Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Average Case Considerations for MergeInsertion

Florian Stober

Course of Study:	Informatik
Examiner:	Prof. Dr. Volker Diekert
Supervisor:	Dr. Armin Weiß
Commenced:	June 4, 2018
Completed:	December 4, 2018

Abstract

The MergeInsertion Algorithm, also known as Ford-Johnson Algorithm, is a sorting algorithm that was discovered by Ford and Johnson in 1959. It was later described by Knuth as MergeInsertion. The algorithm can be divided into three steps: First pairs of elements are compared. then the larger half is sorted using MergeInsertion, and last the remaining elements are inserted. The most interesting property of this algorithm is the number of comparisons it requires, which is close to the information-theoretic lower bound. While the worst-case behavior is well understood, only little is known about the average-case. This thesis takes a closer look at the average case behavior. An upper bound of $n \log n - 1.4005n + o(n)$ is established. For small n the exact values are calculated. Furthermore the impact of different approaches to binary insertion on the number of comparisons is explored. To conclude we perform some experiments to evaluate different approaches on improving MergeInsertion.

Contents

1 Introduction

Sorting a set of elements is an important operation frequently performed by many computer programs. Consequently there exist a variety of algorithms for sorting, each of which comes with its own advantages and disadvantages.

This thesis studies a specific sorting algorithm known as MergeInsertion. It was discovered by Ford and Johnson in 1959[JJ59]. Before D. E. Knuth coined the term MergeInsertion in his study of the algorithm in his book ‘‘The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching’’[Knu98] it was known only as Ford-Johnson Algorithm, named after its creators.

The one outstanding property of MergeInsertion is that the number of comparisons it requires is close to the information-theoretic lower bound. This sets it apart from many other sorting algorithms.

MergeInsertion can be described in three steps: First pairs of elements are compared. In the second step the larger half is sorted recursively. And as a last step the elements belonging to the smaller half are inserted into the already sorted larger half using binary insertion.

The goal of this thesis is to study the number of comparison required in the average case. Currently the best upper bound on the average case is $n \log n - 1.3999n + o(n)$ [EW13]. We aim to improve upon that by analyzing the insertion step of MergeInsertion in greater detail. In general MergeInsertion achieves its good performance by inserting elements in a specific order that in the worst case causes each element to be inserted into a sorted list of $2^k - 1$ elements. When looking at the average case elements are often inserted into less than $2^k - 1$ elements which is slightly cheaper. By calculating those small savings we seek to achieve our goal of a better upper bound on the average case.

Speaking of binary insertion, when an element is inserted less than $2^k - 1$ there are some positions, it can be inserted into with only $k - 1$ instead of k comparisons. Binary Insertion determines the position where the element will be inserted by comparing it to the middle element of the list. In the case of MergeInsertion not all positions are not equally likely so this approach might not be optimal. We investigate other approaches to binary insertion to see if it is possible to use the non-uniform distribution to our advantage.

While it is nice to have an approximation of the number of comparisons required by MergeInsertion, it is also nice to calculate the exact numbers. We present a computer program for efficiently calculating those for small values of n.

Last we perform experiments to evaluate different ideas for improving MergeInsertion. This includes our own ideas as well as those proposed by others.

1.1 Related Work

One of the deficiencies of MergeInsertion is the oscillating number of comparisons it requires. There are points at which it reaches its optimum, but in between are lots of points were it requires more comparisons. Glenn K. Manacher showed in [Man79] that MergeInsertion can be beaten for some n by splitting the sequence in two parts that are both optimal(or close to optimal) for MergeInsertion, sorting both of these with MergeInsertion and then merging the two sorted lists. This was improved upon by [BT85] using a more efficient merging algorithm. While this work was done for the worst-case, as part of our experiments we briefly look at how it can improve the average-case.

Regarding the average case we find that in his studies D. Knuth[Knu98] calculated the number of comparisons required in the average case for small n. An upper bound of $n \log n - 1.3999n + o(n)$ has been discovered by [EW13].

Most recently Iwama and Teruyama showed that MergeInsertion can be improved by combining it with their 1-2-Insertion algorithm resulting in an upper bound of $n \log n - 1.4106n + O(\log n)$[IT17].

2 Average Case Analysis

In this chapter we take a look at how MergeInsertion works, step by step making our way towards a better approximation. First we revisit the three phases of the algorithm. Accompanying the explanations is an example where $n = 21$,

1. **Pairwise comparison**. Pairs of elements are compared, such that there are now a_1 to $a_{\lfloor \frac{n}{2} \rfloor}$ and b_1 to $b_{\lceil \frac{n}{2} \rceil}$ with $\forall_{1 \leq i \leq \lfloor \frac{n}{2} \rfloor} a_i > b_i$. The result of this step is shown in Figure 2.1. We call such a set of relations between individual elements a configuration.

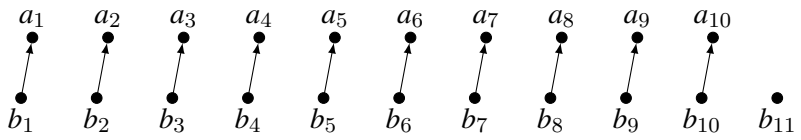


Figure 2.1: Step 1: Pairwise comparison.

2. **Recursion**. The $\lfloor \frac{n}{2} \rfloor$ larger elements, i.e. a_1 to $a_{\lfloor \frac{n}{2} \rfloor}$ are sorted recursively. Then the elements are renamed such that $a_i < a_{i+1}$ and $a_i > b_i$ still holds. This configuration can be seen in Figure 2.2.

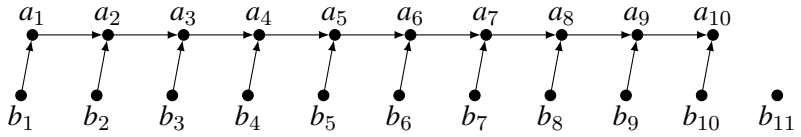


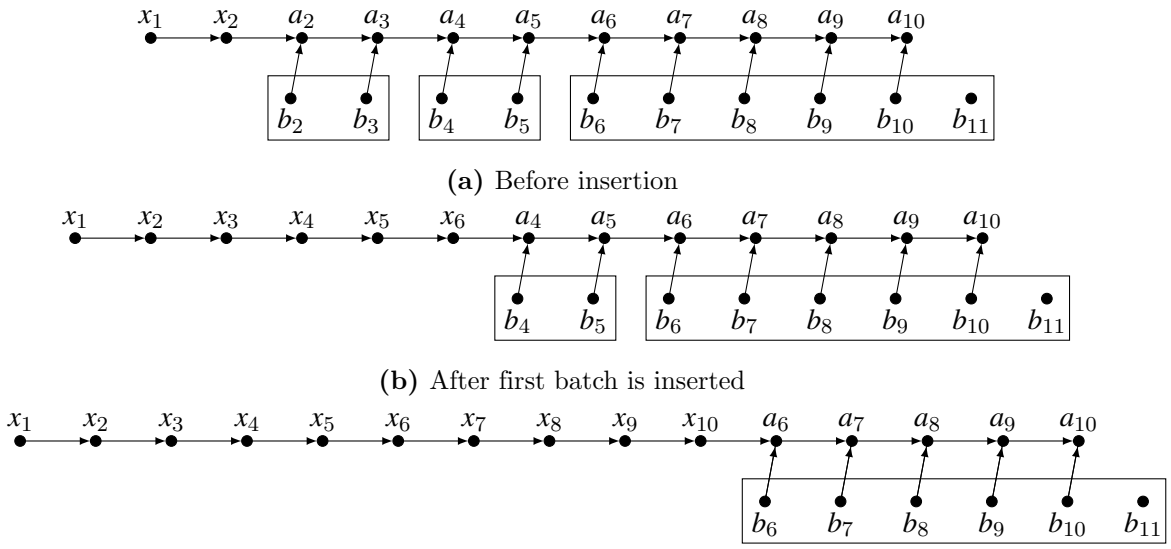
Figure 2.2: Step 2: Recursively sort a_i.

3. **Insertion**. The $\lceil \frac{n}{2} \rceil$ small elements, i.e. the b_i are inserted into the main chain using binary insertion. The term ‘‘main chain’’ is due to Knuth and describes the set of elements containing $x_1, x_2, ..., x_{2t_{k-2}}$ and $a_{t_{k-1}+1}, a_{t_{k-1}+2}, ..., a_{t_k}$ as well as the b_i that have already been inserted into it. It is called main chain because the total order of all elements in the main chain is known.

An important part of the MergeInsertion algorithm is the order in which the b_i are inserted into the main chain. The elements are inserted in batches starting with b_3, b_2. The k-th batch contains the elements b_{t_k}, b_{t_k-1}, ..., $b_{t_{k-1}+1}$, which are inserted in that order. Elements b_j where $j > \lceil \frac{n}{2} \rceil$ are left out[1]. t_k is defined as $t_k = \frac{2^{k+1}+(-1)^k}{3}$.

After inserting all elements of the k-th batch the elements a_1 to a_{t_k} and b_1 to b_{t_k} of which we now know the total order are renamed to x_1 to x_{2t_k} such that $x_i < x_{i+1}$.

As a result of the insertion order an element b_i which is part of the k-th batch can be inserted using at most k comparisons.



(a) Before insertion

(b) After first batch is inserted

(c) After second batch is inserted

Figure 2.3: Step 3: Insertion. The different batches are outlined with boxes. The batches are processed from left to right but elements within each batch are inserted from right to left.

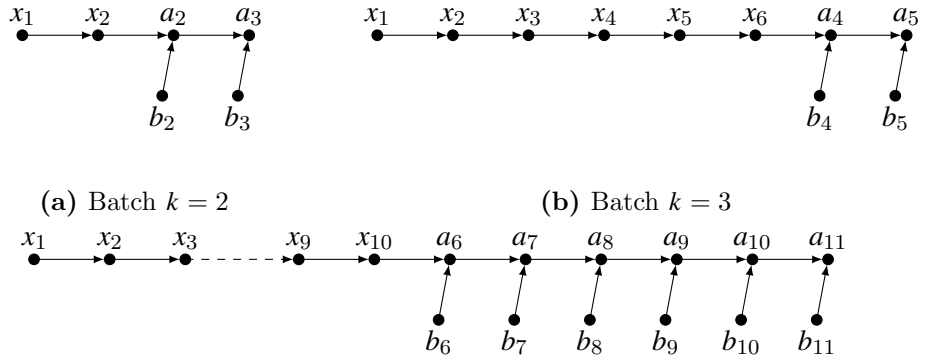
Since the first step always requires $\lfloor \frac{n}{2} \rfloor$ comparisons and the recursion step does not do any comparisons by itself but only depends on the other steps we shall put our focus on the insertion step to find out more about the average case.

2.1 Lots of Numbers

In this section we have a look at different probabilities when inserting one batch of elements, i.e. the elements b_{t_k} to $b_{t_{k-1}+1}$. For each element b_i we want to be able to answer the following questions: What is the probability of b_i being inserted between x_j and x_{j+1}. And what is the probability of b_i being inserted into a specific number of elements.

[1] When sorting only 17 instead of 21 elements as in the example, the last batch would be incomplete. Out of the elements $b_{11}, b_{10}, ..., b_6$ only b_9, b_8, b_7 and b_6 actually exists. In that case the non-existent elements are skipped and the remaining elements, b_9, b_8, b_7 and b_6, are inserted in that order.

When analyzing what happens when inserting one batch of elements we make the following assumptions: All elements of previous batches, i.e. b_1 to $b_{t_{k-1}}$ have already been inserted and together with the corresponding a_i they constitute the main chain and have been renamed to x_1 to $x_{2t_{k-1}}$. Batches that are inserted after the batch we are looking at are ignored, since those do not affect the probabilities we want obtain.



(a) Batch $k = 2$

(b) Batch $k = 3$

(c) Batch $k = 4$

Figure 2.4: Batches of the elements b_{t_k} to $b_{t_{k-1}+1}$ for $k \in \{2, 3, 4\}$

First we define a probability space for the process of inserting one batch of elements. Ω_k is the set of all possible outcomes when sorting the partially ordered elements shown in Figure 2.4 by inserting b_{t_k} to $b_{t_{k-1}+1}$. Each $\omega \in \Omega_k$ is a function that maps an element e_i to its final position, i.e. $\omega(e_i) \in \{1, 2, ..., 2t_k\}$. To simplify things we define $x_{t_{k-1}+j} := a_j$ for all $j \in \{t_{k-1}, t_{k-1} + 1, ..., t_k\}$.

While the algorithm mandates a specific order for inserting the elements $b_{t_{k-1}+1}$ to b_{t_k} during the insertion step, using a different order does not change the outcome, i.e. the elements are still sorted correctly. For this reason we can assume a different insertion in order to simplify calculating the size of the probability space as well as the likelihood of relations between individual elements.

So far things are still simple as we have $P(\omega) = \frac{1}{|\Omega_k|}$. Now we only need to calculate the size of Ω_k to know the probability $P(\omega)$. First we have a look at $k = 2$. The corresponding configuration is shown in Figure 2.4a. It can be seen that when inserting b_2 first there are 3 positions it can go, then when inserting b_3 there are 5 positions it can go. The result is $|\Omega_2| = 3 \cdot 5 = 15$. For $k = 3$ and $k = 4$ we calculate $|\Omega_3| = 7 \cdot 9 = 63$ and $|\Omega_4| = 11 \cdot 13 \cdot 15 \cdot 17 \cdot 19 \cdot 21 = 14549535$ respectively.

