}+\sqrt{\left(20k^3+\frac{432k^2\cdot n}{\ln(2)}\right)^2-1372k^6}}}{6\cdot \sqrt[3]{2}\cdot k} + \frac{1}{6}$$

Proof. By the construction of our algorithm, we perform at most two selections per layer in our recursion tree for LOHification. Because of this, the total cost of LOHification for X and Y will be approximately $c \cdot 4 \cdot \log_2(\frac{\alpha}{\alpha-1})$ in the worst case where c is the runtime constant of our 1-D selection algorithm. Likewise, the cost of our final selection will be approximately $c \cdot k \cdot (\alpha^2 + \alpha + 1)$ in the worst case. Because the cost of all other operations is in $o(n)$, our total runtime for large n and k is approximately $f(\alpha) = c \cdot 4 \cdot \log_2(\frac{\alpha}{\alpha-1}) + c \cdot k \cdot (\alpha^2 + \alpha + 1)$.

We can use the first derivative test to find the optimal α in terms of n and k.

$$\frac{\partial}{\partial \alpha} f(\alpha) = \frac{-c \cdot 4n \cdot \frac{1}{\ln(2)}}{\alpha^2 - \alpha} + c \cdot (2k \cdot \alpha + k)$$

Now, solving for 0 we get:

$$\begin{aligned}
\frac{-c \cdot 4n \cdot \frac{1}{\ln(2)}}{\alpha^2 - \alpha} + c \cdot (2 \cdot k \cdot \alpha + k) &= 0 \\
\frac{\frac{-4}{\ln(2)} \cdot n}{\alpha^2 - \alpha} + 2 \cdot k \cdot \alpha + k &= 0 \\
2k \cdot \alpha^3 + k \cdot \alpha^2 - 2k \cdot \alpha^2 - k \cdot \alpha - \frac{4}{\ln(2)} \cdot n &= 0 \\
2k \cdot \alpha^3 - k \cdot \alpha^2 - k \cdot \alpha - \frac{4}{\ln(2)} \cdot n &= 0
\end{aligned}$$

Simplifying this with Wolfram Mathematica [1] yields:

$$\begin{aligned}
\alpha = & \frac{7k}{3 \cdot 2^{2/3} \cdot \sqrt[3]{20k^3 + \frac{432k^2 \cdot n}{\ln(2)} + \sqrt{\left(20k^3 + \frac{432k^2 \cdot n}{\ln(2)}\right)^2 - 1372k^6}}} \\
& + \frac{\sqrt[3]{20k^3 + \frac{432k^2 \cdot n}{\ln(2)} + \sqrt{\left(20k^3 + \frac{432k^2 \cdot n}{\ln(2)}\right)^2 - 1372k^6}}}{6 \cdot \sqrt[3]{2} \cdot k} + \frac{1}{6}
\end{aligned}$$

[1] Solve[2*k*a^3-k*a^2-k*a-(4/log(2))*n==0,a]

As the only real solution for non-zero real values of n and k. We will denote this as $A(n, k)$.

Using the second derivative test, we can show that this value is, in fact, a minimum.

$$\frac{\partial^2}{\partial \alpha^2} f(\alpha) = \frac{\frac{c \cdot 4n}{\ln(2)} \cdot (2\alpha - 1)}{(\alpha - 1)^2 \cdot \alpha^2} + c \cdot 2k$$

We can see that this value is greater than 0 for $n > 0$, $k > 0$, $c > 0$, and $\alpha \in (1, 2]$. Because LOHification with $\alpha \geq 2 \in O(n)$, the optimal α in the worst case is approximately $\min(A(n, k), 2)$.

□

We evaluate our estimate against empirical data in the Results chapter.

2.3 Top-k on a multinomial

The multinomial distribution is the generalization of the binomial distribution for $k > 2$ possible outcomes per trial. This can be used to model the distribution of isotopologues in chemicals, biological phenomena, election polling, and other things. The application of an online top-k algorithm for multinomials is used in the isotope calculator **NeutronStar** [9]. This works by creating LOHs out of the top-k from the multinomial 'leaves' in an online manner that are fed to a Cartesian product tree [14] to compute the top-k isotopes of a compound.

The depth of a Cartesian product tree is $\log_2(m)$, thus the accumulated overshooting when using the Serang $X + Y$ method to combine each pair in the tree will be $m^{(\log_2(\alpha^2+\alpha+1)}$ which favors an α near 1. The LOHification process favors a large α, so this becomes another balancing act to minimize the overall work.

Let there be k mutually exclusive events: $e_1, \ldots, e_k$, with probabilities: $p_1, \ldots, p_k$

such that $\sum_{i=1}^{k} p_i = 1$. For n independent trials where exactly one of the k events happens per trial, the event that we observe $x_1, \ldots, x_k$ instances of events $e_1, \ldots, e_k$ respectively will be labeled $(x_1, \ldots, x_k)$. The probability of a configuration is $P(x_1, \ldots, x_k) = n! \cdot \prod_{i=1}^{k} \frac{p_i^{x_i}}{x_i!}$.

The multinomial can be thought of as a polynomial to a power. The elements with the k largest coefficients is a type of approximation. It is an especially useful approximation when there are a lot of terms and most coefficients are near zero such as the isotopic distribution of a large homo-elemental compound like C_{60}. For hetero-elemental compounds, the distribution is the coefficients of the product of constituent multinomials.

2.3.1 Convexity of a multinomial

Here we will show that the multinomial distribution has a type of convexity that we can use to both find its mode(s) (a term with a maximal coefficient) and perform a top-k on it. We will start with a simple lemma. Note that $[m]$ is the set of all integers between 1 and m including 1 and m.