When looking at an arbitrary k, the first element of the batch $b_{t_{k-1}+1}$ is inserted into $2t_{k-1}$ elements (x_1 to $x_{2t_{k-1}}$) thus there are $2t_{k-1} + 1$ positions it can go. The second element $b_{t_{k-1}+2}$ is then inserted into $2t_{k-1}+2$ elements (x_1 to $x_{2t_{k-1}}$, $a_{t_{k-1}+1}$ and $b_{t_{k-1}+1}$) i.e. $2t_{k-1}+3$

i	1	2	3	4	5	6
$P(X_i = 0)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 1)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 2)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 3)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 4)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 5)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 6)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 7)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 8)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 9)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 10)$	$\frac{1}{11}$	$\frac{1}{11} \cdot \frac{12}{13}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{18}{19}$	$\frac{1}{11} \cdot \frac{12}{13} \cdot \frac{14}{15} \cdots \frac{20}{21}$
$P(X_i = 11)$	0	$\frac{1}{13}$	$\frac{1}{13} \cdot \frac{14}{15}$	$\frac{1}{13} \cdot \frac{14}{15} \cdot \frac{16}{17}$	$\frac{1}{13} \cdot \frac{14}{15} \cdot \frac{16}{17} \cdot \frac{18}{19}$	$\frac{1}{13} \cdot \frac{14}{15} \cdot \frac{16}{17} \cdots \frac{20}{21}$
$P(X_i = 12)$	0	0	$\frac{1}{15}$	$\frac{1}{15} \cdot \frac{16}{17}$	$\frac{1}{15} \cdot \frac{16}{17} \cdot \frac{18}{19}$	$\frac{1}{15} \cdot \frac{16}{17} \cdot \frac{18}{19} \cdot \frac{20}{21}$
$P(X_i = 13)$	0	0	0	$\frac{1}{17}$	$\frac{1}{17} \cdot \frac{18}{19}$	$\frac{1}{17} \cdot \frac{18}{19} \cdot \frac{20}{21}$
$P(X_i = 14)$	0	0	0	0	$\frac{1}{19}$	$\frac{1}{19} \cdot \frac{20}{21}$
$P(X_i = 15)$	0	0	0	0	0	$\frac{1}{21}$

Table 2.1: Values of $P(X_i = j)$ for $k = 4$.

positions. This continues up to b_{t_k} which is inserted into $2t_k - 2$ elements, thus there are $2t_k - 1$ position where it can go. This leads to the following formula:

$$\begin{aligned}
|\Omega_k| &= (2t_{k-1} + 1) \cdot (2t_{k-1} + 3) \quad \cdots \quad (2t_k - 1) \\
&= \prod_{i=1}^{t_k - t_{k-1}} (2t_{k-1} + 2i - 1) \\
&= \left(\prod_{i=2t_{k-1}+1}^{2t_k - 1} i \right) \left(\prod_{i=t_{k-1}+1}^{t_k - 1} (2i) \right)^{-1} \\
&= \left(\prod_{i=1}^{2t_k - 1} i \right) \left(\prod_{i=1}^{2t_{k-1}} i \right)^{-1} \left(\prod_{i=1}^{t_k - 1} 2i \right)^{-1} \left(\prod_{i=1}^{t_{k-1}} 2i \right) \\
&= 2^{t_{k-1} - t_k + 1} \cdot \frac{(2t_k - 1)!}{(2t_{k-1})!} \cdot \frac{(t_{k-1})!}{(t_k - 1)!}
\end{aligned} \tag{2.1}$$

Next we have a look at where an element will end up after it has been inserted. Not all positions are equally likely.

For this purpose we define the random variable

$$X_i := \omega \to \begin{cases} 0 & \text{if } \omega(b_{t_{k-1}+i}) < \omega(x_1) \\ j & \text{if } \omega(x_j) < \omega(b_{t_{k-1}+i}) < \omega(x_{j+1}) \quad \text{where } j \in \{1, 2, ..., 2^k - 2\} \\ 2^k - 1 & \text{if } \omega(x_{2^k - 1}) < \omega(b_{t_{k-1}+i}). \end{cases} \tag{2.2}$$

Now we want to have a look at the probabilities $P(X_i = j)$. For $k = 4$ these are given in Table 2.1. The values for $P(X_i = j)$ follow a simple pattern.

For an arbitrary k we can calculate them with the following recursive scheme. We start with $P(X_1 = j)$. This corresponds to the insertion of $b_{t_{k-1}+1}$ into $x_1, ..., x_{2t_{k-1}}$. The probability of all those is uniformly distributed, so $P(X_1 = j) = \frac{1}{2t_{k-1}+1}$ for $0 \leq j \leq 2t_{k-1}$.

For $i > 1$ we can express $P(X_i = j)$ in terms of $P(X_{i-1} = j)$. Observe that when inserting $b_{t_{k-1}+i}$ there are $2t_{k-1} + 2i - 2$ elements known to be smaller than $a_{t_{k-1}+i}$. These are $x_1, ..., x_{2t_{k-1}}$ and $a_{t_{k-1}+1}, ..., a_{t_{k-1}+i-1}$ as well as the corresponding b's. The number of elements known to be smaller than $a_{t_{k-1}+i-1}$ is one less: just $2t_{k-1} + 2i - 3$. As a result the probability that $b_{t_{k-1}+i}$ is inserted between $a_{t_{k-1}+i-1}$ and $a_{t_{k-1}+i}$ is $P(X_i = 2t_{k-1} + i - 1) = \frac{1}{2t_{k-1}+2i-1}$. The probability that is ends up in one of the other positions consequently is $P(0 \leq X_i < 2t_{k-1} + i - 1) = \frac{2t_{k-1}+2i-2}{2t_{k-1}+2i-1}$. If we know that $b_{t_{k-1}+i}$ is inserted into one of those other positions, then it is inserted into exactly the same elements as $b_{t_{k-1}+i-1}$, thus we can write $P(X_i = j) = \frac{2t_{k-1}+2i-2}{2t_{k-1}+2i-1} P(X_{i-1} = j)$. This leads to Equation (2.3).

$$P(X_i = j) = \omega \rightarrow \begin{cases} \left(\prod_{l=1}^{i-1} 2t_{k-1} + 2l\right) \cdot \left(\prod_{l=1}^{i} 2t_{k-1} + 2l - 1\right)^{-1} & \text{if } 0 \leq j \leq 2t_{k-1} \\ \left(\prod_{l=j-2t_{k-1}+1}^{i-1} 2t_{k-1} + 2l\right) \cdot \left(\prod_{l=j-2t_{k-1}+1}^{i} 2t_{k-1} + 2l - 1\right)^{-1} & \text{if } 2t_{k-1} < j < 2t_{k-1} + i \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

While these products are intuitive they look quite ugly, so we simplify the above formula starting with the first case.

$$\begin{aligned} &\left(\prod_{l=1}^{i-1} 2t_{k-1} + 2l\right) \cdot \left(\prod_{l=1}^{i} 2t_{k-1} + 2l - 1\right)^{-1} \\ &= \left(\prod_{l=t_{k-1}+1}^{t_{k-1}+i-1} 2l\right) \cdot \left(\prod_{l=2t_{k-1}+1}^{2t_{k-1}+2i-1} l\right)^{-1} \cdot \left(\prod_{l=t_{k-1}+1}^{t_{k-1}+i-1} 2l\right) \\ &= \left(\prod_{l=1}^{t_{k-1}+i-1} 2l\right) \cdot \left(\prod_{l=1}^{t_{k-1}} 2l\right)^{-1} \cdot \left(\prod_{l=1}^{2t_{k-1}+2i-1} l\right)^{-1} \cdot \left(\prod_{l=1}^{2t_{k-1}} l\right) \cdot \left(\prod_{l=1}^{t_{k-1}+i-1} 2l\right) \cdot \left(\prod_{l=1}^{t_{k-1}} 2l\right)^{-1} \\ &= 2^{2i-2} \left(\frac{(t_{k-1} + i - 1)!}{(t_{k-1})!}\right)^2 \frac{(2t_{k-1})!}{(2t_{k-1} + 2i - 1)!} \end{aligned} \tag{2.4}$$

For the second case we have

$$
\begin{aligned}
&\left(\prod_{l=j-2t_{k-1}+1}^{i-1} 2t_{k-1}+2l\right)\cdot\left(\prod_{l=j-2t_{k-1}+1}^{i}(2t_{k-1}+2l-1\right)^{-1}\\
&=\left(\prod_{l=j-t_{k-1}+1}^{t_{k-1}+i-1} 2l\right)\cdot\left(\prod_{l=2j-2t_{k-1}+1}^{2t_{k-1}+2i-1} l\right)^{-1}\cdot\left(\prod_{l=j-t_{k-1}+1}^{t_{k-1}+i-1} 2l\right)\\
&=\left(\prod_{l=1}^{t_{k-1}+i-1} 2l\right)\cdot\left(\prod_{l=1}^{j-t_{k-1}} 2l\right)^{-1}\cdot\left(\prod_{l=1}^{2t_{k-1}+2i-1} l\right)^{-1}\cdot\left(\prod_{l=1}^{2j-2t_{k-1}} l\right)\cdot\left(\prod_{l=1}^{t_{k-1}+i-1} 2l\right)\cdot\left(\prod_{l=1}^{j-t_{k-1}} 2l\right)^{-1}\\
&=2^{4t_{k-1}-2j+2i-2}\left(\frac{(t_{k-1}+i-1)!}{(j-t_{k-1})!}\right)^2\frac{(2j-2t_{k-1})!}{(2t_{k-1}+2i-1)!}
\end{aligned}
\tag{2.5}
$$

By substitution of (2.4) and (2.5) in (2.3) we obtain (2.6) which already looks a lot nicer.

$$
P(X_i=j)=\omega\rightarrow\begin{cases}
2^{2i-2}\left(\frac{(t_{k-1}+i-1)!}{(t_{k-1})!}\right)^2\frac{(2t_{k-1})!}{(2t_{k-1}+2i-1)!} & \text{if } 0\le j\le 2t_{k-1}\\
2^{4t_{k-1}-2j+2i-2}\left(\frac{(t_{k-1}+i-1)!}{(j-t_{k-1})!}\right)^2\frac{(2j-2t_{k-1})!}{(2t_{k-1}+2i-1)!} & \text{if } 2t_{k-1}<j<2t_{k-1}+i\\
0 & \text{otherwise}
\end{cases}
\tag{2.6}
$$

Next we want to look at the probability that an b_i is inserted into a particular number of elements. For that purpose we define the random variable

$$
Y_i:\omega\rightarrow\left|\left\{v\mid\omega(w)<\omega(a_{t_{k-1}+i})\wedge v\notin\{b_{t_{k-1}+1},b_{t_{k-1}+2},...,b_{t_{k-1}+i}\}\right\}\right| \tag{2.7}
$$

Theorem 1
The probability $P(Y_i=j)$, that $b_{t_{k-1}+i}$ is inserted into j elements is given by

$$
P(Y_i=j)=\begin{cases}
\frac{(2t_k-i-j-1)!}{2^{2^k-j-1}(j-2t_{k-1}-i+1)!(2^k-j-1)!}2^{t_k-t_{k-1}-i}\frac{(i+j)!}{(2t_k-1)!}\frac{(t_k-1)!}{(t_{k-1}+i-1)} & \text{if } 1\le i\le t_k-t_{k-1}\\
0 & \text{and } 2t_{k-1}+i-1\le j\le 2^k-1\\
 & \text{otherwise}
\end{cases}
\tag{2.8}
$$

To proof 2.8 we introduce another random variable

$$
\tilde{Y}_{i,q}:\omega\rightarrow\left|\left\{v\mid\omega(w)<\omega(a_{t_{k-1}+i})\wedge v\in\{b_{t_{k-1}+i+1},...,b_{t_{k-1}+i+q}\}\right\}\right| \tag{2.9}
$$

Informally speaking $\tilde{Y}_{i,q}$ is the number of elements in $\{b_{t_{k-1}+i+1},...,b_{t_{k-1}+i+q}\}$ that are inserted before $a_{t_{k-1}+i}$.

The probability $P(\tilde{Y}_{i,q} = j)$ is described by the following equation[2] which we will proof below.

$$P(\tilde{Y}_{i,q} = j) = \frac{(2q-j)!}{2^{q-j}j!(q-j)!}2^q \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-1)!}\frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \tag{2.10}$$

From the definition of $\tilde{Y}_{i,q}$ we can see that $0 \leq \tilde{Y}_{i,q} \leq q$ thus $P(\tilde{Y}_{i,0} = 0) = 1$. This also holds for Equation (2.10).

$$P(\tilde{Y}_{i,0} = 0) = \frac{0!}{2^0 \cdot 0! \cdot 0!}2^0 \frac{(2t_{k-1}+2i-1)!}{(2t_{k-1}+2i-1)!}\frac{(t_{k-1}+i-1)!}{(t_{k-1}+i-1)!} = 1 \tag{2.11}$$

Now if $\tilde{Y}_{i,q} = j$ there are two possibilities:

1. $\tilde{Y}_{i,q-1} = j-1$ and $X_{i+q} < 2t_{k-1}+i$. Informally speaking that means out of $\{b_{t_{k-1}+i+1}, ..., b_{t_{k-1}+i+q-1}\}$ there have been $j-1$ elements inserted before $a_{t_{k-1}+i}$ and $b_{t_{k-1}+i+q}$ is inserted before $a_{t_{k-1}+i}$.
2. $\tilde{Y}_{i,q-1} = j$ and $X_{i+q} \geq 2t_{k-1}+i$. Informally speaking that means out of $\{b_{t_{k-1}+i+1}, ..., b_{t_{k-1}+i+q-1}\}$ there have been j elements inserted before $a_{t_{k-1}+i}$ and $b_{t_{k-1}+i+q}$ is inserted after $a_{t_{k-1}+i}$.

We use that knowledge to proof Equation (2.10). Note that the first case requires $j > 0$ and the second case requires $j < q$ so we look at $j = 0$ and $j = q$ separately.