Lemma 2.3.0.1 (Directional Convexity). *Let $(x_1, x_2, \ldots, x_m)$ be a mode in our distribution. Then for $j, k \in [m]$, $\frac{p_j \cdot x_k}{p_k \cdot (x_j + 1)} \leq 1$*

Proof.

$$\begin{aligned} P(x_1, x_2, \ldots, x_m) &= n! \cdot \prod_{i=1}^{m} \frac{p_i^{x_i}}{x_i!} \\ &\quad \text{because } P(x_1, x_2, \ldots, x_m) \text{ is a mode, we have:} \\ P(x_1, x_2, \ldots, x_m) &\geq P(\ldots, x_j + 1, \ldots, x_k - 1, \ldots) \\ n! \cdot \prod_{i=1}^{m} \frac{p_i^{x_i}}{x_i!} &\geq n! \cdot \frac{p_j^{x_j+1} \cdot p_k^{x_k-1}}{(x_j+1)! \cdot (x_k-1)!} \cdot \prod_{i \in [m] \setminus \{j,k\}} \frac{p_i^{x_i}}{x_i!} \end{aligned}$$

$$\begin{aligned}
\frac{p_j^{x_j} \cdot p_k^{x_k}}{x_j! \cdot x_k!} &\geq \frac{p_j^{x_j+1} \cdot p_k^{x_k-1}}{(x_j+1)! \cdot (x_k-1)!} \\
p_j^{x_j} \cdot p_k^{x_k} \cdot (x_j+1) &\geq p_j^{x_j+1} \cdot p_k^{x_k-1} \cdot x_k \\
p_k \cdot (x_j+1) &\geq p_j \cdot x_k \\
1 &\geq \frac{p_j \cdot x_k}{p_k \cdot (x_j+1)}.
\end{aligned}$$

□

Using this, we will demonstrate in the following theorems that every time the i^{th} entry in the index tuple, (a tuple that describes how many of each event is represented in the element), is increased and the j^{th} entry is decreased, thus moving further from the mode in $\mathbb{L}_1$ (or Manhattan) distance, the probability never increases.

Theorem 2.3.1 (Pure Case). *Let $(x_1, x_2, \ldots, x_m)$ be a mode in our distribution. Then for $j, k \in [m]$, and $b \in [\min(x_j, x_k)]$; $P(\ldots, x_j + b, \ldots, x_k - b, \ldots) \geq P(\ldots, x_j + (b+1), \ldots, x_k - (b+1), \ldots)$*

Proof.

$$\begin{aligned}
P(\ldots, x_j + b, \ldots, x_k - b, \ldots) &= n! \cdot \frac{p_j^{x_j+b} \cdot p_k^{x_k-b}}{(x_j+b)! \cdot (x_k-b)!} \cdot \prod_{i \in [m] \setminus \{j,k\}} \frac{p_i^{x_i}}{x_i!} \\
P(\ldots, x_j + b, \ldots, x_k - b, \ldots) &= P(\ldots, x_j + (b+1), \ldots, x_k - (b+1), \ldots) \\
&\quad \cdot \frac{p_j \cdot (x_k - b)}{p_k(x_j + (b+1))} \\
&\quad \text{because} \\
\frac{p_j \cdot x_k}{p_k \cdot (x_j+1)} &\geq \frac{p_j \cdot (x_k - b)}{p_k(x_j + (b+1))} \\
&\quad \text{and} \\
1 &\geq \frac{p_j \cdot x_k}{p_k \cdot (x_j+1)} \text{ Lemma 2.3.0.1} \\
&\quad \text{we have}
\end{aligned}$$

$$\begin{aligned} 1 &\geq \frac{p_j \cdot (x_k - b)}{p_k(x_j + (b+1))} \\ &\text{hence;} \\ P(\dots, x_j + b, \dots, x_k - b, \dots) &\geq P(\dots, x_j + (b+1), \dots, x_k - (b+1), \dots). \end{aligned}$$

□

Theorem 2.3.2 (Mixed Case). *Let $(x_1, x_2, \dots, x_m)$ be a mode in our distribution. Then for $i, j, k \in [m], P(\dots, x_i+1, \dots, x_j-1, \dots) \geq P(\dots, x_i+2, \dots, x_j-1, \dots, x_k-1, \dots)$*

Proof.

$$\begin{aligned} P(\dots, x_i + 1, \dots, x_j - 1, \dots) &= P(\dots, x_i + 2, \dots, x_j - 1, \dots, x_k - 1, \dots) \\ &\quad \cdot \frac{p_i \cdot (x_k)}{p_k(x_i + 2)} \\ &\text{because} \\ \frac{p_i \cdot x_k}{p_k \cdot (x_i + 1)} &\geq \frac{p_i \cdot (x_k)}{p_k(x_i + 2)} \\ &\text{and} \\ 1 &\geq \frac{p_i \cdot x_k}{p_k \cdot (x_i + 1)} \text{ Lemma 2.3.0.1} \\ &\text{we have} \\ 1 &\geq \frac{p_i \cdot (x_k)}{p_k(x_i + 2)} \\ &\text{hence;} \\ P(\dots, x_i + 1, \dots, x_j - 1, \dots) &\geq P(\dots, x_i + 2, \dots, x_j - 1, \dots, x_k - 1, \dots). \end{aligned}$$

□

Theorem 2.3.3 (Heterogeneous Case). *Let $(x_1, x_2, \dots, x_m)$ be a mode in our distribution. Then for $i, j, k, \ell \in [m], P(\dots, x_i+1, \dots, x_j-1, \dots) \geq P(\dots, x_i+1, \dots, x_j-$*

$1, \ldots, x_k + 1, \ldots, x_\ell - 1, \ldots)$

Proof.

$$\begin{aligned} P(\ldots, x_i + 1, \ldots, x_j - 1, \ldots) &= P(\ldots, x_i + 1, \ldots, x_j - 1, \ldots, x_k + 1, \ldots, \\ &\quad x_\ell - 1, \ldots) \cdot \frac{p_k \cdot (x_\ell)}{p_\ell(x_k + 1)} \\ 1 &\geq \frac{p_k \cdot (x_\ell)}{p_\ell(x_k + 1)} \text{ Lemma 2.3.0.1} \\ &\quad \text{hence;} \\ P(\ldots, x_i + 1, \ldots, x_j - 1, \ldots) &\geq P(\ldots, x_i + 1, \ldots, x_j - 1, \ldots, x_k + 1, \ldots, \\ &\quad x_\ell - 1, \ldots). \end{aligned}$$

□

By the previous theorems, the relationship between the $\mathbb{L}_1$ distance from the mode and the probability still holds when other index tuple entries have been perturbed away from the mode. This means that if there is a shortest path from the mode to element z that contains element y, then $y \geq z$.

Since the probability never decreases as we move closer to a mode, then we can reach a mode by hill-climbing using any element in the multinomial as a starting point. Once we are at a location which can not increase in probability, we have reached the mode. Because the distribution is discrete and we move by the smallest possible amount (incrementing and decrementing a pair of indices by one), we will never overshoot a mode. Consequently, if there are multiple modes, they must be adjacent (increasing exactly one entry in the tuple and decreasing exactly one other entry yields the other tuple).