Using Bayes' theorem we obtain the following identities:

$$\begin{aligned} P(X_{i+q} \geq 2t_{k-1}+i \wedge \tilde{Y}_{i,q-1} = 0) &= P\left(X_{i+q} \geq 2t_{k-1}+i \mid \tilde{Y}_{i,q-1} = 0\right) \cdot P(\tilde{Y}_{i,q-1} = 0) \\ P(X_{i+q} < 2t_{k-1}+i \wedge \tilde{Y}_{i,q-1} = q-1) &= P\left(X_{i+q} < 2t_{k-1}+i \mid \tilde{Y}_{i,q-1} = q-1\right) \cdot P(\tilde{Y}_{i,q-1} = q-1) \end{aligned} \tag{2.12}$$

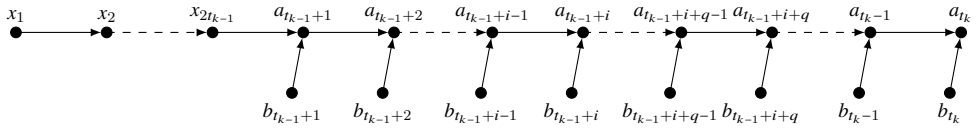


Figure 2.5: Configuration where one batch of $t_k - t_{k-1}$ elements remains to be inserted. The elements $b_{t_{k-1}+i}$ and $b_{t_{k-1}+i+d}$ are drawn.

The probability $P\left(X_{i+q} < 2t_{k-1}+i \mid Y_{i,q-1} = d\right)$ can be obtained by looking at Figure 2.5 and counting elements. When $b_{t_{k-1}+i+q}$ is inserted, the elements on the main chain which are smaller than $a_{t_{k-1}+i}$ are x_1 to $x_{2t_{k-1}}$, $a_{t_{k-1}+1}$ to $a_{t_{k-1}+i-1}$ and d elements out of

[2] The first part of Equation (2.10): $\frac{(2q-j)!}{2^{q-j}j!(q-j)!}$, when substituting $q = n$ and $j = n-k$ yields $a(n,k) = \frac{(n+k)!}{2^k(n-k)!k!}$ which is the number sequence A001498 from The On-Line Encyclopedia of Integer Sequences[OEIS].

$\{b_{t_{k-1}+i+1}, ..., b_{t_{k-1}+i+q-1}\}$ which is a total of $2t_{k-1} + 2i - 1 + d$ elements. Combined with the fact that the main chain consists of $2t_{k-1} + 2i + 2q - 2$ elements smaller than $a_{t_{k-1}+i+q}$ we obtain the following formula

$$P\left(X_{i+q} < 2t_{k-1} + i \mid Y_{i,q-1} = d\right) = \frac{2t_{k-1} + 2i + d}{2t_{k-1} + 2i + 2q - 1} \tag{2.13}$$

From that we can calculate

$$\begin{aligned}
&P(X_{i+q} \geq 2t_{k-1} + i | Y_{i,q-1} = d) \\
&= 1 - P(X_{i+q} < 2t_{k-1} + i | Y_{i,q-1} = d) \\
&= 1 - \frac{2t_{k-1} + 2i + d}{2t_{k-1} + 2i + 2q - 1} \\
&= \frac{2t_{k-1} + 2i + 2q - 1 - 2t_{k-1} - 2i - d}{2t_{k-1} + 2i + 2q - 1} \\
&= \frac{2q - d - 1}{2t_{k-1} + 2i + 2q - 1}
\end{aligned} \tag{2.14}$$

Now we have all the necessary ingredients to proof Equation (2.10) using induction.

1. Proof of Equation (2.10) where $j = 0$ using $\tilde{Y}_{i,q} = 0 \Leftrightarrow X_{i+q} \geq 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = 0$

$$\begin{aligned}
&P(X_{i+q} \geq 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = 0) \\
&= P\left(X_{i+q} \geq 2t_{k-1} + i \mid \tilde{Y}_{i,q-1} = 0\right) \cdot P(\tilde{Y}_{i,q-1} = 0) \\
&\overset{(2.8),(2.14)}{=} \frac{2q-1}{2t_{k-1} + 2i + 2q - 1} \cdot \frac{(2q-2)!}{2^{q-1}0!(q-1)!} 2^{q-1} \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 3)!} \frac{(t_{k-1} + i + q - 2)!}{(t_{k-1} + i - 1)!} \\
&= (2q-1)(2t_{k-1} + 2i + 2q - 2) \cdot \frac{(2q-2)!}{2^{q-1}0!(q-1)!} 2^{q-1} \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 1)!} \frac{(t_{k-1} + i + q - 2)!}{(t_{k-1} + i - 1)!} \\
&= (2q-1)2(t_{k-1} + i + q - 1) \cdot \frac{(2q-2)!}{2^q 0!(q-1)!} 2^q \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 1)!} \frac{(t_{k-1} + i + q - 2)!}{(t_{k-1} + i - 1)!} \\
&= (2q-1)2 \cdot \frac{(2q-2)!}{2^q 0!(q-1)!} 2^q \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 1)!} \frac{(t_{k-1} + i + q - 1)!}{(t_{k-1} + i - 1)!} \\
&= (2q-1)2 \cdot \frac{q}{(2q)(2q-1)} \cdot \frac{(2q-0)!}{2^q 0!(q-0)!} 2^q \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 1)!} \frac{(t_{k-1} + i + q - 1)!}{(t_{k-1} + i - 1)!} \\
&= \frac{(2q-0)!}{2^q 0!(q-0)!} 2^q \frac{(2t_{k-1} + 2i - 1)!}{(2t_{k-1} + 2i + 2q - 1)!} \frac{(t_{k-1} + i + q - 1)!}{(t_{k-1} + i - 1)!} \\
&= P(\tilde{Y}_{i,q} = 0)
\end{aligned} \tag{2.15}$$

2. Proof of Equation (2.10) where $j = q$ using $\tilde{Y}_{i,q} = q \Leftrightarrow X_{i+q} < 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = q-1$

$$
\begin{aligned}
&P(X_{i+q} < 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = q-1) \\
&= P\left(X_{i+q} < 2t_{k-1} + i \mid \tilde{Y}_{i,q-1} = q-1\right) \cdot P(\tilde{Y}_{i,q-1} = q-1) \\
&\overset{(2.8),(2.13)}{=} \frac{2t_{k-1}+2i+q-1}{2t_{k-1}+2i+2q-1} \cdot \frac{(q-1)!}{2^0\,(q-1)!0!} 2^{q-1} \frac{(2t_{k-1}+2i+q-2)!}{(2t_{k-1}+2i+2q-3)!} \frac{(t_{k-1}+i+q-2)!}{(t_{k-1}+i-1)!} \\
&= (2t_{k-1}+2i+q-1) \cdot \frac{(q-1)!}{2^0\,(q-1)!0!} 2^{q} \frac{(2t_{k-1}+2i+q-2)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= \frac{(q-1)!}{2^0\,(q-1)!0!} 2^{q} \frac{(2t_{k-1}+2i+q-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= \frac{(q)!}{2^0\,(q)!0!} 2^{q} \frac{(2t_{k-1}+2i+q-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= P(\tilde{Y}_{i,q} = q)
\end{aligned}
\tag{2.16}
$$

3. Proof of Equation (2.10) where $0 < j < q$ using
$\tilde{Y}_{i,q} = j \Leftrightarrow \left(X_{i+q} < 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = j-1\right) \vee \left(X_{i+q} \geq 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = j\right)$

$$
\begin{aligned}
&P(X_{i+q} < 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = j-1) \\
&\quad + P(X_{i+q} \geq 2t_{k-1} + i \wedge \tilde{Y}_{i,q-1} = j) \\
&= P\left(X_{i+q} < 2t_{k-1} + i \mid \tilde{Y}_{i,q-1} = j-1\right) \cdot P(\tilde{Y}_{i,q-1} = j-1) \\
&\quad + P\left(X_{i+q} \geq 2t_{k-1} + i \mid \tilde{Y}_{i,q-1} = j\right) \cdot P(\tilde{Y}_{i,q-1} = j) \\
&\overset{(2.8),(2.13),(2.14)}{=} \frac{2t_{k-1}+2i+j-1}{2t_{k-1}+2i+2q-1} \cdot \frac{(2q-j-1)!}{2^{q-j}\,(j-1)!\,(q-j)!} 2^{q-1} \frac{(2t_{k-1}+2i+j-2)!}{(2t_{k-1}+2i+2q-3)!} \frac{(t_{k-1}+i+q-2)!}{(t_{k-1}+i-1)!} \\
&\quad + \frac{2q-j-1}{2t_{k-1}+2i+2q-1} \cdot \frac{(2q-j-2)!}{2^{q-j}\,(j-1)!\,(q-j)!} 2^{q-1} \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-3)!} \frac{(t_{k-1}+i+q-2)!}{(t_{k-1}+i-1)!} \\
&= (2t_{k-1}+2i+j-1) \cdot \frac{(2q-j-1)!}{2^{q-j}\,(j-1)!\,(q-j)!} 2^{q} \frac{(2t_{k-1}+2i+j-2)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&\quad + (2q-j-1) \cdot \frac{(2q-j-2)!}{2^{q-j}\,(j-1)!\,(q-j)!} 2^{q} \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= \left(\frac{(2q-j-1)!}{2^{q-j}\,(j-1)!\,(q-j)!} + \frac{(2q-j-1)!}{2^{q-j}\,(j-1)!\,(q-j)!}\right) 2^{q} \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= \left(\frac{j}{2q-j} + \frac{2(q-j)}{2q-j}\right) \frac{(2q-j)!}{2^{q-j} j!\,(q-j)!} 2^{q} \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= \frac{(2q-j)!}{2^{q-j} j!\,(q-j)!} 2^{q} \frac{(2t_{k-1}+2i+j-1)!}{(2t_{k-1}+2i+2q-1)!} \frac{(t_{k-1}+i+q-1)!}{(t_{k-1}+i-1)!} \\
&= P(\tilde{Y}_{i,q} = j)
\end{aligned}
\tag{2.17}
$$

From Equation (2.10) we can derive Equation (2.8) using the relation

$$
Y_i = \tilde{Y}_{i,t_k - t_{k-1} - i} + 2t_{k-1} + i - 1 \tag{2.18}
$$

which leads to

$$
\begin{aligned}
P(Y_i = j) &= P(\tilde{Y}_{i,t_k-t_{k-1}-i} + 2t_{k-1} + i - 1 = j) \\
&= P(\tilde{Y}_{i,t_k-t_{k-1}-i} = j - 2t_{k-1} - i + 1) \\
&= \frac{(2t_k - 2t_{k-1} - 2i - j + 2t_{k-1} + i - 1)!}{2^{t_k-t_{k-1}-i-j+2t_{k-1}+i-1}\,(j - 2t_{k-1} - i + 1)!\,(t_k - t_{k-1} - i - j + 2t_{k-1} + 1 - 1)!} \\
&\quad \cdot 2^{t_k-t_{k-1}-i} \frac{(2t_{k-1} + 2i + j - 2t_{k-1} - i + 1 - 1)!}{(2t_{k-1} + 2i + 2t_k - 2t_{k-1} - 2i - 1)!} \frac{(t_{k-1} + i + t_k - t_{k-1} - i - 1)!}{(t_{k-1} + i - 1)!} \\
&= \frac{(2t_k - i - j - 1)!}{2^{2^k-j-1}\,(-2t_{k-1} - 1 + j + 1)!\,(2^k - j - 1)!} \cdot 2^{t_k-t_{k-1}-i} \frac{(i+j)!}{(2t_k - 1)!} \frac{(t_k - 1)!}{(t_{k-1} + i - 1)!}
\end{aligned}
\tag{2.19}
$$

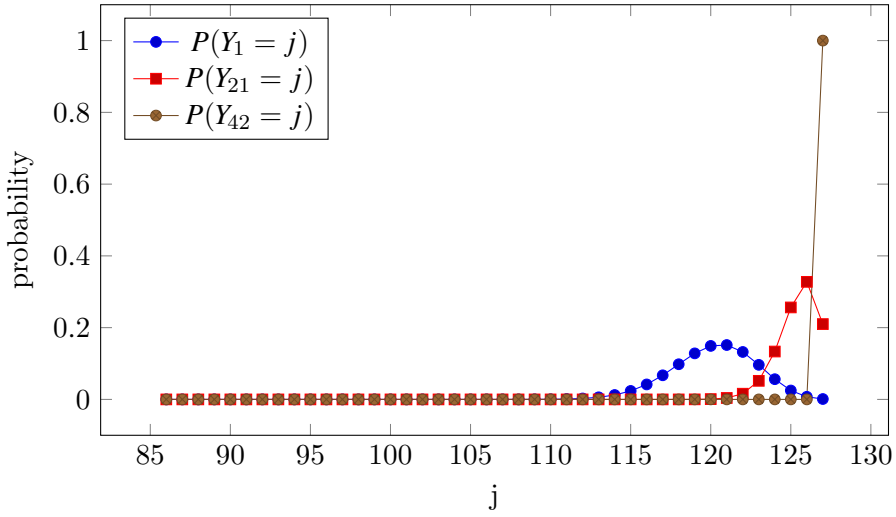


Figure 2.6: Probability distribution of Y_i where $k = 7$.

Figure 2.6 shows the probability distribution for Y_1, Y_{21} and Y_{42} where $k = 7$. As Y_{42} corresponds to the insertion of the first element of the batch (i.e. b_{t_k}) it is always inserted into $2^k - 1$ elements, which in our case is 127. Thus the probability $P(Y_{42} = j)$ is 1 at $j = 127$ and 0 for all other values of j. On the other hand there is Y_1 which corresponds to the insertion of $b_{t_{k-1}+1}$, i.e. the last element of the batch. Its probability distribution reaches its maximum at $j = 121$, which is 6 elements less than the worst case. Then there is the probability distribution of Y_{21} which was chosen simply because it is between the two border cases. There is nothing spectacular to it, though its maximum is a bit further to the right than one might expect.

In addition to those three probability distributions Figure 2.7 shows the mean of all Y_i for $k = 7$. Looking at the values from above, for $i = 42$ the mean (non-surprisingly) is 127, for $i = 1$ it is close to 120 and in the case of $i = 21$ it is between 125 and 126. In between we have what appears to be a roughly exponential curve.