It should be noted that these facts about the mode of a multinomial have been discovered earlier [15]; however, their application to top-k is novel.

2.3.2 An algorithm for top-k on a multinomial

Top-k on a multinomial begins with a mode of the distribution. This starting position is found by using the modes of each the binomial marginals and correcting if the sum is not n, although any starting position will lead to a mode because there are no local maxima.

We will begin with a max-priority queue, Q, that initially contains only the mode. In the software, Q is implemented as a binary max-heap. Then we keep popping the next best element until we have the k-best elements.

In order to populate Q with the best possible element that has not been popped, any element in Q must have all elements between itself and the mode already in Q (or have been popped from Q); this ensures that the $\mathbb{L}_1$ distance of proposed index tuples is always increasing and thus index tuples are visited in descending order of probability. This is accomplished by pushing all neighbors of the element that has been popped from the heap. These neighbors are found as with the search for the mode: from some starting point, one index is increased and one index is decreased, thereby holding the sum constant; however, unlike the search for the mode, here we must guarantee that the $\mathbb{L}_1$ distance from the mode always increases, and so proposed neighbors that would move closer to the mode on any axis are discarded.

Subsequent elements are generated in descending order of likelihood using Q, where keys are the probabilities of each element.

It is necessary to prevent duplicates from being inserted into Q. For example, index tuple $(30, 4, 3)$ has neighbors $(29, 5, 3)$, $(29, 4, 4)$, $(31, 3, 3)$, $(31, 4, 2)$, $(30, 5, 2)$, $(30, 3, 4)$. Of these, $(29, 5, 3)$ and $(30, 5, 2)$ both have neighbor $(29, 6, 2)$. One way to prevent these duplicates from being reinserted into Q is to store a set of the Q's contents; however, that requires additional memory and time. Although it is asymp-

totically comparable to the cost of pushing to and popping from Q, it significantly harms performance in practice.

For this reason, we use a proposal scheme that can reach all elements in increasing $\mathbb{L}_1$ order from the mode, but without duplicates. When an element is popped from the heap, it proposes new elements to enter the heap based on their index tuple.

A proposal can be characterized by the two axes, i, j that are perturbed (without loss of generality, let index i increase and index j decrease). If two chains of neighbors, $(i_1, j_1), (i_2, j_2), \ldots$ and $(i'_1, j'_1), (i'_2, j'_2), \ldots$, collide then $multiset(i_1, i_2, \ldots) = multiset(i'_1, i'_2, \ldots)$, $multiset(j_1, j_2, \ldots) = multiset(j'_1, j'_2, \ldots)$. Multisets are unique when their contents are sorted, and thus chains whose i and j are both in lexicographic order (by the index of the largest entry that has been perturbed as i and j respectively) will visit each index tuple only once. This proposal scheme means any index tuple may be proposed by only one unique neighboring index tuple.

As mentioned earlier, a multinomial distribution can be used to model the abundance of isotopologues of one-element compounds. For example, the first few proposals for the most abundant isotopologues of K_{100}, (a compound consisting of 100 potassium atoms), may be seen in (Figure 2.4). A version of this algorithm is implemented in the open-source isotopologue calculator **NeutronStar** [9] that can be found at `https://figshare.com/articles/software/NeutronStar_version_1/16837387`.

2.3.3 A theoretical improvement to our existing algorithm

While the existing algorithm is practically efficient for it's use in generating the top k isotopologues of a compound, the fact that we get the elements in sorted order from a binary heap puts our complexity in $\Omega(k \cdot \log(k))$. Because every element in a multinomial distribution can propose $2 \cdot \binom{m}{2}$ neighbors in the worst case, our lower

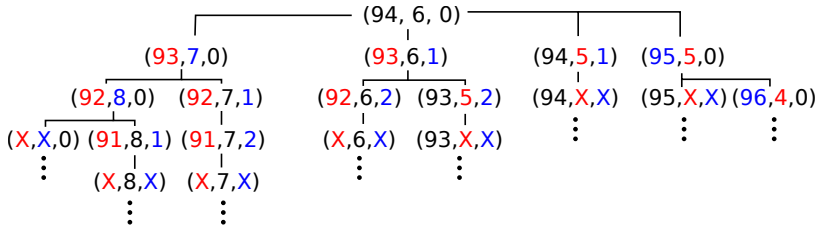

Figure 2.4 **First few multinomial proposals for the most abundant isotopologues of K_{100} (a compound consisting of 100 potassium atoms).** The figure shows index tuples in the multinomial and the neighbors they propose, from top to bottom, starting with the mode, $(94, 6, 0)$ (94 copies of ^{39}K, 6 copies of ^{41}K, and no copies of ^{40}K). Each index tuple proposes its neighbors in lexicographical order where, if the i^{th} index has been incremented, it cannot propose any neighbors by incrementing an index less than i (this same pattern is used for decrementing an index). In the figure, the largest index to be incremented is in blue and the largest to be decremented is in red. In order to move away from the mode, any index which has been incremented may not be decremented to create a proposed tuple, and vice versa. Note that for clarity not all proposed indices are included.

bound becomes $\Omega(k \cdot m^2 \cdot \log(k \cdot m))$. When we account for the fact that computing the weight of a term (i.e. its probability) is linear in m, our final floor, with getting the k elements in sorted order, becomes $\Omega(k \cdot m^3 \cdot \log(k \cdot m))$.

This can be improved by using the Frederickson algorithm for selection in a min-heap [16]. Because the elements can only propose inferior neighbors and every element has at most $2 \cdot \binom{m}{2}$, we can represent our data as a d-ary heap where $d = 2 \cdot \binom{m}{2}$ and elements are only generated as they are needed. Using this, we can adapt the Frederickson algorithm to perform a top-k on a multinomial in $O(km^3)$ plus the time it takes to compute the mode.

The mode can be computed by calculating the modes of the binomial cross-sections of our multinomial and considering the two possible values for each of the m terms in our multinomial. From the Finucan derivation [15], one of these 2^m configurations

is a mode. Because the mode necessarily satisfies the multinomial support (i.e. the terms sum to n), we only need to check at most $\binom{m}{2}$ of these configurations. Because the time it takes to check one configuration is in $O(m)$, we can compute the mode in $O(m^3)$. Thus, we can compute the top k elements of a multinomial in $O(km^3)$.