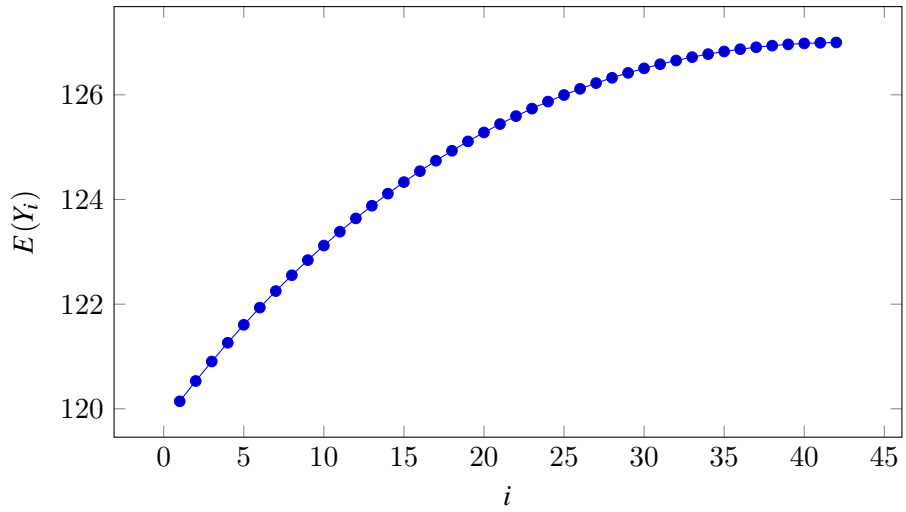


Figure 2.7: Mean of Y_i for different i where $k = 7$.

If you are wondering why $k = 7$ was chosen and if the graphs would look any different for other values of k then here is the answer: In the case of $k = 7$ there are 42 elements to be inserted. That is enough data points for the curves to look smooth, yet sufficiently few so one can still see the individual points. Using larger (or smaller) values of k results in similar curves. The difference comes down to scaling.

2.2 Binary Insertion

The Binary Insertion step is an important part of MergeInsertion. In the worst case each element is inserted into $2^k - 1$ elements, thus always requiring k comparisons. In contrast to that, in the average case many elements are inserted into slightly less than $2^k - 1$ elements. As a result inserting an element requires either k or $k - 1$ comparisons, i.e. potentially less than the worst case. However inserting into less than $2^k - 1$ elements leads to ambiguous decision trees for the binary insertion.

Figure 2.8 shows different strategies for inserting an element into 5 elements. First there are the `center-left` and `center-right` strategies. `center-left` compares the element to be inserted with the middle element, rounding down in case of odd number. `center-right` is similar as it also compares the element to be inserted with the middle element. The difference is that in case of an odd number we are rounding up, i.e. choosing the element to the right of the middle. Now those two options are what one would naturally use when implementing binary insertion. They have the property that the corresponding decision trees have their leaves distributed across at most two layers. That makes them optimal in case of a uniform distribution.

The `left` and `right` strategies also fulfill this property. The `left` strategy compares the element to be inserted with the left-most element such that the corresponding decision tree has its leaves distributed across at most two layers. As a result inserting an element at the

(a) `center-left` **(b)** `center-right`

(c) `left` **(d)** `right`

Figure 2.8: Different strategies for binary insertion.

Algorithm 2.1 Binary Insertion

```
1:  procedure INSERT(a,x_1, ..., x_n)
2:      if n = 0 then
3:          return a
4:      end if
5:      k ← ⌊log n⌋
6:      c ← { ⌊(n+1)/2⌋                  strategy center-left
            { ⌈(n+1)/2⌉                  strategy center-right
            { max{n − 2^k + 1, 2^(k−1)}  strategy left
            { min{2^k, n − 2^(k−1) + 1}  strategy right
7:      if a < x_c then
8:          y_1, ..., y_c ←INSERT(a,x_1, ..., x_(c−1))
9:          return y_1, ..., y_c, x_c, ..., x_n
10:     else
11:         y_c, ..., y_n ←INSERT(a,x_(c+1), ..., x_n)
12:         return x_1, ..., x_c, y_c, ..., y_n
13:     end if
14: end procedure
```

left using this strategy can be cheaper than inserting it further to the right. The `right` strategy is similar to `left`, the difference is that here the right-most element is chosen such that the corresponding decision tree has its leaves distributed across at most two layers. Algorithm 2.1 shows a pseudo code implementation of binary insertion including all four strategies presented here.

Since all of the presented strategies for binary insertion cause the leaves of the decision tree to be spread across at most two layers they are all optimal if each position is equally likely. Unfortunately that is not the case with MergeInsertion. The positions on the left have a slightly higher probability. See Figure 2.9 for an example. Therefore the way we do the binary insertion affects the number of comparisons the algorithm requires.

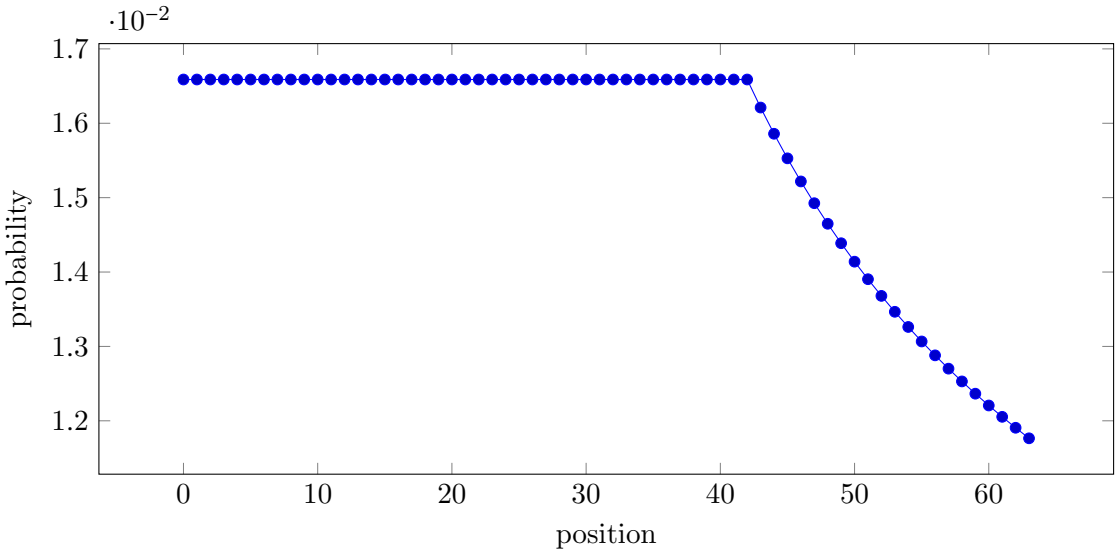


Figure 2.9: Probabilities of different positions when inserting b_{t_k} where $k = 6$.

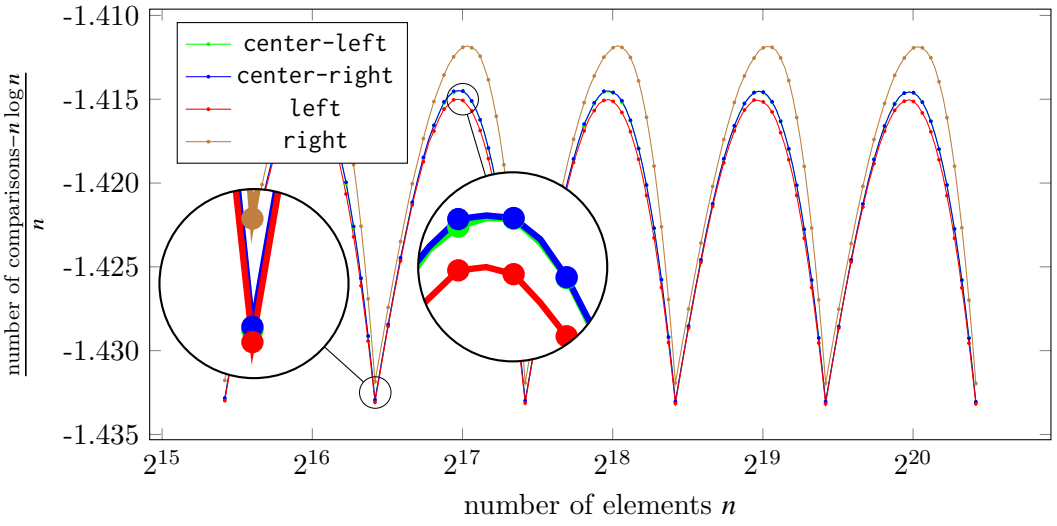


Figure 2.10: Experimental results on the effect of different strategies for binary insertion on the number of comparisons.

Figure 2.10 shows experimental results comparing the different strategies for binary insertion presented in Figure 2.8 regarding their effect on the average-case of MergeInsertion. As we can see the `left` strategy performs the best, closely followed by `center-left` and `center-right`.

`right` performs the worst. The `left` strategy performing best is no surprise if we look at Figure 2.9: The probability that an element is inserted into one of the left positions is higher that it being inserted to the right. Thus insertions to the left being cheaper increases the performance of the algorithm.

2.3 A First Approximation

The goal of this section is to combine the probability given by Equation (2.8) that an element $b_{t_{k-1}+i}$ is inserted into j elements with an approximation for the number of comparisons required for binary insertion.

The number of comparisons required for binary insertion when inserting into $m-1$ elements was determined to be $T_{\text{InsAvg}}(m) = \lceil \log m \rceil + 1 - \frac{2^{\lceil \log m \rceil}}{m}$ by [EW13]. While only being correct in case of a uniform distribution, this formula acts as an upper bound in our case, where the probability is monotonically decreasing with the index.

Combining that with Equation (2.8) we obtain the following upper bound for the cost of inserting $b_{t_{k-1}+i}$:

$$T_{\text{Ins}}(i,k) = \sum_j P(Y_i = j) \cdot T_{\text{InsAvg}}(j+1) \tag{2.20}$$

Unfortunately we have not been able to simplify this equation or use it for any further analysis, mostly because Equation (2.8) is a very complicated formula. However a computer can evaluate that equation just fine, so that is what we are going to do.

According to [Knu98] the number of comparisons required by MergeInsertion is

$$F(n) = \left\lfloor \frac{n}{2} \right\rfloor + F\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + G\left(\left\lceil \frac{n}{2} \right\rceil\right) \tag{2.21}$$

where $\left\lfloor \frac{n}{2} \right\rfloor$ is the number of pairwise comparisons, $F\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ is the number of comparisons required in the recursion step and $G\left(\left\lceil \frac{n}{2} \right\rceil\right)$ corresponds to the work done in the insertion step.

We can express $G(m)$ in terms of $T_{\text{Ins}}(i,k)$ as follows:

$$G(m) = \sum_{i=1}^{m-t_{k_m-1}} T_{\text{Ins}}(i,k_m) + \sum_{1 \le k < k_m} \left(\sum_{i=1}^{t_k - t_{k-1}} T_{\text{Ins}}(i,k) \right) \tag{2.22}$$

Where $t_{k-1} \le m < t_k$.

In Figure 2.11 we can see the resulting approximation in comparison with experimental data on the number of comparisons required by MergeInsertion. The difference is rather small.

However the computational effort required to calculate the approximation grows quickly. The results presented here correspond to what can be calculated on a home computer within reasonable amount of time.

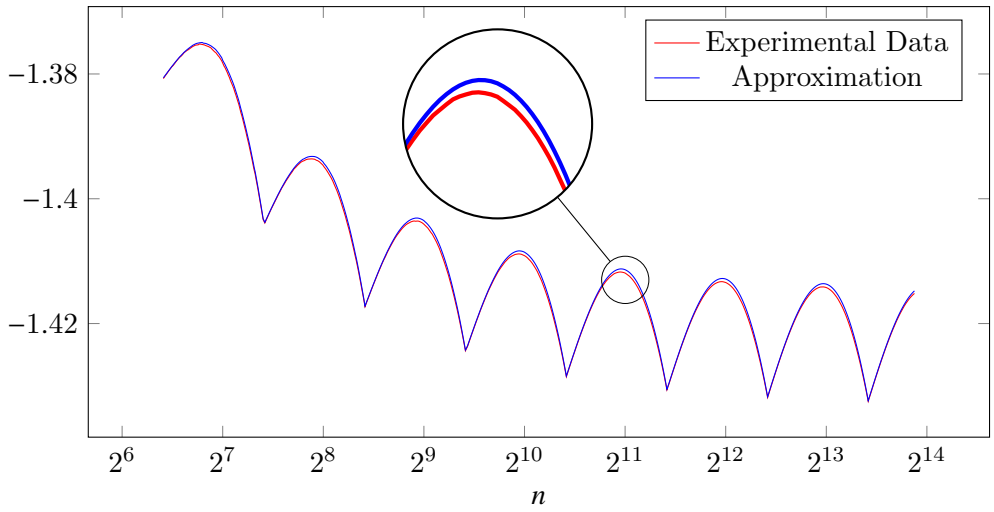


Figure 2.11: Comparing our approximation with experimental data on the number of comparisons required by MergeInsertion.

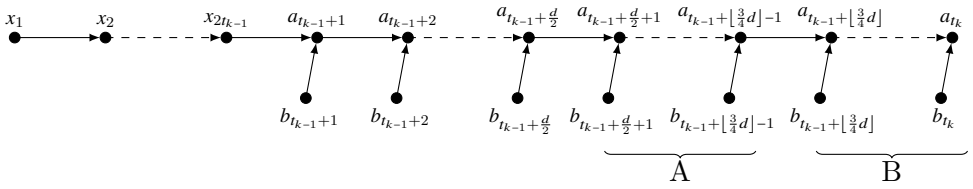


Figure 2.12: Configuration where one batch is to be inserted.

2.4 Improving Upon the Upper Bound

In this section we improve upon the upper bound by analyzing the insertion of one batch of elements. The exact probability that $b_{t_{k-1}+i}$ is inserted into j elements is given by Equation (2.8). We are especially interested in the case of $b_{t_{k-1}+u}$ where $u = \lfloor\frac{t_k - t_{k-1}}{2}\rfloor$, because if we know $P(Y_u < m)$ then we can use that for all $q < u$ the probability of $b_{t_{k-1}+q}$ being inserted into less than m elements is at least $P(Y_u < m)$, i.e. $P(Y_q < m) \geq P(Y_u < m)$. This is because when $b_{t_{k-1}+i}$ is inserted into m elements, then no matter which position it is inserted into, the next element, $b_{t_{k-1}+i-1}$, is inserted into at most m elements.

However Equation (2.8) is hard to work with, so we approximate it with a binomial distribution. For a given k let $d = t_k - t_{k-1}$ be the number of elements that are inserted as part of the batch. This configuration is illustrated in Figure 2.12. Remember $u = \frac{t_k - t_{k-1}}{2} = \frac{d}{2}$. To calculate into how many elements $b_{t_{k-1}+u} = b_{t_{k-1}+\frac{d}{2}}$ is inserted, we ask how many elements out of $b_{t_{k-1}+\lfloor\frac{3}{4}d\rfloor}$ to b_{t_k} (marked as section B in Figure 2.12) are inserted between $a_{t_{k-1}+\frac{d}{2}+1}$ and $a_{t_{k-1}+\lfloor\frac{3}{4}d\rfloor-1}$ (marked as section A).

The rationale is that for each element from section B that is inserted into section A, $b_{t_{k-1}+u}$ is inserted into one less element. As a lower bound for the probability that an element from section B is inserted into one of the positions in section A we use the probability that b_{t_k} is inserted between a_{t_k-1} and a_{t_k} which is $\frac{1}{2t_k-1}$.

That is because if we assume that all b_i with $i < t_k$ are inserted before inserting b_{t_k}, then b_{t_k} is inserted into $2t_k - 2$ elements, so the probability for each position is $\frac{1}{2t_k-1}$. Since none of the b_i with $i < t_k$ can be inserted between a_{t_k-1} and a_{t_k} because they are all smaller than a_{t_k-1}, the probability that b_{t_k} is inserted between a_{t_k-1} and a_{t_k} does not change when we insert it first as the algorithm demands.

To calculate the probability that an element b_{t_k-q} with $q > 0$ is inserted into the rightmost position we assume that all b_i with $i < t_k - q$ are inserted before inserting b_{t_k-q}. Then b_{t_k-q} is inserted into at most $2t_k - q - 2$ elements, i.e. the elements x_1 to $x_{2t_{k-1}}$, $a_{t_{k-1}+1}$ to a_{t_k-q-1}, $b_{t_{k-1}+1}$ to b_{t_k-q-1} and at most q elements out of b_{t_k-q+1} to b_{t_k}.

Hence the probability for each position is greater than $\frac{1}{2t_k-q-1}$ which is greater than $\frac{1}{2t_k-1}$. Since none of the b_i with $i < t_k - q$ can be inserted to the right of a_{t_k-q-1}, the probability that $b_{t_k} - q$ is inserted into any of the positions between a_{t_k-q-1} and a_{t_k-q} remains unchanged when inserting the elements in the correct order.

The probability that an element is inserted at a specific position is monotonically decreasing with the index. This is because if an element b_i is inserted to the left of an element a_{i-h} then b_{i-h} is inserted into one more element than it would be if b_i had been inserted to the right of a_{i-h}. As a result any position further to the left is more likely than the right-most position, so we can use that as a lower bound.

There are $\lfloor \frac{d}{4} \rfloor - 1$ elements in section A, i.e. there are at least $\lfloor \frac{d}{4} \rfloor$ positions where an element can be inserted. Hence the probability that an element from section B is inserted into section A is at least $\frac{\lfloor \frac{d}{4} \rfloor}{2t_k-1}$ and consequently the probability that it is not inserted before $b_{t_{k-1}+u}$ is at least $\frac{\lfloor \frac{d}{4} \rfloor}{2t_k-1}$. That is because all positions part of section A are after $a_{t_{k-1}+u}$.

Section B contains $\lceil \frac{d}{2} \rceil$ elements. Using that and substituting $u = \frac{d}{2}$ we obtain the binomial distribution with the parameters $n_B = \lceil \frac{1}{2} \rceil$ and $p_B = \frac{\lfloor \frac{d}{4} \rfloor}{2t_k-1}$. As a result we have

$$p(j) = \binom{\lceil \frac{1}{2} \rceil}{q} (\frac{\lfloor \frac{u}{2} \rfloor}{2t_k - 1})^q (\frac{2t_k - 1 - \lfloor \frac{u}{2} \rfloor}{2t_k - 1})^{\lceil \frac{1}{2} \rceil - q} \tag{2.23}$$

with $q = 2^k - 1 - j$, that by construction fulfills the property given in Equation (2.24) for all j_0.

$$\sum_{j=0}^{j_0} p(j) \leq \sum_{j=0}^{j_0} P(Y_u = j) = P(Y_u \leq j_0) \tag{2.24}$$

Figure 2.13 compares our approximation $p(j)$ with real distribution $P(Y_u = j)$. We observe that the maximum of our approximation is further to the right than the one of the real distribution.

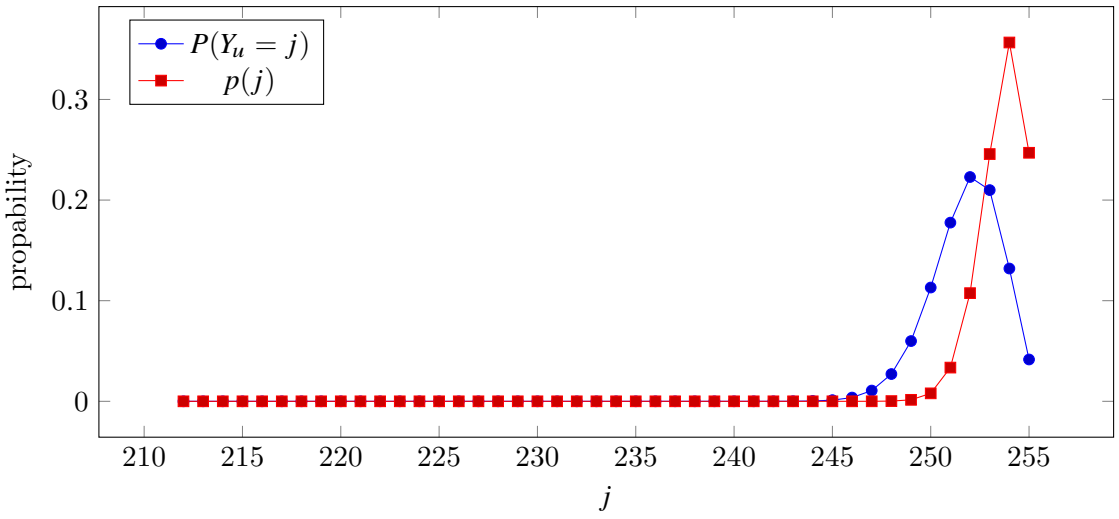


Figure 2.13: Difference between the real distribution and our approximation for $k = 8$ and $u = 43$.

By using the approximation $P(Y_u = j) \approx p(j)$ we can calculate a lower bound for the median of $Y_{\frac{t_k - t_{k-1}}{2}}$

$$
\begin{aligned}
&2^k - 1 - \lfloor n_B \cdot p_B \rfloor \\
&= 2^k - 1 - \left\lfloor \left\lceil \frac{t_k - t_{k-1}}{4} \right\rceil \frac{\left\lfloor \frac{t_k - t_{k-1}}{4} \right\rfloor}{2t_k - 1} \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{k-2} + (-1)^k - 3}{3} + \frac{1}{2}(-1)^k + \frac{1}{2} \right) \left(\frac{2^{k-2} + (-1)^k - 3}{3} + \frac{1}{2}(-1)^k - \frac{1}{2} \right) \frac{1}{2t_k - 1} \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{k-2}}{3} + \frac{1}{6}(-1)^k + \frac{1}{2} \right) \left(\frac{2^{k-2}}{3} + \frac{1}{6}(-1)^k - \frac{1}{2} \right) \frac{1}{2t_k - 1} \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{2k-4}}{9} + \frac{2^{k-2}}{9}(-1)^k + \frac{1}{36} - \frac{1}{4} \right) \frac{1}{2t_k - 1} \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{2k-4}}{9} + \frac{2^{k-2}}{9}(-1)^k + \frac{1}{36} - \frac{1}{4} \right) \left(\frac{1}{2\frac{2^{k+1} + (-1)^k}{3} - 1} \right) \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{2k-4}}{9} + \frac{2^{k-2}}{9}(-1)^k + \frac{1}{36} - \frac{1}{4} \right) \left(\frac{1}{2\frac{2^{k+1}}{3}} \pm O(2^{-k}) \right) \right\rfloor \\
&= 2^k - 1 - \left\lfloor \left(\frac{2^{2k-4}}{9} + \frac{2^{k-2}}{9}(-1)^k + \frac{1}{36} - \frac{1}{4} \right) \frac{1}{2\frac{2^{k+1}}{3}} \pm O(1) \right\rfloor \\
&= 2^k - 1 - \left\lfloor \frac{2^{k-6}}{3} + \frac{1}{3}(-1)^k] \pm O(1) \right\rfloor \\
&\in 2^k - 1 - \frac{2^{k-6}}{3} + O(1)
\end{aligned}
\tag{2.25}
$$

This tells us that with a probability $\geq 50\%$, $b_{t_{k-1}+u}$ is inserted into $2^k - 1 - \frac{2^{k-6}}{3} \pm O(1)$ or less elements. In conclusion all b_i with $i \leq u = \frac{t_k - t_{k-1}}{2}$ are inserted into less than $2^k - 1 - \frac{2^{k-6}}{3} \pm O(1)$ elements with a probability $\geq 50\%$.

Using that result we can calculate a better upper bound for the average case performance of the entire algorithm.

According to Knuth [Knu98] in its worst case MergeInsertion requires $W(n) = n \log n - (3 - \log 3)n + n(y + 1 - 2^y) + O(\log n)$ comparisons.

We calculate the number of comparisons required in the average case in a similar fashion to [EW13]. $F(n)$ shall be the number of comparisons required by the algorithm.

$$F(n) = \left\lfloor \frac{n}{2} \right\rfloor + F\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + G\left(\left\lceil \frac{n}{2} \right\rceil\right) \tag{2.26}$$

$G(m)$ corresponds to the work done in the third step of the algorithm and is given by

$$G(m) = (k_m - \alpha_m)(m - t_{k_m - 1}) + \sum_{1 \leq k < k_m} (k - \beta_k)\,(t_k - t_{k-1}) \tag{2.27}$$

where $t_{k_m-1} \leq m < t_{k_m}$ and $\alpha_m, \beta \in [0, 1]$. Inserting an element b_i with $t_{k_{i-1}} < i \leq t_{k_i}$ requires at most k_i comparisons. However since we are looking at the average case we need to consider that in some cases b_i can be inserted using just $k_i - 1$ comparisons. This is reflected by α_m and β_k, the first of which has already been studied by [EW13].

To estimate the cost of an insertion we use the formula $T_{\text{InsAvg}}(m) = \lceil \log m \rceil + 1 - \frac{2^{\lceil \log m \rceil}}{m}$ by [EW13]. Technically this formula is only correct if the probability of an element being inserted is the same for each position. This is not the case with MergeInsertion. Instead the probability is monotonically decreasing with the index. Binary insertion can be implemented to take advantage of this property, as explained in Section 2.2, in which case $T_{\text{InsAvg}}(m)$ acts as an upper bound on the cost of an insertion.

Using our result from above that on average $\frac{1}{4}$ of the elements are inserted in less than $2^k - 1 - \frac{2^{k-4}}{9} \pm O(1)$ elements we can calculate β_k as the difference of the cost of an insertion in the worst-case (k) and in the average case.

$$
\begin{aligned}
\beta_k &\geq k - \left(\frac{3}{4}T_{\text{InsAvg}}\left(2^k\right) + \frac{1}{4}T_{\text{InsAvg}}\left(2^k - \frac{2^{k-6}}{3} \pm O(1)\right)\right) \\
&= k - \left(\frac{3}{4}\left(k + 1 - \frac{2^k}{2^k}\right) + \frac{1}{4}\left(k + 1 - \frac{2^k}{2^k - \frac{2^{k-6}}{3} \pm O(1)}\right)\right) \\
&= -1 + \frac{3}{4} + \frac{1}{4} \cdot \frac{1}{1 - \frac{1}{1 - \frac{2^{-6}}{3}} \pm O(2^{-k})} \\
&= -\frac{1}{4} + \frac{1}{4} \cdot \frac{1}{1 - \frac{1}{192}} \pm O(2^{-k}) \\
&= -\frac{1}{4} + \frac{1}{4} \cdot \frac{1}{\frac{191}{192}} \pm O(2^{-k}) \\
&= -\frac{1}{4} + \frac{1}{4} \cdot \frac{192}{191} \pm O(2^{-k}) \\
&= \frac{1}{764} \pm O(2^{-k})
\end{aligned}
\tag{2.28}
$$

Combining this with Equation (2.27) we can calculate the difference between the worst-case and the average-case as

$$
\begin{aligned}
&G_{\text{worst-case}}(m) - G_{\text{average-case}}(m) \\
&= k_m(m - t_{k_m-1}) + \sum_{1 \leq k < k_m} k\,(t_k - t_{k-1}) - (k_m - \alpha_m)(m - t_{k_m-1}) - \sum_{1 \leq k < k_m} (k - \beta_k)\,(t_k - t_{k-1}) \\
&= \alpha_m(m - t_{k_m-1}) + \sum_{1 \leq k < k_m} \beta_k\,(t_k - t_{k-1}) \\
&\geq \alpha_m(m - t_{k_{m-1}}) + \sum_{1 \leq k < k_m} \left(\frac{1}{764} \pm O(2^{-k})\right)(t_k - t_{k-1}) \\
&= \alpha_m(m - t_{k_m-1}) + \frac{1}{764}(t_{k_{m-1}} - t_1) \pm O(\log m) \\
&= \alpha_m(m - t_{k_m-1}) + \frac{1}{764}t_{k_{m-1}} \pm O(\log m) \\
&= \alpha_m(m - t_{k_m-1}) + \frac{1}{764}\frac{2^k_m + (-1)^{k_m-1}}{3} \pm O(\log m) \\
&= \alpha_m(m - t_{k_m-1}) + \frac{1}{764}\frac{2^{k_m}}{3} \pm O(\log m)
\end{aligned}
\tag{2.29}
$$

By writing m as $m = 2^{l_m - \log 3 + x}$ with $x \in [0, 1)$ we get $l_m = \lfloor \log 3m \rfloor$. To approximate k_m with l_m we need to show that $k_m \geq l_m$. Recall that $t_{k_m-1} \leq m < t_{k_m}$. For all $t_{k_m-1} < m < t_{k_m}$ we have

$$
\frac{2^{k_m} + (-1)^{k_m-1}}{3} < m < \frac{2^{k_m+1} + (-1)^{k_m}}{3} \tag{2.30}
$$

Since $m \in \mathbb{N}$ and $t_k \in \mathbb{N}$ adding/subtracting $\frac{1}{3}$ does not alter the relation, so we obtain

$$
\frac{2^{k_m}}{3} < m < \frac{2^{k_m+1}}{3} \tag{2.31}
$$

which resolves to

$$k_m < \log 3n < k_m + 1 \tag{2.32}$$

Thus $k_m = \lfloor \log 3m \rfloor = l_m$.