We can improve this even further by reusing work done to compute weights of elements. We can compute the weight of element, E_1, by multiplying the weight of an adjacent element, E_o, by $\frac{p_j \cdot (x_i - 1)}{p_i \cdot (x_j + 2)}$ where substituting the i^{th} and j^{th} term in E_0 with its i^{th} term -1 and j^{th} term $+1$ (respectively) yields element E_1 (this can be seen in Theorem 2.3.1). Since we necessarily have the weight of the proposing term at the point any element is proposed in our heap, we need only compute the weight of a single term in $O(m)$. This holds for the mode as well due to the fact that all candidates are adjacent to another candidate. Thus, we can reduce our complexity to $O(km^2)$. The ordering we have on the elements of a multinomial distribution make this possible.

CHAPTER 3 Results

All tests were performed on a machine with an AMD Ryzen 9 3900x processor and 64GB of RAM running Ubuntu 18.04 LTS and were all compiled using GCC Version 7.5.0 with the following flags: `-std=c++17 -g -O3 -Wall -march=native -mtune=native` .

3.1 Computing a False Discovery Rate Threshold with LOHs

We will first compare the runtimes of various LOHify algorithms used to compute the most permissive score threshold at which a given false discovery rate (FDR) [17] τ occurs. This is traditionally accomplished by sorting the scored hypotheses (which are labeled as TP, for true positive, or FP, for false positive) in descending order and then advancing one element at a time, updating the FDR to the current $FDR = \frac{\#FP}{\#FP+\#TP}$ at the threshold, finding the worst score at which $FDR \leq \tau$ occurs.

The LOH-based methods for this behave similarly, but they compute bounds on the FDRs in each layer. When these bounds may include τ, the layer is recursed on until the size of the layer is $\in O(1)$ or the bounds may no longer contain τ. It continues checking candidate layers until the location of the threshold is determined.

Table 3.1 demonstrates the performance benefit of using LOHs over sorting and the practical performance of Quick-LOHify, which handedly wins overall. It especially demonstrates the scale advantage that LOHs have over sorting. Table 3.2 provides a

n	SORT	SLWGI	SDRPIH	PPCCA	QUICK
2^{28}	27.1712	3.60989	6.64702	3.55166	1.20981
2^{27}	13.4568	2.67432	2.98285	2.74130	0.840145
2^{26}	6.44227	1.03872	1.86184	1.05260	0.596104
2^{25}	3.06724	0.58973	0.890603	0.58956	0.266691

Table 3.1 **Runtimes (seconds) of different LOHification methods for computing FDR cutoffs on data of various sizes.** Reported runtimes are averages over 10 iterations, $\alpha = 6.0$ (where applicable). SORT is sorting, SLWGI is selecting the layer with the greatest index, SDRPIH is selecting to divide the remaining pivot indices in half, PPCCA is partitioning on the pivot closest to the center of the array, and QUICK is Quick-LOHify. Quick-LOHify generates its own partition indices, which are not determined by an α parameter.

	α	SLWGI	SDRPIH	PPCCA
	1.05	29.9283	16.8750	14.4878
	1.1	19.3675	14.8553	12.9062
	1.5	8.92916	11.0468	8.12265
Size $= 2^{28}$	2.0	8.24106	9.24010	8.21261
	3.0	5.73344	7.94349	5.60023
	4.0	3.83187	6.35753	3.84014
	6.0	3.60989	6.64702	3.55166
	8.0	4.62627	6.90307	4.5759

Table 3.2 **Runtimes (seconds) of different LOHification methods for computing FDR cutoffs with various α.** Reported runtimes are averages over 10 iterations. The abbreviations are the same as Table 3.1

demonstration of the influence α has on practical performance.

From Table 3.1, we see that computing FDR cutoffs can be done efficiently with a large α. This is in part due to the fact that we recurse on layers when more detail is needed and that we can only overshoot in one dimension. In cases like this, performance does not suffer from using a naive LOHification algorithm over the optimal one. Such applications can also benefit from Quick-LOHify which has a massive expected α (the expected α for $n = 2^{28}$ is approximately 19.985).

3.2 Computing an m-dimensional Cartesian Product with LOHs

There are other problems, such as selecting the top k values of a Cartesian product of m arrays that each contain n values, that require α to be much closer to one for both theoretical and practical performance. Assuming the cost of LOHification is linear in n, the fastest algorithm for this to date is in $O\left(n \cdot m + k \cdot m^{\log_2(\alpha^2)}\right)$ [7]. Using the optimal method of LOHification puts the real complexity in $O\left(n \cdot m \cdot \log(\frac{\alpha}{\alpha-1}) + k \cdot m^{\log_2(\alpha^2)}\right)$ (assuming $1 < \alpha \leq 2$). From this, we can see that the LOHification algorithm can have a major theoretical impact on the performance of this algorithm. As can be seen in Table 3.3, this translates to real performance as well.

Finding the best α for this algorithm given the values of the other parameters is a difficult problem. The choice of α affects not only the cost to LOHify the leaves, but also the upper bound on the total number of elements that will be generated before the final selection. It should be noted that α affects the pessimal amount of 'overshooting' that the algorithm can experience, not the optimal. This fact is highlighted in Table 3.3 where the top-k with $\alpha = 1.9$ has over 10 times the overshoot of $\alpha = 2.0$. It should be stressed; however, that using $\alpha = 2.0$ will have more overshoot, on average, than using $\alpha = 1.9$.

α	SLWGI	SDRPIH	PPCCA	Elements Generated
1.001	241.392	21.7816	21.6659	1,000,581,716
1.005	61.7801	19.6801	19.4993	1,001,518,376
1.01	41.1727	20.3318	20.1373	1,001,441,269
1.05	24.1046	21.414	20.8471	1,032,606,487
1.1	22.3487	21.7075	21.3163	1,071,965,768
1.2	22.4767	21.5742	21.3030	1,030,388,107
1.3	26.0196	25.7922	24.4722	1,171,197,582
1.4	23.3544	23.9249	23.2303	1,045,909,512
1.5	30.4798	29.3044	29.7394	1,275,242,999
1.6	36.8700	38.1468	35.5288	1,596,349,521
1.7	33.4038	34.8689	34.4841	1,388,065,287
1.8	29.5818	30.4977	29.0531	1,075,097,102
1.9	55.6099	46.5149	54.8003	1,756,411,702
2.0	29.8097	31.5725	29.6259	1,073,741,823

Table 3.3 **Runtimes (seconds) of generating the top billion values of the Cartesian product of 8 arrays of length 10 million using different LOHification methods with various α.** Reported runtimes are averaged over 10 iterations. Method abbreviations are the same as Table 3.1. The right most column is the number of elements generated in the complete product before the final selection.