For $m = t_{k_m-1}$ we get

$$\begin{aligned} & \frac{2^{k_m} + (-1)^{k_m-1}}{3} = m \\ \iff & 2^{k_m} = 3m + (-1)^{k_m} \\ \iff & k_m = \log\left(3m + (-1)^{k_m}\right) \end{aligned} \tag{2.33}$$

If $k_m = \log(3m+1)$ that resolves to $k_m = \log(3m+1) > \log(3m) > \lfloor \log 3m \rfloor = l_m$.

If instead $k_m = \log(3m-1)$ using $k_m \in \mathbb{N}$ we have $k_m = \lfloor \log(3m-1) \rfloor$ and for all $m \geq 1$ this is equal to $\lfloor \log 3m \rfloor = l_m$.

Hence in all cases $l_m \leq k_m$ holds. Therefore we can replace k_m with l_m in Equation (2.29):

$$G_{\text{worst-case}}(m) - G_{\text{average-case}}(m) \geq \alpha_m(m - t_{k_m-1}) + \frac{1}{764}\frac{2^{l_m}}{3} \pm O(\log m) \tag{2.34}$$

From [EW13] we know that the $\alpha_m(m - t_{k_m-1})$ term can be approximated with $\left(m - 2^{l_m - \log 3}\right)\left(\frac{2^{l_m}}{m + 2^{l_m - \log 3}} - 1\right)$.

$$G_{\text{worst-case}}(m) - G_{\text{average-case}}(m) \geq \left(m - 2^{l_m - \log 3}\right)\left(\frac{2^{l_m}}{m + 2^{l_m - \log 3}} - 1\right) + \frac{1}{764}\frac{2^{l_m}}{3} \pm O(\log m) \tag{2.35}$$

Now we calculate

$$\begin{aligned} S(n) &= F_{\text{worst-case}}(m) - F_{\text{average-case}}(m) \\ &= \left\lfloor \frac{n}{2} \right\rfloor + F_{\text{worst-case}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + G_{\text{worst-case}}\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &\quad - \left\lfloor \frac{n}{2} \right\rfloor - F_{\text{average-case}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) - G_{\text{average-case}}\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &= S(\left\lfloor \frac{n}{2} \right\rfloor) + G_{\text{worst-case}}\left(\left\lceil \frac{n}{2} \right\rceil\right) - G_{\text{average-case}}\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &\geq S(\left\lfloor \frac{n}{2} \right\rfloor) + \left(m - 2^{l_m - \log 3}\right)\left(\frac{2^{l_m}}{m + 2^{l_m - \log 3}} - 1\right) + \frac{1}{764}\frac{2^{l_m}}{3} \pm O(\log m) \end{aligned} \tag{2.36}$$

We split $S(n)$ into $S_\alpha(n) + S_\beta(n)$ with

$$\begin{aligned} S_\alpha(n) &\geq S_\alpha(\left\lfloor \tfrac{n}{2} \right\rfloor) + \left(m - 2^{l_m - \log 3}\right)\left(\tfrac{2^{l_m}}{m + 2^{l_m - \log 3}} - 1\right) \\ S_\beta(n) &\geq S_\beta(\left\lfloor \tfrac{n}{2} \right\rfloor) + \tfrac{1}{764}\tfrac{2^{l_m}}{3} \pm O(\log m) \end{aligned} \tag{2.37}$$

From [EW13] we know $S_\alpha(n) \geq \left(n - 2^{l_n - \log 3}\right)\left(\frac{2^{l_n}}{n + 2^{l_n - \log 3}} - 1\right) + O(1)$.

For $S_\beta(n)$ we obtain

$$\begin{aligned} S_\beta(n) &\geq \sum_{i=1}^{l_n-1} \frac{2^i}{764 \cdot 3} \pm O(\log 2^i) \\ &= \frac{2^{l_n}}{2292} \pm O(\log^2 n) \end{aligned} \tag{2.38}$$

With $n = 2^{k-\log 3+x}$

$$\begin{aligned} \frac{S(n)}{n} &= \frac{S_\alpha(n) + S_\beta(n)}{n} \\ &= \frac{2^{k-\log 3+x} - 2^{k-\log 3}}{2^{k-\log 3+x}} \left(\frac{2^k}{2^{k-\log 3+x} + 2^{k-\log 3}} - 1 \right) + \frac{2^k}{2292 \cdot 2^{k-\log 3+x}} \pm O(\frac{\log^2 n}{n}) \\ &= (1 - 2^{-x}) \left(\frac{3}{2^x + 1} - 1 \right) + \frac{2^{\log 3-x}}{2292} \pm O(\frac{\log^2 n}{n}) \end{aligned} \tag{2.39}$$

By writing $F(n)$ as $F(n) = n \log n - c(n) \cdot n \pm O(\log^2 n)$ we get

$$\begin{aligned} c(n) &\geq -\frac{(F(n) - n \log n)}{n} \\ &= -\frac{(W(n) - S(n) - n \log n)}{n} \\ &= (3 - \log 3) - (y + 1 - 2^y) + (1 - 2^{-x}) \left(\frac{3}{2^x + 1} - 1 \right) + \frac{2^{\log 3-x}}{2292} \end{aligned} \tag{2.40}$$

With $y = 1 - x$ the result is

$$c(n) \geq (3 - \log 3) - (2 - x - 2^{1-x}) + (1 - 2^{-x}) \left(\frac{3}{2^x + 1} - 1 \right) + \frac{2^{\log 3-x}}{2292} \geq 1.4005 \tag{2.41}$$

A visual representation of $c(n)$ is provided in Figure 2.14. The worst case is near $x = 0.6$ where $c(n)$ is just slightly larger than 1.4005.

Hence we have obtained a new upper bound for the average case of MergeInsertion which is $n \log n - 1.4005n + O(\log^2 n)$.

2.5 Computing the Number of Comparisons

In this section we explore how we can calculate the number of comparisons the algorithm requires in the average case using a computer. The most straight forward way of doing this is to compute the external path length of the decision tree and dividing by the number of leaves($n!$ when sorting n elements). Unfortunately doing so is very expensive (in terms of computation time), so we would only be able to do it for a few small n.

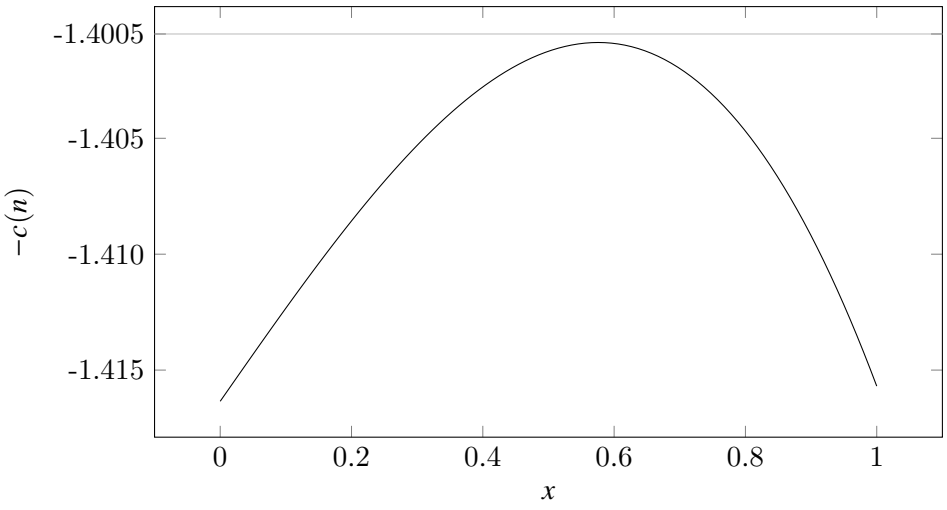


Figure 2.14: Plot of $c(n)$.

Algorithm 2.2 Computation of $F(n)$

```
1: procedure ComputeF(n)
2:     if n = 1 then
3:         return 0
4:     else
5:         return ⌊n/2⌋ +ComputeF(⌊n/2⌋)+ComputeG(⌈n/2⌉)
6:     end if
7: end procedure
```

Remember this formula for the number of comparisons:

$$F(n) = \left\lfloor \frac{n}{2} \right\rfloor + F\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + G\left(\left\lceil \frac{n}{2} \right\rceil\right) \tag{2.42}$$

The three terms of the sum represent the three steps of the algorithm. In the first step $\left\lfloor \frac{n}{2} \right\rfloor$ pairs of elements are compared. Next the $\left\lfloor \frac{n}{2} \right\rfloor$ larger elements are sorted recursively, requiring $F\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ comparisons. In the last step the $\left\lceil \frac{n}{2} \right\rceil$ smaller elements are inserted using $G\left(\left\lceil \frac{n}{2} \right\rceil\right)$ comparisons.

We can easily implement the calculation of $F(n)$ as shown in Algorithm 2.2. Calculating $G(n)$ is not that straight forward. Since the insertion step of MergeInsertion works by inserting elements in batches, we write $G(n)$ as the sum of the cost of those batches:

$$G(n) = \left(\sum_{1<k\leq k_n} \texttt{Cost}(t_{k-1}, t_k) \right) + \texttt{Cost}(t_{k_n}, n) \text{ where } t_{k_n} \leq n < t_{k_n+1} \tag{2.43}$$

Here $\texttt{Cost}(s, e)$ is the cost of inserting one batch of elements starting from b_{s+1} up to b_e. Such a configuration is shown in Figure 2.15.

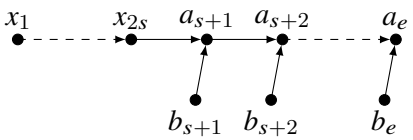


Figure 2.15: Configuration where one batch of $e - s$ elements, b_{s+1} to b_e, remains to be inserted.

Algorithm 2.3 Computation of $G(n)$

```
1: procedure ComputeG(n)
2:     k ← 2
3:     c ← 0
4:     while t_k < n do
5:         c ← c+Cost(t_{k−1}, t_k)
6:         k ← k + 1
7:     end while
8:     c ← c+Cost(t_{k−1}, n)
9:     return c
10: end procedure
```

As we can see from Equation (2.43) most often s is t_{k-1} and e is t_k. The only exception is the last batch. It also starts at $s = t_{k-1}$ but it has less elements in total, thus e is set to whatever number of elements we have. An implementation of Equation (2.43) can be seen in Algorithm 2.3.

Now the idea for computing $\texttt{Cost}(s, e)$ is to calculate the external path length of the decision tree corresponding to the insertion of that batch of elements and then dividing by the number of leaves. We already talked about computing the external path length of the decision tree being very expensive. Doing it only for a part of the problem instead of the entire algorithm makes it a little cheaper, but it is still very expensive.

To save some computation time we apply the following optimization: We collapse ''identical'' branches of the decision tree. E.g. whether b_e is inserted between x_1 and x_2 or between x_2 and x_3 does not influence the number of comparisons required to insert the subsequent elements. So we can neglect that difference. However if b_e is inserted between a_{e-1} and a_e then the next element (and all thereafter) is inserted into one less element. So this is a difference we need to acknowledge. Same if an element is inserted between any a_i and a_{i+1}. By the time we insert b_i the element inserted between a_i and a_{i+1} is known to be larger than b_i and thus is no longer part of the main chain, resulting in b_i being inserted into one element less. In conclusion that means that our algorithm needs to keep track of the elements inserted between any a_i and a_{i+1} as well as those inserted at any position before a_{s+1} as two branches of the decision tree that differ in any of these cannot be collapsed. Algorithm 2.4 shows how this is implemented.

For $n \in \{1, ..., 14\}$ the computed values are shown in Table 2.2, for larger n Figure 2.16 shows the values we computed.

Algorithm 2.4 Computation of $\texttt{Cost}(s, e)$

```
 1: procedure Cost(s, e)
 2:     r ← e − s                                   // next element to be inserted is b_r
 3:     q_1 ← 2s                  // number of elements on the main chain that are < a_{s+1}
 4:     q_2, ..., q_r ← 0         // q_i is the number of elements between as + i − 1 and a_{s+i}
 5:     (p, l) ←CostInsert(r, q_1, ..., q_r)
 6:     return p/l
 7: end procedure
 8:
 9: procedure CostInsert(r, q_1, ..., q_r)
10:     if r = 0 then
11:         return (0, 1)                           // We reached a leave
12:     end if
13:     elements ← r − 1 + Σ q_i                    // number of elements b_r is inserted into
14:     k ← ⌈log(elements + 1)⌉
15:     cheap_insertions ← 2^k − elements − 1
16:     p ← 0                                       // external path length
17:     l ← 0                                       // number of leaves
18:     index ← 0                 // We iterate over all indices where b_r can be inserted
19:     for all 0 < i ≤ r do
20:         (p_c, l_c) ←CostInsert(r − 1, q_1, ..., q_{i−1}, q_i + 1, q_{i+1}, ..., q_{r−1})
21:         repeat q_i + 1 times                    // q_i + 1 positions between a_{s+i−1} and a_{s+i}
22:             if index < cheap_insertions then
23:                 p ← p + p_c + (k − 1) · l_c
24:             else
25:                 p ← p + p_c + k · l_c
26:             end if
27:             l ← l + l_c
28:             index ← index + 1
29:         end
30:     end for
31:     return (p, l)
32: end procedure
```

$n =$	1	2	3	4	5	6	7	8	9	10
$F(n) \cdot n! =$	0	2	16	112	832	6912	62784	623232	6743808	79292160
$n =$	11			12			13		14	
$F(n) \cdot n! =$	1013736960			13921182720			204489999360		3199119114240	

Table 2.2: Computed values of $F(n) \cdot n!$

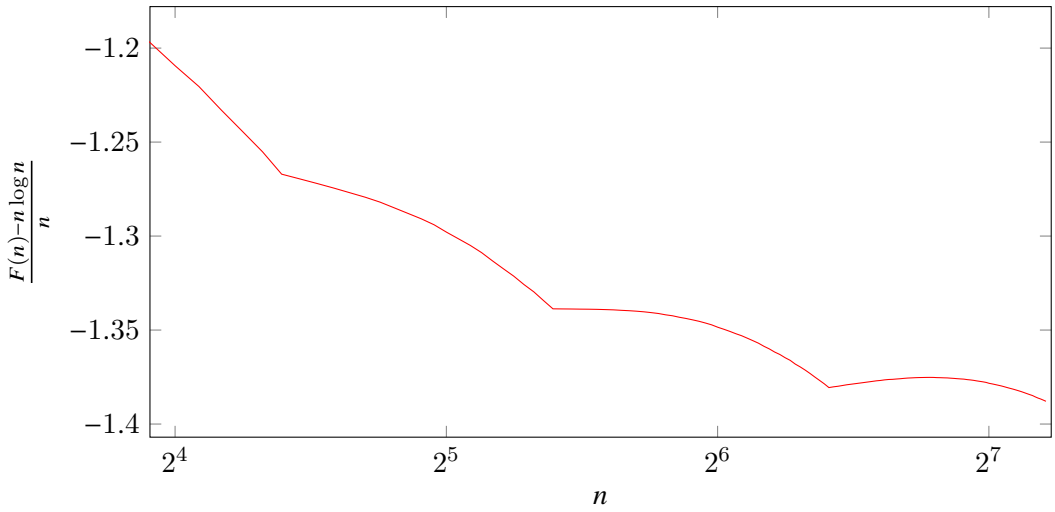


Figure 2.16: Computed values of $F(n)$.

3 Experiments

In this chapter the experiments performed are discussed. We begin by presenting our implementation of MergeInsertion, which has been used as base for all experiments. Then we evaluate whether increasing t_k by some constant factor can reduce the number of comparisons. Last we evaluate some techniques proposed by others to improve MergeInsertion. One is splitting the input in two pieces, sorting them separately and merging the afterwards. The other is the combination with the 1-2-Insertion algorithm.

3.1 Implementing MergeInsertion

To perform experiments we first need to implement the algorithm. For the purpose of our implementation we assume that each element is unique. This condition is easy to fulfill for synthetic test data. You can see our implementation in Algorithm 3.1. We now go over some of the key challenges when implementing MergeInsertion.

1. MergeInsertion requires elements to be inserted into arbitrary positions. When using a simple array to store the elements this operation requires moving $O(n)$ elements. Since MergeInsertion inserts each element exactly once this results in a complexity of $O(n^2)$. To avoid this we store the elements in a custom data structure inspired by the Rope data structure[BAP95] used in text processing. Being based on a tree it offers $O(\log n)$ performance for lookup, insertion and deletion operations, thus putting our Algorithm in $O(n \log^2 n)$.

2. In the second step of the algorithm we need to rename the b_i after the recursive call. Our chosen solution is to store which a_i corresponds to which b_i in a hash map(line 11) before the recursive call and use the information to reorder the b_i afterwards(line 13). The disadvantage of this solution is that it requires each element to be unique and the hash map might introduce additional comparisons.

   An alternative would be to have the recursive call generate the permutation it applies to the larger elements and then apply that to the smaller ones. That is a cleaner solution as it does not require the elements to be unique and it avoids potentially introducing additional comparisons. It is also potentially faster, though not by much. However we stuck with using a hash map as that solution is easier to implement.

3. In the insertion step we need to know into how many elements a specific b_i is inserted. For b_{t_k} this is $2^k - 1$ elements. However for other elements that number can be smaller depending on where the previous elements have been inserted. To account for that we create the variable u in line 21. It holds the position of the a_i corresponding to the element b_i that is inserted next. Thus b_i is inserted into $u-1$ elements (since $b_i < a_i$).

Algorithm 3.1 MergeInsertion

```
1: procedure MergeInsertion(d : array of n elements)
2:     Step 1: Pairwise comparison
3:     for all 1 ≤ i ≤ ⌊n/2⌋ do                          // Split into larger and smaller half
4:         a_i ← max{d_i, a_{i+⌊n/2⌋}}
5:         b_i ← min{d_i, a_{i+⌊n/2⌋}}
6:     end for
7:     if n mod 2 = 1 then
8:         b_{⌈n/2⌉} ← d_n
9:     end if
10:    Step 2: Recursion and Renaming
11:    m ← {(a_i, b_i) | 1 ≤ i ≤ ⌊n/2⌋}                  // Store mapping
12:    a ←MergeInsertion(a)
13:    for all 1 ≤ i ≤ ⌊n/2⌋ do                          // Permute smaller half
14:        b_i ← e where (a_i, e) ∈ m
15:    end for
16:    Step 3: Insertion
17:    d ← b_1, a_1, ..., a_{⌊n/2⌋}
18:    k ← 2
19:    while t_{k−1} < ⌈n/2⌉ do
20:        m ← min{t_k, ⌈n/2⌉}                           // first element of the batch
21:        u ← t_{k−1} + m                               // position of a_m in d
22:        for i in m down to t_{k−1} + 1 do
23:            d ←BinaryInsertion(b_i, d_1, ..., d_{u−1}), d_u, ..., d_{2m+t_{k−1}−i}
24:            while d_u ≠ a_{i−1} do                    // adjust u
25:                u ← u − 1
26:            end while
27:        end for
28:        k ← k + 1
29:    end while
30:    return d
31: end procedure
```

After the insertion of b_i, we decrease u in line 25 until it matches the position of a_{i-1}, which is what we want as b_{i-1} is the next element to be inserted. This step also makes use of the requirement that each element is unique.

At this point we have to be aware that testing whether the element at position u is a_{i-1} might introduce additional comparisons to the algorithm. This is acceptable because we do not count these comparisons. Also these are not necessary. We could keep track of the positions of the elements a_i however we choose not to, in order to keep the implementation simple.

Figure 3.1 shows experimental results on the number of comparisons required by MergeInsertion and compares them to the upper bound from Section 2.4.

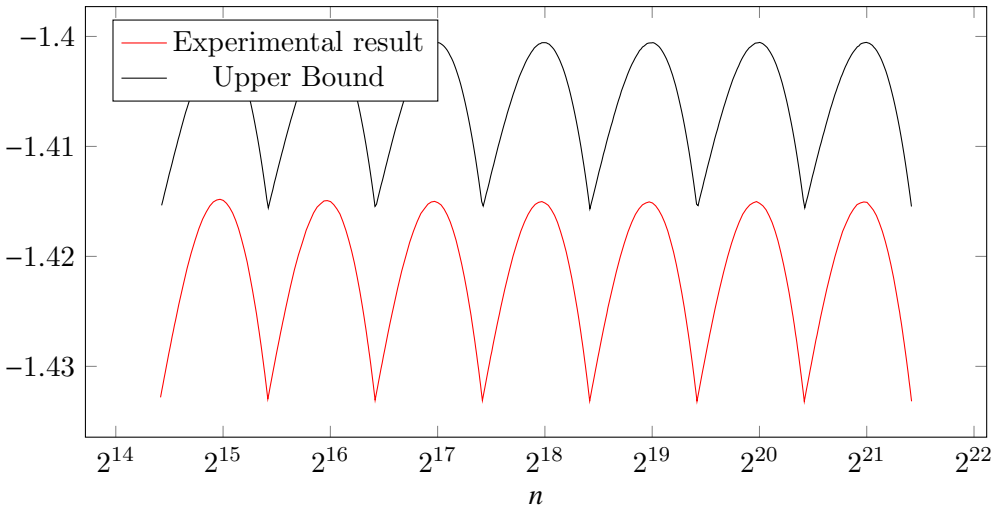


Figure 3.1: Comparing experimental results with the approximation from Section 2.4

Note that all experiments use the `left` strategy for binary insertion as it was determined to be best in Section 2.2. All experiments have been averaged over 10 to 10000 runs, depending on the size of the input.

3.2 Increasing t_k by a Constant Factor

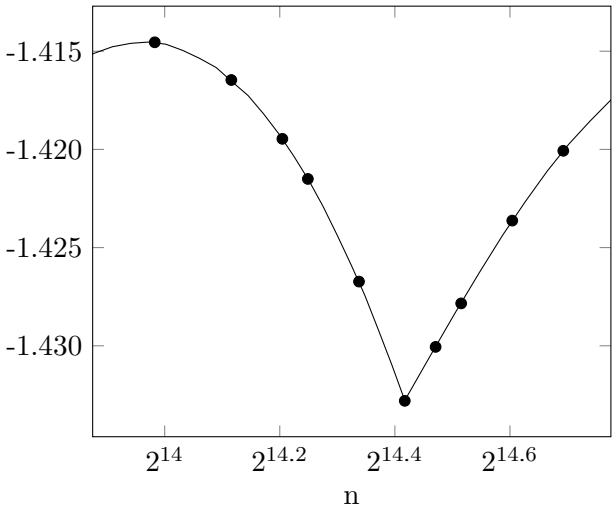


Figure 3.2: Values of n for which different factors f are evaluated in Figure 3.3.

In this section we replace t_k with

$$\hat{t_k} = \lfloor f \cdot t_k \rfloor \tag{3.1}$$

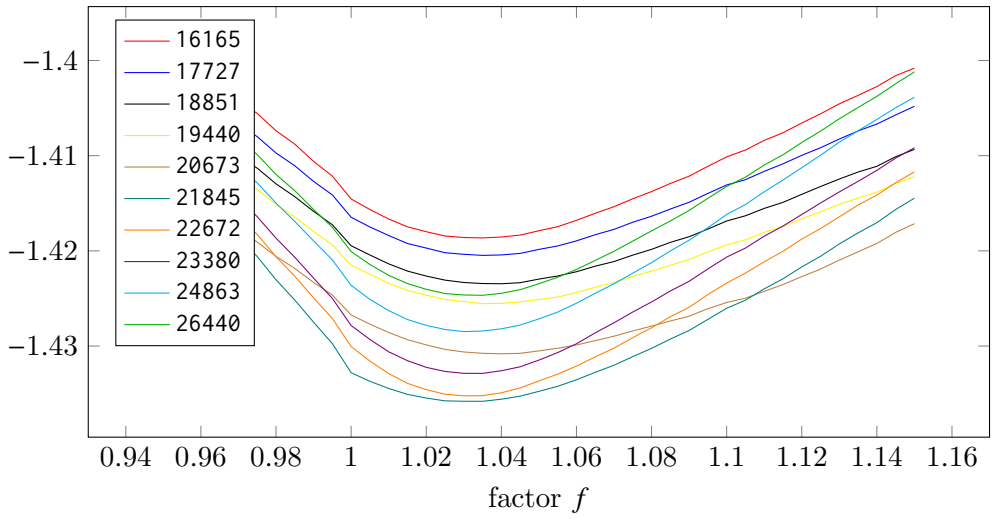


Figure 3.3: Effects of replacing t_k with $\hat{t_k}$.

Originally the numbers t_k have been chosen, such that each element b_i with $t_{k-1} < i < t_k$ is inserted into at most $2^k - 1$ elements. As we have seen in previous chapters many elements are inserted into slightly less than $2^k - 1$ elements. The idea behind increasing t_k by a constant factor f is to allow more elements to be inserted into close to $2^k - 1$ elements.

Figure 3.3 shows how different factors f affect the number of comparisons required by MergeInsertion. The different lines represent different input lengths. For instance 21845 is an input for which MergeInsertion is optimal. An overview of the different input lengths and how MergeInsertion performs for these can be seen in Figure 3.2. We assume the chosen values to be representative for the entire algorithm. We observe that for all shown input lengths, multiplying t_k by a factor f between 1.02 and 1.05, leads to an improvement.

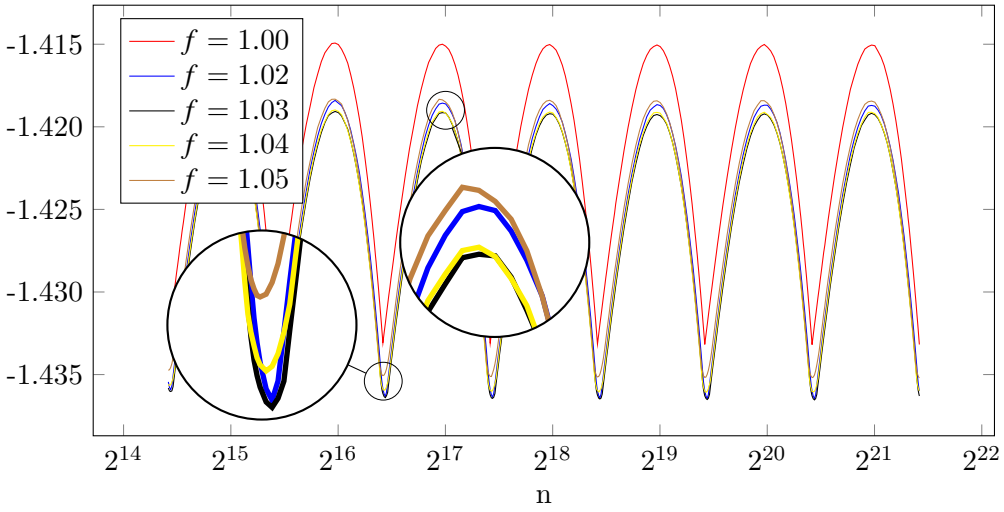


Figure 3.4: Comparison of different factors f for $\hat{t_k}$.