As can be seen in Table 3.3, there can be a massive penalty from using a naive implementation over the optimal one. This highlights the importance of not treating parameters as constants in the algorithmic analysis.

3.3 Selection on $X + Y$ as a function of α

Here, we present the runtime results of selection on $X + Y$ using Serang's method [6], but using the optimal algorithm for the LOHification step. For all of our plots $|X| = |Y| = 10,000,000$. These plots show that the α computed using the formula derived in Theorem 2.2.1 is a fair estimate of the optimal α and performs much better than $\alpha = 2$ in many cases.

The "fan-out" as α gets larger is an effect of the overshoot. A larger α means larger layer products which means a faster runtime if the overshoot is near optimal, but a larger penalty if the overshoot is near pessimal.

In Figure 3.1, where $n = k$, there does not seem to be much of a benefit for using a small α, but looking at Figures 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7; we can see that as the ratio of k to n grows, the optimal α approaches 1.0 both theoretically and empirically.

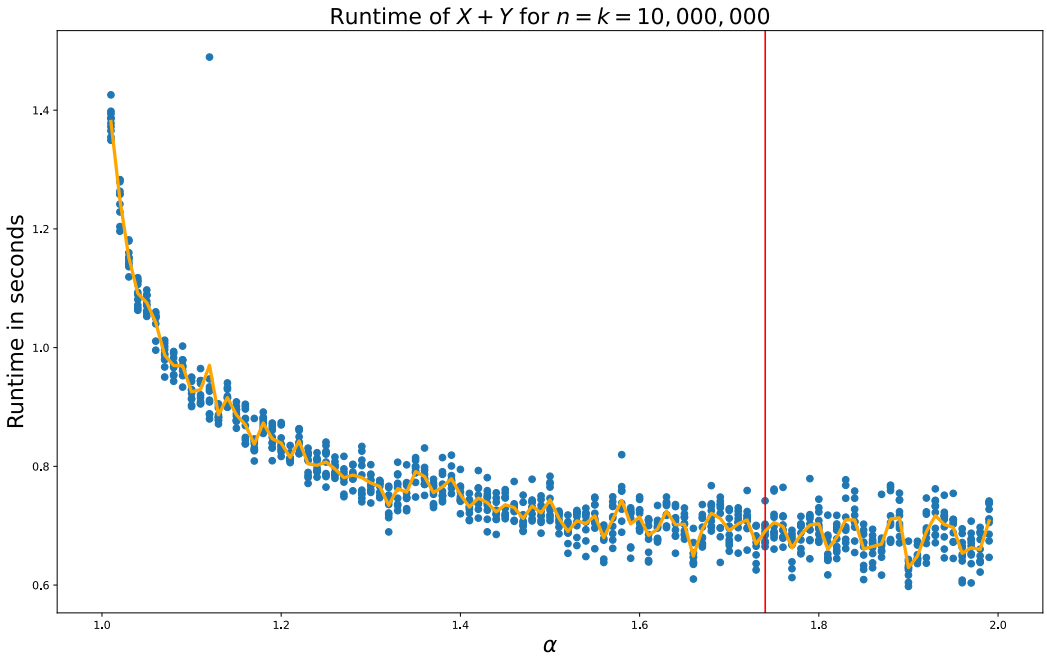

Figure 3.1 **Plot of runtime for selection on $X + Y$ for $n = k = 10,000,000$.** The plot captures the runtimes of selecting the minimum 10,000,000 elements from two arrays of length 10,000,000. The arrays are filled with 64-bit floating point numbers drawn from the C++ rand() function. The values of α tested range from 1.01 to 1.99 (inclusive) in increments of 0.01 with ten trials for every value of α tested. Each trial used a different random seed. The red line represents the optimal value of α calculated using the formula derived in Theorem 2.2.1. The orange line represents the average runtime for a given α.

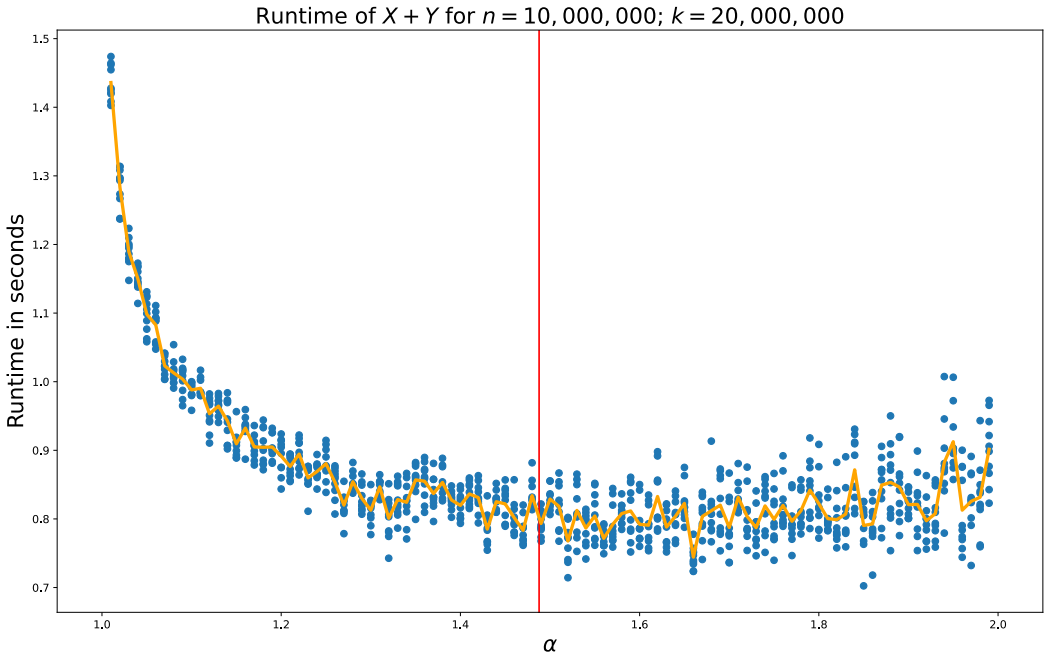

Figure 3.2 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 20,000,000$.** All other aspects of its construction are identical to Figure 3.1.

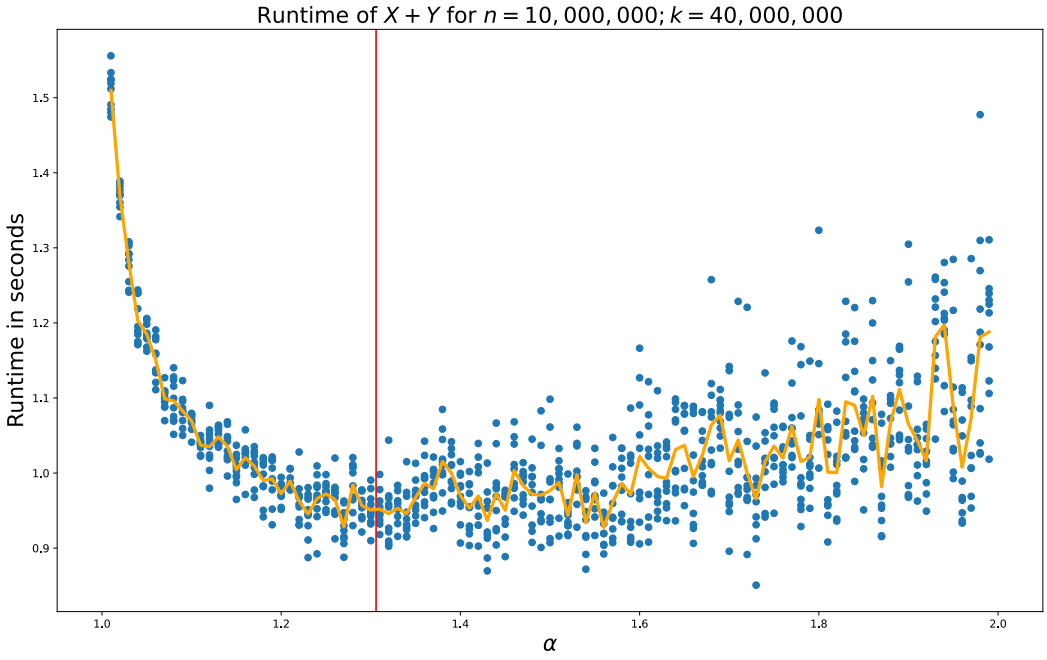

Figure 3.3 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 40,000,000$.** All other aspects of its construction are identical to Figure 3.1.

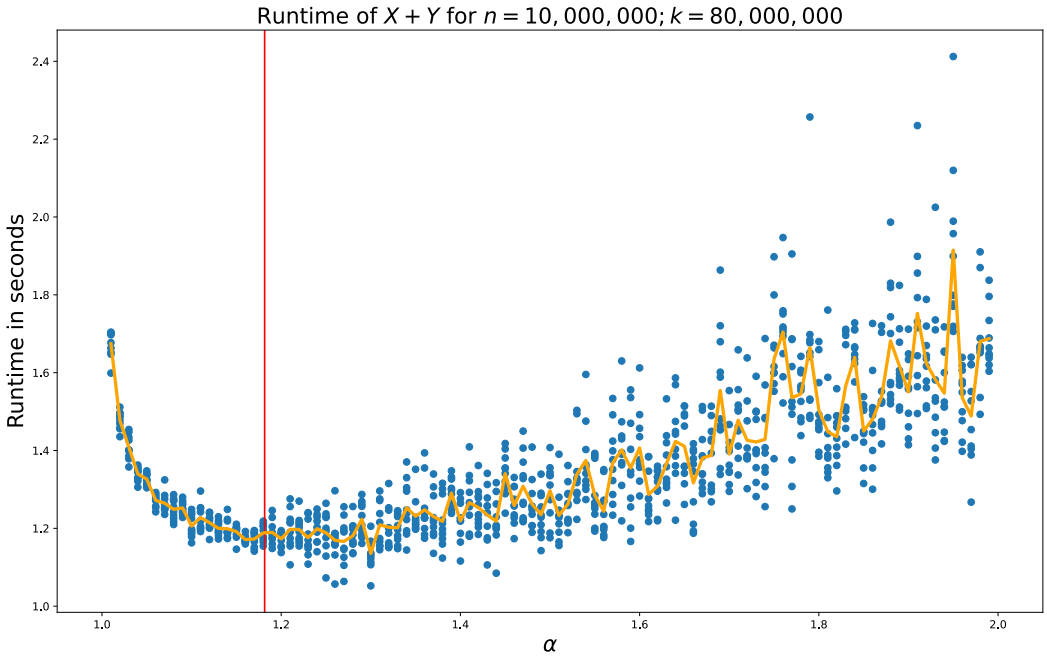

Figure 3.4 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 80,000,000$.** All other aspects of its construction are identical to Figure 3.1.

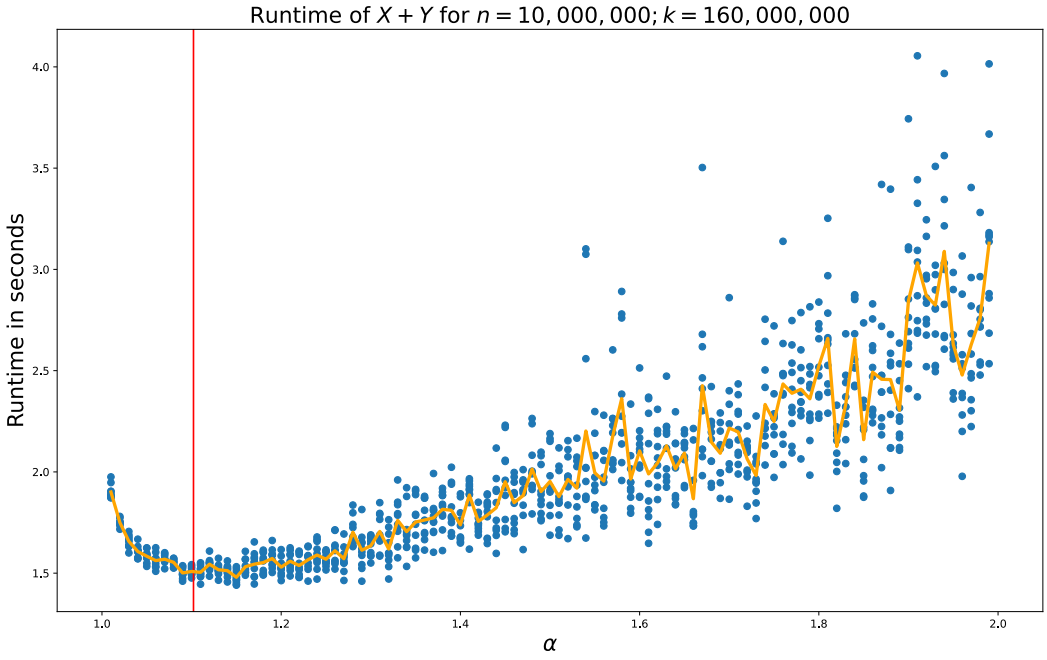

Figure 3.5 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 160,000,000$.** All other aspects of its construction are identical to Figure 3.1.

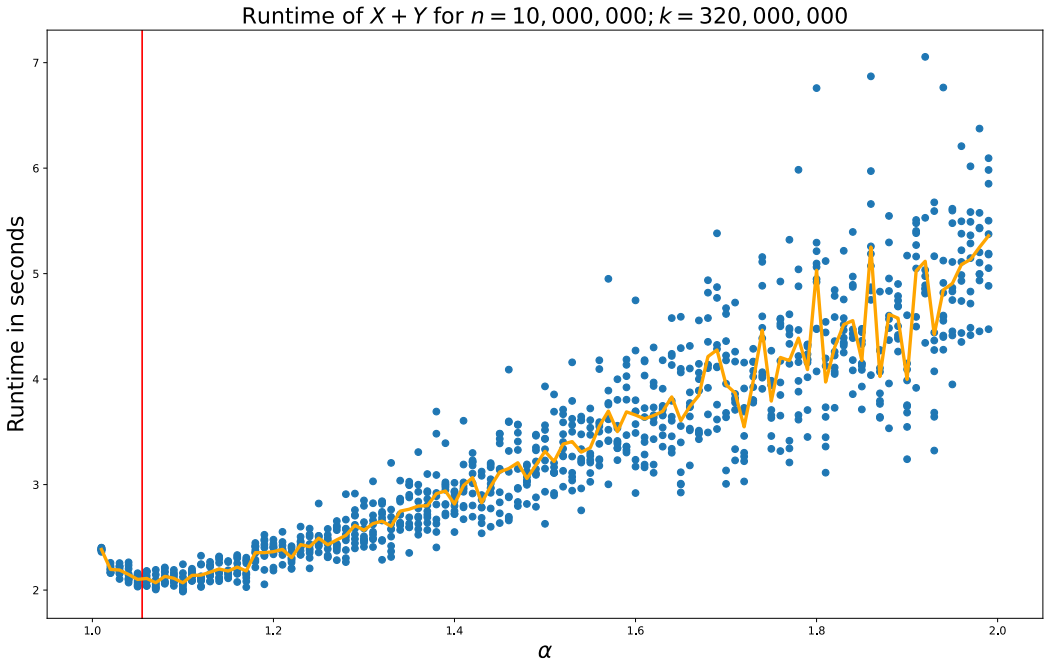

Figure 3.6 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 320,000,000$.** All other aspects of its construction are identical to Figure 3.1.

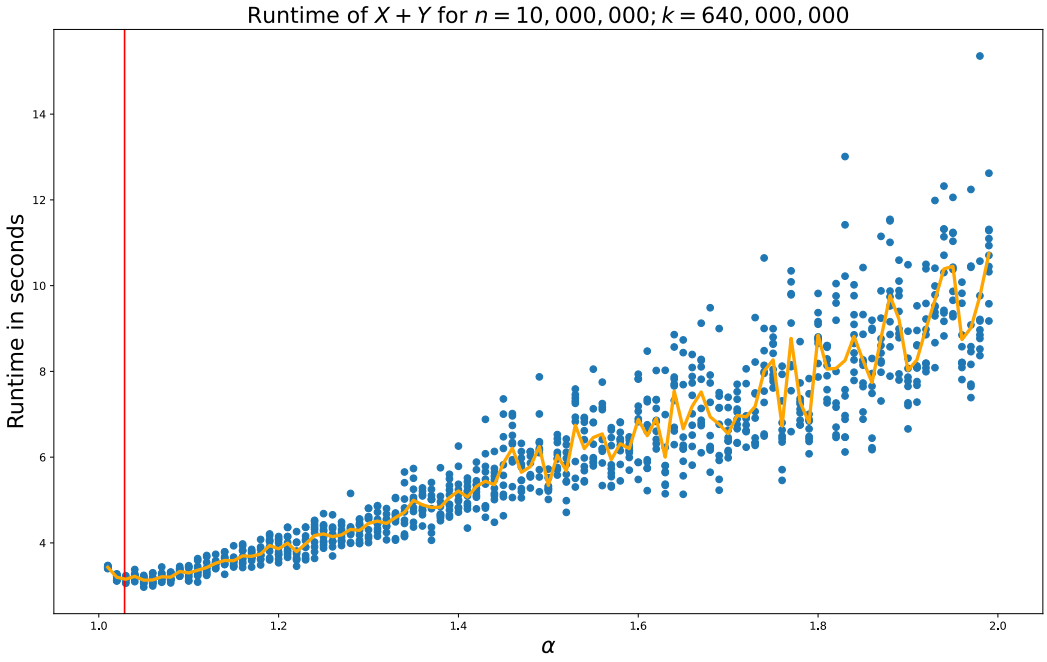

Figure 3.7 **Plot of runtime for selection on $X + Y$ for $n = 10,000,000; k = 640,000,000$.** All other aspects of its construction are identical to Figure 3.1.

3.4 NeutronStar v. Isospec

The sort-based algorithm for top-k on a multinomial described in Methods is implemented in the open-source isotopologue calculator, NeutronStar [9]. Particularly, it is used to calculate the most likely configurations of sub-isotopologues, or the parts of a compound consisting of the same element (e.g. the sub-isotopologues of fructose, $C_6H_{12}O_6$, are C_6, H_{12}, and O_6). These sub-isotopologue configurations are generated in layers that are sent to a cartesian product tree that the top k elements are selected from using the Kreitzberg algorithm for top-k on $X_1 + X_2 + \cdots + X_m$ [7].

Isospec is a different open-source isotopologue calculator [18]. It uses an approximation based on the Central Limit Theorem to compute probablistic bounds on which parts of a multinomial are the most abundant in the top p where p is a fraction of the total aubundance. This differs from NeutronStar in several ways. Isospec has no option to generate the top k configurations, but NeutronStar includes an option to generate the top p as well as the top k. NeutronStar does not rely on Central Limit Theorem and thus does not overshoot values generated in the multinomial, wheras Isospec does not sort the values generated in the multinomial and thus is not bound by the complexity of sorting.

In Table 3.4, we compare the runtimes of both NeutronStar and Isospec. From this, we can see that, NeutronStar performs well in practice, even with the complexity bound on sorting the elements in the multinomial.

Compound	p	k	IsoSpec	NeutronStar α=1.01	NeutronStar α=1.1
Averagine	0.1	698,668	0.0821208	0.0290156	0.026539
	0.3	3958,459	0.132122	0.114437	0.105707
	0.5	11,442,227	0.965732	0.285948	0.272625
	0.7	30,264,581	1.40907	0.682967	0.608624
	0.9	110,437,547	3.96185	2.33534	1.99549
Ostalloy	0.1	6719,141	1.64834	0.121507	0.118957
	0.3	65,366,950	3.18777	0.936333	1.03014
	0.5	279,712,408	5.89366	3.86705	4.15506
Palladium	0.1	9,134	0.0094108	0.0023928	0.0018632
alloy Pgc	0.3	52,855	0.0110916	0.0070442	0.0076462
	0.5	162,857	0.0483886	0.0142928	0.0158198
	0.7	473,917	0.076751	0.0309766	0.030281
	0.9	2,074,266	0.177315	0.0877584	0.0938606
	0.99	13,466,926	0.399824	0.40483	0.372335
	0.999	47,409,787	1.84854	1.13571	1.09194
$Sn_{20}Xe_{20}Nd_{20}Dy_{20}$	1e-12	1	0.0001034	9.34e-05	8.88e-05
	1e-11	5	17.7561	0.0001052	0.0001
	1e-10	50	16.2101	0.000154	0.0001682
	1e-9	554	–	0.000473	0.0003332
	1e-7	72,222	–	0.0088738	0.0034802
	1e-5	13,415,245	–	0.240625	0.264156

Table 3.4 **Table of runtimes (seconds) for generating the k most abundant isotopologues of several large compounds using NeutronStar and Isospec.** These times are an average of 10 runs. p is the fraction of the total abundance that is represented by the top k most abundant configurations. – indicates the program ran out of memory.

CHAPTER 4 Discussion

Due to the $\Omega(n \log(n))$ bound on comparison-based sorting, ordering values using only pairwise comparison is generally considered to be an area for little practical performance benefit; however, LOHs have provided significant performance benefits, both theoretically and empirically, when replacing sorting in applications where sorting is a limiting factor. Optimal LOHify for any α has been used to replace sorting in applications such as finding the most abundant isotopologue peaks of a compound [9] and generating the top k elements in a Cartesian product of two [6] or more [7] arrays (fast in practice with $1 < \alpha \ll 2$). Quick-LOHify has improved performance in finding the score at which a desired FDR threshold occurs [8].

For the algorithms that perform a top-k on a Cartesian product using LOHs, the α parameter has a deeper impact on more than just the LOHification step. Specifically, it affects the upper bound on how far we can "overshoot" the actual k which affects the time it takes to perform a one-dimensional selection. The effect this "overshooting" has on the runtime can be seen in Table 3.3, specifically at $\alpha = 1.9$ where the actual number of elements generated is closer to the pessimal. Now, we can to dynamically choose a good α at runtime for top-k on $X + Y$. The optimal α for top-k on a Cartesian product tree is difficult to compute, but the runtime of optimal LOHify is needed to solve for it.

The theoretic algorithm presented for top-k on a multinomial demonstrates that an $O(km^2)$ complexity can be achieved. While it is unlikely to be efficient in practice, it

should help pave the way for an efficient, LOH-based algorithm that achieves similar bounds. We should be able to do this by selecting elements from the multinomial in layers. In the future, we should also be able to dynamically choose a good α for every extant algorithm that uses LOHs, and we should be able to apply LOHs to problems where sorting is a significant part of the runtime and/or theoretical complexity.

BIBLIOGRAPHY

[1] L.R. Ford Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, 1959.

[2] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1968–.

[3] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, 47(6):1012–1027, 2000.

[4] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[5] H. Kaplan, L. Kozma, O. Zamir, and U. Zwick. Selection from heaps, row-sorted matrices and $X + Y$ using soft heaps. *Symposium on Simplicity in Algorithms*, pages 5:1–5:21, 2019.

[6] O Serang. Optimally selecting the top k values from X + Y with layer-ordered heaps. *PeerJ Computer Science*, 7:e501, 2021.

[7] P. Kreitzberg, K. Lucke, J. Pennington, and O. Serang. Selection on $X_1 + X_2 + \cdots + X_m$ via Cartesian product trees. *PeerJ Computer Science*, 7:e483, 2021.

[8] K. Lucke, J. Pennington, P. Kreitzberg, L. Käll, and O. Serang. Performing selection on a monotonic function in lieu of sorting using layer-ordered heaps. *Journal of Proteome Research*, 20(4):1849–1854, 2021.

[9] P. Kreitzberg, J. Pennington, K. Lucke, and O. Serang. Fast exact computation of the k most abundant isotope peaks with layer-ordered heaps. *Analytical Chemistry*, 92(15):10613–10619, 2020.

[10] P. Kreitzberg and O. Serang. On solving probabilistic linear Diophantine equations. *jmlr*, 2021.

[11] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[12] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, Semptember 1980.

[13] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, May 1998.

[14] P. Kreitzberg, K. Lucke, and O. Serang. Selection on $X_1 + X_2 + \cdots + X_m$ with layer-ordered heaps. *arXiv preprint arXiv:1910.11993*, 2020.

[15] H. M. Finucan. The mode of a multinomial distribution. *Biometrika*, 51(3/4):513–517, 1964.

[16] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.

[17] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society B*, 57:289–300, 1995.

[18] M. K. Łącki, M. Startek, D. Valkenborg, and A. Gambin. Isospec: Hyperfast fine structure calculator. *Analytical Chemistry*, 89(6):3272–3277, 2017.