Figure 3.4 compares different factors from 1.02 to 1.05. The factor 1.0 is also included as a reference. We observe that all the other factors lead to a large improvement compared to 1.0. The difference between the factors in the chosen range is rather small. However 1.03 appears to be best out of the tested values.

Another observation we make from Figure 3.4 is that the plot periodically repeats itself with each power of two. This means that the improvement that we achieve is a constant per element.

From these experiments we conclude that replacing t_k with $\hat{t_k}$ with $f \in [1.02, 1.05]$ reduces the number of comparisons required per element by some constant. The differences between the individual factors in the given range are rather small, though 1.03 appears to be close to the optimum.

3.3 Merging

In this section we evaluate how splitting the input into two parts sorting them separately and then merging the sorted parts does affect the average case of the algorithm. The idea was presented first in [Man79]. It relies on the fact that there are specific points $u_k = \left\lfloor \left(\frac{4}{3}\right) 2^k \right\rfloor$ where MergeInsertion is optimal. The input is split into two parts with size m_1 and m_2 as defined in (3.2). As a result we have $m_1 > m_2$ and m_1 is of a size for with MergeInsertion is optimal.

$$\begin{aligned} m_1 &= \max\{u_k \mid u_k \leq n\} \\ m_2 &= n - m_1 \end{aligned} \tag{3.2}$$

For merging the two lists we use the Hwang-Lin Algorithm[HL72], for it is a simple general purpose merging algorithm. It works by comparing the first element of the small list with the 2^r-th element of the large list, where $r = \left\lfloor \log \frac{m_1}{m_2} \right\rfloor$. If it is larger then the 2^r first elements of the large list are removed and appended to the result, and the process is repeated. If it is smaller then it is inserted into the $2^r - 1$ first elements of the large list using binary insertion and then all elements up to the freshly inserted one are removed from the large list and appended to the result. This is repeated until one of the lists is exhausted.

To be precise, we actually use a variant of the Hwang-Lin Algorithm called Static Hwang-Lin Algorithm presented in [Man79]. The difference is that $r = \left\lfloor \log \frac{m_1}{m_2} \right\rfloor$ is only calculated once (hence "static") instead of every iteration. This leads to slightly improved performance in cases where $\frac{m_1}{m_2}$ is close to a power of 2.

Both [Man79] and [BT85] present more advanced merging algorithms which would yield better results than the ones we have obtained. However they are also more complicated than the Hwang-Lin Algorithm so we did not implement them. Even without using those advanced merging algorithms our experiments show that this technique does indeed improve upon MergeInsertion.

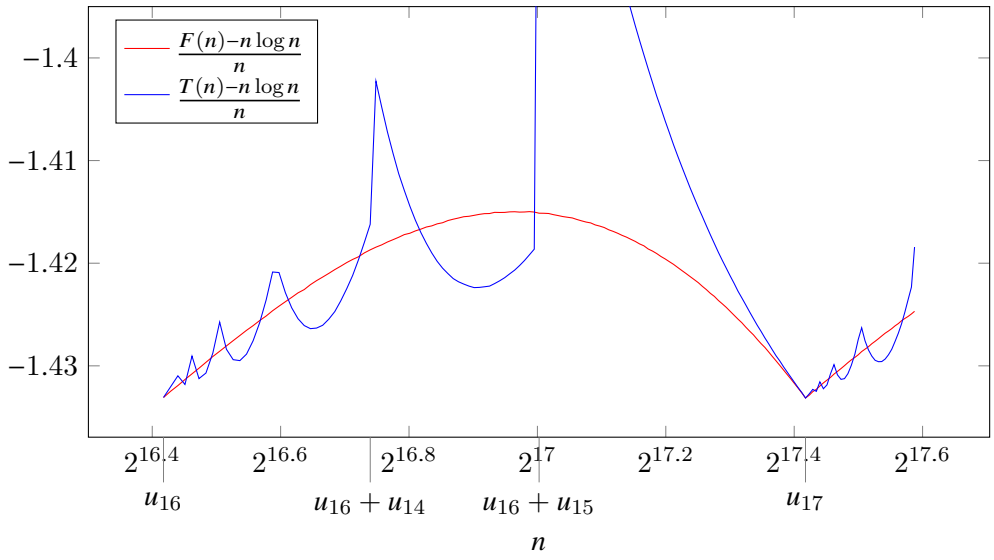


Figure 3.5: Experimental results

The results of our experiments are presented in Figure 3.5. $F(n)$ denotes the number of comparisons required by MergeInsertion in the average-case and $T(n)$ is the number of comparisons required when splitting the list into two parts sorting them separately and merging them afterwards as explained above. As we can see $T(n)$ beats $F(n)$ for some ranges of n.

However we also observe that the points where $T(n)$ is best are not those where m_2 is optimal for MergeInsertion. Instead this happens where m_2 is a bit smaller than what would be optimal for MergeInsertion. From this we conclude that $T(n)$ is defined mostly by cost of merging m_1 and m_2 and that the cost of MergeInsertion for m_2 plays only a minor role.

3.4 Combination with 1-2-Insertion

1-2-Insertion is a sorting algorithm presented in [IT17]. It works by inserting either a single element or two elements at once into an already sorted list.

On its own 1-2-Insertion is worse than MergeInsertion, however it can be combined with MergeInsertion. The combined algorithm works by sorting $m = \max\{u_k \mid u_k \leq n\}$ elements with MergeInsertion. Then the remaining elements are inserted using 1-2-Insertion.

In Figure 3.6 we can see that at the point u_k MergeInsertion and the Combined Algorithm perform the same. However in the values following u_k the Combined Algorithm surpasses MergeInsertion until at one point close to the next optimum MergeInsertion is better once again.

In their paper Iwama and Teruyama calculated that for $0.638 \leq \frac{n}{2^{\lceil \log n \rceil}} \leq \frac{2}{3}$ MergeInsertion is better than the Combined Algorithm. The fraction $\frac{2}{3}$ corresponds to the point where MergeInsertion is optimal. The constant 0.638 they derived from their theoretical analysis using the upper bound for MergeInsertion presented in [EW13].

Comparing this to our experimental results we observe that the range where MergeInsertion is better than the Combined Algorithm starts at $n \approx 2^{17.242}$. This yields $\frac{2^{17.242}}{2^{18}} = 2^{17.242-18} = 2^{-0.758} \approx 0.591$. Hence the range where MergeInsertion is better than the Combined Algorithm is $0.591 \leq \frac{n}{2^{\lceil \log n \rceil}} \leq \frac{2}{3}$, which is slightly larger than the theoretical analysis suggested.

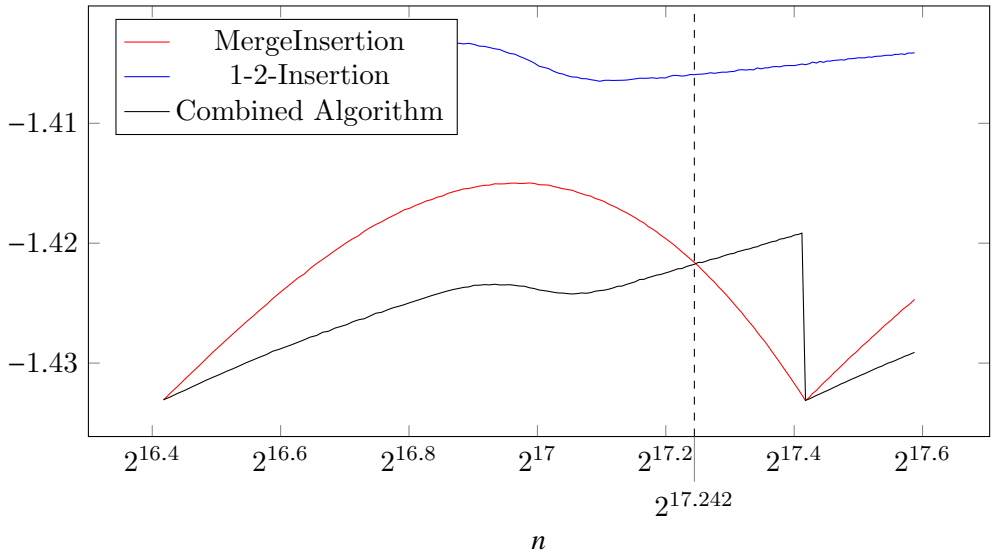


Figure 3.6: Experimental results comparing MergeInsertion, 1-2-Insertion and the Combined Algorithm

4 Conclusion and Outlook

This thesis has studied the average case behavior of MergeInsertion. An in depth analysis of the insertion step led to a closed form equation of the probability $P(Y_i = j)$ that $b_{t_{k-1}+i}$ is inserted into exactly j elements. This probability is described by Equation (2.8) and allows us to compute a good approximation for the average-case of MergeInsertion.

However that formula is pretty complicated and thus unsuitable for further analysis. An approximation based on a binomial distribution was presented, which led to an upper bound of $n \log n - 1.4005n + o(n)$ for the average-case. This improves upon the best previously known upper bound of $n \log n - 1.3999n + o(n)$.

Furthermore an algorithm for calculating the exact number of comparisons required in the average case for small values of n has been presented. Using that algorithm we were able to calculate the exact number of comparisons for all $n \leq 148$.

Concluding our study of MergeInsertion, a series of experiments has been performed, comparing our upper bound to the actual number of comparisons required by MergeInsertion as well as evaluating different ideas for reducing the number of comparisons.

Our suggestion of increasing t_k by a constant factor f appears to reduce the number of comparisons required per element by some constant. Different values for f have been tested and we found that $f = 1.03$ appears to be close to optimal.

Splitting the input into two parts, sorting them separately and merging the sorted parts afterwards as proposed by [Man79] has been shown to lead to an improvement in the average-case too. And this is even using the Hwang-Lin Algorithm for merging instead of any of the more advanced algorithms presented in [Man79] and [BT85].

The combination of MergeInsertion with 1-2-Insertion has been implemented. This Combination beats MergeInsertion for infinitely many n. We have experimentally evaluated for which n this happens and compared these results with the theoretical analysis presented in [IT17].

Outlook

There still is a gap between the number of comparisons required by MergeInsertion and the presented upper bound. Further analysis might be able to close that gap.

- In the insertion step the likelihood of an element being inserted into a specific position is not the same for all positions. Since we do not know that probability distribution we created an upper bound for the cost of inserting one element with binary insertion using a uniform distribution. Knowledge of that probability distribution would potentially allow for creating a better approximation for the average case.
- In Section 2.4 we used a binomial distribution to approximate the probability of an element being inserted into a specific number of elements during the insertion step. However the difference between our approximation and the actual probability distribution is rather large. Finding an approximation which reduces that gap while still being simple to analyze with respect to its mean would facilitate further improvements to the upper bound.

The presented idea of improving MergeInsertion by increasing t_k by a constant factor has only been studied experimentally. It could be formally analyzed to determine the optimal value for the factor f as well as to study how this suggestion affects the worst-case.

Bibliography

[BAP95] H.-J. Boehm, R. Atkinson, M. Plass. "Ropes: An Alternative to Strings". In: *Softw. Pract. Exper.* 25.12 (Dec. 1995), pp. 1315–1330. ISSN: 0038-0644. DOI: 10.1002/spe.4380251203. URL: http://dx.doi.org/10.1002/spe.4380251203 (cit. on p. 35).

[BT85] T. Bui, M. Thanh. "Significant improvements to the Ford-Johnson algorithm for sorting". In: *BIT Numerical Mathematics* 25.1 (1985), pp. 70–75 (cit. on pp. 8, 39, 43).

[EW13] S. Edelkamp, A. WeiSS. "QuickXsort: Efficient Sorting with n log n - 1.399n +o(n) Comparisons on Average". In: *CoRR* abs/1307.3033 (2013). arXiv: 1307.3033. URL: http://arxiv.org/abs/1307.3033 (cit. on pp. 7, 8, 22, 26, 28, 41).

[HL72] F. K. Hwang, S. Lin. "A simple algorithm for merging two disjoint linearly ordered sets". In: *SIAM Journal on Computing* 1.1 (1972), pp. 31–39 (cit. on p. 39).

[IT17] K. Iwama, J. Teruyama. "Improved average complexity for comparison-based sorting". In: *Workshop on Algorithms and Data Structures*. Springer. 2017, pp. 485–496 (cit. on pp. 8, 40, 41, 43).

[JJ59] L. R. F. Jr., S. M. Johnson. "A Tournament Problem". In: *The American Mathematical Monthly* 66.5 (1959), pp. 387–389. DOI: 10.1080/00029890.1959.11989306. eprint: https://doi.org/10.1080/00029890.1959.11989306. URL: https://doi.org/10.1080/00029890.1959.11989306 (cit. on p. 7).

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0 (cit. on pp. 7, 8, 22, 26).

[Man79] G. K. Manacher. "The Ford-Johnson Sorting Algorithm Is Not Optimal". In: *J. ACM* 26.3 (July 1979), pp. 441–456. ISSN: 0004-5411. DOI: 10.1145/322139.322145. URL: http://doi.acm.org/10.1145/322139.322145 (cit. on pp. 8, 39, 43).

[OEIS] *The On-Line Encyclopedia of Integer Sequences. A001498*. URL: https://oeis.org/A001498 (cit. on p. 15).

All links were last followed on November 27, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature