Data-Specific Analysis of String Sorting

Raimund Seidel*

Abstract

We consider the complexity of sorting strings in the model that counts comparisons between symbols and not just comparisons between strings. We show that for any set of strings S the complexity of sorting S can naturally be expressed in terms of the trie induced by S. This holds not only for lower bounds but also for the running times of various algorithms. Thus this "data-specific" analysis allows a direct comparison of different algorithms running on the same data. We give such "data-specific" analyses for various versions of quicksort and versions of mergesort. As a corollary we arrive at a very simple analysis of quicksorting random strings, which so far required rather sophisticated mathematical tools. As part of this we provide insights in the analysis of tries of random strings which may be interesting in their own right.

1 Introduction

The complexity of sorting n keys is usually measured in terms of the number of comparisons between keys: $O(n \log n)$ comparisons suffice, if you use one the optimal algorithms such as Mergesort or Heapsort, and $\Omega(n \log n)$ comparisons are necessary in the worst case, if you restrict yourself to key-comparison based algorithms. These are standard results taught in most introductory algorithms courses.

If the keys to be sorted are strings and the order relation is the usual lexicographic order, then this simple complexity measure of counting key comparisons does not adequately reflect the actual running time of sorting algorithms. Lexicographically comparing two strings takes time proportional to the length of their longest common prefix plus one, since this many symbols have to be compared in order to resolve the comparison. Thus a more appropriate measure for the complexity of sorting strings is the number of *symbol comparisons* performed.

However it is not clear at all how many comparisons in a run of a sorting algorithm involve keys with a long common prefix, hence take a long time, and how many comparisons involve keys with short longest common prefix, hence take only little time. As a matter of fact, it is not even clear what parameters should be used to express this symbol comparison complexity of a string sorting algorithm. Just n, the number of strings to be sorted, is certainly inadequate: by adding a common prefix to all n strings of a set you can increase the symbol comparison complexity of an algorithm arbitrarily while n stays the same. Introducing m, the sum of all the string lengths, as a second parameter is not really that helpful either: you still cannot distinguish between adding a common prefix to all strings, which forces many additional symbol comparisons, from adding a common suffix to all strings, which causes no additional symbol comparisons.

This problem of how to parametrize the symbol comparison complexity of string sorting algorithms was bypassed in two ways. The first was the development of specialized string sorting algorithms that avoid full lexicographic comparisons between strings and achieve a symbol comparison complexity of $O(m + n \log n)$, which can be argued to be worst case optimal (in a comparison based model), see e.g. [1] or [9, section III. 6.3]. The second was to assume that the n strings are generated by some random process and to analyze a general, key comparison based algorithm, in particular Quicksort, for its average performance with respect to symbol comparison complexity [5, 14].

Both approaches are unsatisfying: The first one, since frequently general purpose sorting routines (such as the sort command in UNIX) are used to sort strings and not specialized routines. The second one, since in most cases it is very difficult to justify the assumption that a particular set of input strings can be viewed as generated by a particular random process.

This paper is based on the simple observation that when sorting a set S of n strings using the usual lexicographic comparisons the entire common prefix structure of the strings in S ought to be used to describe the number of symbol comparisons performed. Moreover, this common prefix structure is encoded by the trie $T(S)$ induced by S. Such a trie is a tree that has as its node set the set $P(S)$ of all prefixes of strings

*Universität des Saarlandes, Fachrichtung Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. rseidel@cs.uni-saarland.de

in S, where v is a parent of w iff string v is a prefix of w and v is one symbol shorter than w. We define the *thickness* $\vartheta(w)$ *of a node* w to be the number of leaves in the subtree of $T(S)$ rooted at w. In other words, $\vartheta(w)$ is the number of strings in S that have w as a prefix. In this paper tries are not used as data structures but as mathematical objects that facilitate complexity analysis. We define the *reduced trie* $\widehat{T}(S)$ to be the subtree of $T(S)$ induced by the node set $\widehat{P}(S) = \{w \in P(S) \,|\, \vartheta(w) > 1\}$.

The vector $\big(\vartheta(w)\big)_{w\in P(S)}$ seems to determine the complexity of sorting S: the running time of many sorting algorithms when applied to S can be expressed in terms of this vector. So far this only seems to have been observed for radix sort. It is also possible to prove a lower bound for the number of symbol comparisons necessary to sort S that is in terms of this vector.

Here are some of our results. We use the notation $H_n = \sum_{1\le i\le n} 1/i \approx 1 + \log n$. The *standard string comparison procedure* compares two strings symbol by symbol until the first difference is found. We assume that the input string set S is presented in random order with each permutation equally likely. This assumption is innocuous since we can precede the actual sorting by a linear time randomized procedure that permutes S randomly.

THEOREM 1.1. *Let* $Q(n) = 2(n+1)H_n - 4n$ *be the expected number of key comparisons performed by quicksort when sorting a set of* n *keys.*

Assume quicksort is applied to string set S *employing the standard string comparison procedure. Then the expected number of symbol comparisons performed is* exactly

$$\sum_{w\in P(S)} Q(\vartheta(w))\,.$$

As a rather straightfoward consequence of this theorem we get that quicksorting n strings over an alphabet of size a takes, up to lower order terms, in expectation *at least* $(n\log^2 n)/\log a$ symbol comparisons, since the expression in the theorem is minimized when the trie for S is as balanced as possible.

Also note that if we are dealing with a string set S generated by some random process, then each $\vartheta(w)$ (and hence $Q(\vartheta(w))$ is a random variable of this process, and in order to study the expected complexity of quicksorting such a random set it suffices to study $\mathrm{Ex}[Q(\vartheta(w))]$ and exploit the linearity of expectations. This observation leads to an analysis of the average complexity of quicksort when applied to uniformly generated random strings that is substantially simpler than the one presented in [5].

THEOREM 1.2. *Let* $R(n) = n\log_2 n$. *Assume mergesort is applied to a set* S *of strings employing the standard string comparison procedure.*

Then the expected number of symbol comparisons performed is at most

$$\sum_{w\in P(S)} R(\vartheta(w))\,.$$

Let us remind you that we assume that we initially randomly permute the set S. Thus, although mergesort is deterministic, it still makes sense to talk about expected performance. Also note that Theorems 1 and 2 immediately imply that in expectation for *any* set of strings mergesort makes fewer symbol comparisons than quicksort, as n goes to infinity by a factor of $2\log 2 \approx 1.39$. This immediately implies the following:

COROLLARY 1.1. *All the expected running time bounds proved for quicksort in [5] and [14] carry over to mergesort with the constant for the leading term improved by a factor of* $2\log 2 \approx 1.39$.

THEOREM 1.3. *Let* $\lg n = \lceil \log_2 n \rceil$ *and let* $L(n) = \lg(n!)$, *the information theoretic lower bound for sorting by key comparisons. Let* S *be a set of* n *strings.*

1. *For any deterministic comparison-based sorting algorithm* A *that is used to sort* S *using the standard string comparison procedure, there is a permutation of* S *that forces* A *to perform at least*

$$\sum_{w\in\widehat{P}(S)} \max\{\, L(\vartheta(w)) - \lg n\,,\, \vartheta(w) - 1\}$$

symbol comparisons.

2. *For any randomized comparison-based sorting algorithm* A *that is used to sort* S *using the standard string comparison procedure, there is a permutation of* S *that forces* A *to perform at least*

$$\sum_{w\in\widehat{P}(S)} \max\{\, L(\vartheta(w)) - 3\cdot\lg n\,,\, \vartheta(w) - 1\}$$

symbol comparisons in expectation.

The dominant term in the summands of this theorem is $L(\vartheta(w))$. The other terms only play a role if in $T(S)$ there are exceedingly long branches of nodes with small thickness. (Most likely these terms are just artifacts of our proof anyway.) The functions $L(n)$, $R(n)$, and $Q(n)$ are all essentially proportional to $n\log n$. Thus Theorem 1.3 tells that considering algorithms that employ the standard string comparison procedure quicksort and mergesort are about as good as possible.

Also note that considering sets S of n distinct strings over an alphabet of size a the sums in Theorem 1.3 are again minimized when $T(S)$ is as balanced as possible, i.e. S consists of n strings of length about $\log_a n$. This implies a lower bound of $\Omega((n \log^2 n)/\log a)$ for the number of symbol comparisons that any comparison based algorithm must use when sorting n strings just using the standard string comparison procedure. Such a lower bound in terms of n and independent of S has already been shown by Mikkel Thorup [13].

These lower bounds imply that when sorting strings it is suboptimal to just use a standard comparison based sorting algorithm employing the standard string comparison procedure. Derived knowledge about the length of common prefixes should be exploited. For quicksort there are at least two variations that achieve just that.

The first one is the algorithm presented by Bentley and Sedgewick in [1]. It tri-partitions[1] S just looking at the leading character of each string into $S_{<c}$, $S_{=c}$, and $S_{>c}$ for some pivot character c. These three sets are then sorted recursively looking at the full strings in the case of $S_{<c}$ and $S_{>c}$, but in case of $S_{=c}$ only looking at the suffixes that just exclude the first character.

Our second variation of quicksort performs standard lexicographic comparisons between strings. However, if two strings are known to have some common prefix of length k, then the lexicographic comparison starts with the $(k+1)$-st character. We employ only one mechanism for discovering common prefixes of strings: if a subarray is to be sorted that is framed by two previous pivots and they have a common prefix w of length k, then all strings in the subarray have w as a common prefix of length k. We leave further details to the reader. Let us refer to this algorithm as the *lazy quicksort variation.*

THEOREM 1.4. *Let $Q(n) = 2(n+1)H_n - 4n$ again be the expected number of key comparisons performed by quicksort when sorting a set of n keys. Let S be a set of n strings. Let $P_0(S)$ denote $P(S)$ without the root of $T(S)$.*

1. *If the Bentley-Sedgewick variation of quicksort is used to sort S, then the expected number of symbol comparisons is* exactly
$$Q(n) + \sum_{w \in P_0(S)} (\vartheta(w) - 1)\,.$$
2. *If the lazy variation of quicksort is used to sort S, then the expected number of symbol comparisons is* exactly
$$Q(n) + \sum_{w \in P_0(S)} 2(\vartheta(w) - H_{\vartheta(w)})\,.$$

It is also possible to alter mergesort so as to take advantage of common prefixes. The trick is to maintain sorted lists of strings that are augmented in that each string x stores with it the length $\ell[x]$ of the longest common prefix with its predecessor in the list. This information can be easily maintained when merging two such sorted lists X and Y into list Z. Simply maintain the invariant that if x and y are the first elements of the current lists X and Y and z is the last element of Z, then $\ell[x] = \mathrm{lcp}(x, z)$ and $\ell[y] = \mathrm{lcp}(y, z)$, where $\mathrm{lcp}(u, v)$ denotes the length of the longest common prefix of u and v. Then x and y are guaranteed to have a common prefix of length at least $k = \min\{\ell[x], \ell[y]\}$ and the lexicographic comparison between x and y can be started at symbol $k+1$. This comparison will determine $\mathrm{lcp}(x, y)$, and the $\ell[]$ value of the winning string (that does not go into Z) can be set accordingly to maintain the invariant. We again leave further details to the reader. Let us call this version of mergesort *lazy mergesort.* A similar method was proposed in [10].

THEOREM 1.5. *Let $M(n) \le n\lceil \log_2 n \rceil$ be an upper bound for the number of key comparisons performed when mergesorting n keys. Let S be a set of n strings.*

If lazy mergesort is applied to sort S, then the number of symbol comparisons ist at most
$$M(n) + \sum_{w \in P_0(S)} (\vartheta(w) - 1)\,.$$

Note that $\sum_{w \in P(S)} (\vartheta(w) - 1)$ is at most $m(S)$, the sum of the lengths of the strings in S. Thus Theorems 1.4 and 1.5 yield optimal $O(n \log n + m(S))$ bounds in the traditional sense. Also note that the bounds for lazy mergesort are again better by a constant factor than the bounds for the two versions of quicksort. Note however, that these bound just count comparisons between symbols. Lazy mergesort also performs comparison between prefix lengths (although not more than $M(n)$ of them), and comparisons of this kind are ignored in our model.

Note that specific algorithms are not the main message of this paper. We believe the main insight is that for some problems simple instance characteristics such as number of strings and total size are not necessarily

[1] Throughout the paper we assume that a symbol comparison produces in one step 3 possible outcomes: <,=, or >.

that well suited to describe the computational complexity of the problem or the performance of an algorithm. Much more informative descriptors may exist that allow a much more fine-grained distinction between instances. For the case of string sorting we argue that the thickness vector $\big(\vartheta(w)\big)_{w\in P(S)}$ constitutes such a descriptor. Since this vector just depends on the data S and not on the presentation (i.e. ordering) of the data we talk about *data-specific analysis.*

We are aware of at least one other case of data-specific analysis, although it was not called this way. Randomized incremental algorithms [2, 3] to contruct the convex hull of a set S of n points in $\mathbb{R}^d$ are usually analyzed with respect to the vector $\big(f_r(S)\big)_{1\le r\le n}$, where $f_r(S)$ denotes the expected number of facets of the convex hull of a random subset of S of size r. Again this vector just depends on S and not on its presentation. It remains to be seen what other instances of data-specific analysis exist.

2 Quicksort and some variants

First we want to analyze the expected number of symbol comparisons made by quicksort when applied to a set S of strings using the standard string comparison procedure and thus prove Theorem 1.1.

Here is an abstract description of quicksort, when applied to a set X of n distinct keys: It runs in n stages, and it maintains the invariant that after stage r there is a "partition" of X of the form

$$B_0 < p_1 < B_1 < p_2 < B_2 < \cdots < p_{r-1} < B_{r-1} < p_r < B_r$$

with r "pivots" $p_1 < p_2 < \cdots < p_r$ from X and $r+1$ "bags" $B_0, \ldots, B_r \subset X$, where $p < B$ denotes that for each $b \in B$ we have $p < b$ (and likewise $B < p$). Initially we have $B_0 = X$ and in the end all B_i's are empty, leaving X as a sorted sequence of pivots.

You proceed from stage $r-1$ to stage r by arbitrarily choosing a non-empty bag B, choosing an element $p \in B$ uniformly at random, comparing p to every other element of B, thus dividing B into $B_{<p}, p, B_{>p}$, and replacing B in the current "partition" by $B_{<p} < p < B_{>p}$.

Let us write X as $\{x_1, \ldots, x_n\}$ with $x_1 < x_2 < \cdots < x_n$. For $i < j$ the keys x_i and x_j are compared during a run of quicksort iff among $x_i, x_{i+1}, \ldots, x_j$ either x_i or x_j were the first to ever be chosen as a pivot. This happens with probability $2/(j-i+1)$ and thus we get that C_{ij}, the expected number between x_i and x_j, is $2/(j-i+1)$. The expected number of all key comparisons during a run of quicksort is then $\sum_{1\le i<j\le n} C_{ij}$, which evaluates to $Q(n)$ as given in Theorem 1.1. All this is of course well known. But we will need the setup, when we analyze our lazy version of quicksort.

Proof. (of Theorem 1.1): Let S be a set of n strings $s_1 < s_2 < \cdots < s_n$, and we apply quicksort to it using the standard string comparison procedure. Such a standard string comparison consists of an unknown number of symbol comparisons. Let us classify symbol comparisons by their outcome: it the two symbols compared turn out to be equal we call this comparison a *tie*, otherwise we call it a *win*. A standard string comparison then consists of an unknown number of ties followed by exactly one win. We immediately get that the expected number of wins when quicksorting S is exaclty the same as the number of string comparisons, and hence it is $Q(n)$.

It remains to estimate the expected number of ties. Note that a tie can only happen when the ℓ-th symbols of two strings are compared and these two strings have a common prefix of length $\ell > 0$. For a word w of length ℓ let S_w be the strings in S with prefix w. Let us ignore the uninteresting case $|S_w| \le 1$. The set S_w must be $\{s_h, \ldots, s_k\}$ for some $h < k$ with $k - h + 1 = \vartheta(w)$, considering the trie $T(S)$. For $h \le i < j \le k$ a tie happens when comparing the ℓ-th symbol of s_i and s_j iff these two strings are compared. But we know this to happen with probability C_{ij}. Thus the expected number of ties when comparing ℓ-th symbols of strings in S_w is exactly $\sum_{h\le i<j\le k} C_{ij} = Q(k-h+1) = Q(\vartheta(w))$.

Summing over all prefixes we get that the expected total number of ties is exactly[2] $\sum_{w\in\widehat{P}(S)\setminus\{\varepsilon\}} Q(\vartheta(w))$. Since $Q(1) = 0$ and since $\vartheta(\varepsilon) = n$ and hence $Q(\vartheta(\varepsilon))$ can be taken to count the expected number of wins, we get that the total expected number of symbol comparisons is the claimed $\sum_{w\in P(S)} Q(\vartheta(w))$.

The essence of the previous proof is really captured by the following: Call a comparison-based sorting algorithm *$f(t)$-faithful* if, when sorting a set X of randomly permuted keys $x_1 < x_2 < \cdots < x_N$, for any $0 < t \le N$ and for any "range" $X_{(i,i+t]} = \{x_{i+1}, \ldots, x_{i+t}\}$ of t order-consecutive elements of X the expected number of comparisons among elements of $X_{(i,i+t]}$ is at most $f(t)$. Call it *strongly $f(t)$-faithful* if this expected number of comparisons among elements of $X_{(i,i+t]}$ is *exactly* $f(t)$.

LEMMA 2.1. *If an $f(t)$-faithful (strongly $f(t)$-faithful) sorting algorithm is used to sort a set S of randomly permuted strings using the standard string comparison procedure, then the expected number of symbol compar-*

[2] Recall that the empty string ε is the root of the trie $T(S)$ and that $\widehat{P}(S) = \{w \in P(S) | \vartheta(w) > 1\}$.

isons performed is at most (exactly)

$$\sum_{w \in \widehat{P}(S)} f(\vartheta(w)) .$$

The proof of this lemma is analogous to the proof just given. Obviously quicksort is strongly $Q(t)$-faithful. As we shall see later, mergesort is $O(n \log n)$-faithful.

Let us now turn to the versions of quicksort that are tailored towards string sorting.

Proof. (of Theorem 1.4): We again argue about wins and ties separately. In both versions of quicksort to be considered the expected number of wins is $Q(n)$. Thus it just remains to argue about the number of ties.

Consider the Bentley-Sedgewick version first. It actually can be seen as (at least implicitly) computing the trie $T(S)$. Whenever a tri-partitioning step occurs producing three stringsets $U_{<c}$, $U_{=c}$, and $U_{>c}$ for some pivot character c, then all the strings have a common prefix v and $U_{=c}$ is exactly S_{vc}, the set of all strings in S with prefix vc. Determining U_c then took exactly $|S_{vc}| - 1 = \vartheta(vc) - 1$ symbol comparisons, all of which were ties. This must happen for every non-empty prefix in S, and the first part of the theorem follows.

Now consider our lazy quicksort variation. Again we just need to count ties. Let w be some prefix of length $\ell > 0$ and let $S_w = \{s_h < \cdots < s_k\}$ for some $h \leq k$ with $k - h + 1 = \vartheta = \vartheta(w)$, as in the proof of Theorem 1.1. We want to count the number of symbol comparisons that happen between the ℓ-th symbols of the strings in S_w. If we consider all possible prefixes this will count all ties.

Renumber S_w into $s_1 < s_2 < \cdots < s_\vartheta$. How can such a symbol comparison happen with s_k as pivot? Let us just consider such comparisons with smaller string s_i, i.e. where $1 \leq i < k$. By the way our algorithm discovers common prefixes this happens iff among $\{s_1, \ldots, s_k\}$ the string s_k was the first to be chosen as a pivot. This happens with probability $1/k$. In that case s_k is compared to each of the $k - 1$ strings s_i with $i < k$, each resulting in a tie in the ℓ-th symbol. This yields an expected number of $(k-1)/k = 1 - 1/k$ such ties involving s_k. Summing over all k yields an expected number of $\vartheta - H_\vartheta$ such ties. By symmetry we get the same number for the ties between a pivot s_k and a string s_i with $k < i \leq \vartheta$. Summing over all nonempty prefixes yields the second part of the theorem.

3 Random strings

Let us make a short excursion into the study of tries of random strings. There are several reasons. Firstly, we can reprove some of the results in [5, 14, 4], arguably in a much simpler way. Secondly, such tries turn up in our analysis of mergesort. And finally, we are interested in making this paper self-contained and its results accessible to, say, a typical graduate algorithms class. It turns out that textbooks that deal with tries of random strings (e.g. [7, 11, 8, 12]) in many cases present the analyses as a first applications of quite sophisticated methods of asymptotic analysis. Mellin transforms and Rice's method are typically beyond the reach of a standard graduate algorithms course. Thus we will attempt to give simple, "quick and dirty" derivations based on a few insights that appear not to be that well known. This leads to surprisingly good bounds, in most cases a correct constant in front of the leading term. But of course this can in no way fully replace the sophisticated established methods which yield better bounds and in the case of complicated random sources are still the only known viable approach.

We will use a somewhat standard model. There is some random process (sometimes called source) that generates an infinite string over some alphabet A, which for the sake of simplicity we assume to be finite. We make only one assumption about this process: for any $w \in A^*$ there is a fixed probability p_w that the produced string has prefix w. We will consider a set S of N such strings that were produced independently by the same process. We will be interested in the reduced trie $\widehat{T}(S)$. Recall that this is the subgraph of $T(S)$ induced by all nodes of thickness greater than 1. We are interested in three random variables:

$$V(S) = \sum_{w \in \widehat{T}(S)} 1$$

$$K(S) = \sum_{w \in \widehat{T}(S)} \vartheta(w)$$

$$Z(S) = \sum_{w \in \widehat{T}(S)} \vartheta(w) \log \vartheta(w)$$

$V(S)$ counts the number of nodes in $\widehat{T}(S)$; by Theorem 1.1 the quantity $Z(S)$ can be used to bound the number of symbol comparisons performed when quicksorting S using the standard string comparison procedure; finally $K(S)$ is the sum of the thickness of the nodes, and at the same time it is the sum of the "string lengths" in the trie, sometimes also referred to as "external path length", a quantity that has been studied extensively in the context of tries.

We are interested in the expected values of these random variables. These random variables can also be represented as sums of random variables over all nodes

$w \in A^*$ of $T(A^*)$. For $w \in A^*$ define

$$V_w(S) = \begin{cases} 1 & \text{if } \vartheta(w) > 1, \\ 0 & \text{otherwise} \end{cases}$$

$$K_w(S) = \begin{cases} \vartheta(w) & \text{if } \vartheta(w) > 1, \\ 0 & \text{otherwise} \end{cases}$$

$$Z_w(S) = \begin{cases} \vartheta(w)\log\vartheta(w) & \text{if } \vartheta(w) > 1, \\ 0 & \text{otherwise,} \end{cases}$$

where $\vartheta(w)$ is now to be taken as a random variable depending on S, with the Bernoulli distribution

$$\Pr[\vartheta(w) = t] = b(p_w, N, t) = \binom{N}{t} p_w^t (1-p_w)^{N-t}.$$

Considering V we get that

$$\begin{aligned} \mathrm{Ex}[V_w] &= \sum_{2\le t\le N} b(p_w, N, t)\cdot 1 \\ &= 1 - (1-p_w)^N - Np_w(1-p_w)^{N-1} \\ &= 1 - (1+(N-1)p_w)(1-p_w)^{N-1}, \end{aligned}$$

which uses the fact that the sum over all t evaluates to 1. Thus

$$\mathrm{Ex}[V] = \sum_{w\in A^*} \mathrm{Ex}[V_w] \tag{3.1}$$

$$= \sum_{w\in A^*} \bigl(1 - (1+(N-1)p_w)(1-p_w)^{N-1}\bigr). \tag{3.2}$$

Similarly for K we get that

$$\begin{aligned} \mathrm{Ex}[K_w] &= \sum_{2\le t\le N} b(p_w, N, t)\cdot t \\ &= Np_w - Np_w(1-p_w)^{N-1} \\ &= Np_w\bigl(1 - (1-p_w)^{N-1}\bigr), \end{aligned}$$

which uses the fact that the sum over all t evaluates to Np_w. Thus

$$\mathrm{Ex}[K] = \sum_{w\in A^*} \mathrm{Ex}[K_w] \tag{3.3}$$

$$= N\cdot \sum_{w\in A^*} p_w\bigl(1 - (1-p_w)^{N-1}\bigr). \tag{3.4}$$

Finally we consider Z. We get

$$\begin{aligned} \mathrm{Ex}[Z_w] &= \sum_{2\le t\le N} b(p_w, N, t)\cdot t\log t \\ &= \sum_{2\le t\le N} \binom{N}{t} p_w^t (1-p_w)^{N-t}\cdot t\log t \\ &= Np_w \sum_{2\le t\le N} \binom{N-1}{t-1} p_w^{t-1}(1-p_w)^{(N-1)-(t-1)}\log t \\ &= Np_w \sum_{1\le \tau\le N-1} \binom{N-1}{\tau} p_w^{\tau}(1-p_w)^{(N-1)-\tau}\log(\tau+1) \\ &= Np_w\cdot \mathrm{Ex}[\log(1+X)], \end{aligned}$$

where X is the sum of $N-1$ Bernoulli Variables with bias p_w. We thus have $\mathrm{Ex}[X] = (N-1)p_w \le Np_w$, and since $\log(1+x)$ is a concave function we can apply Jensen's inequality to get

$$\mathrm{Ex}[\log(1+X)] \le \log(1+\mathrm{Ex}[X])$$

and therefore

$$\mathrm{Ex}[Z_w] \le Np_w\log(1+Np_w).$$

So we can conclude that

$$\mathrm{Ex}[Z] = \sum_{w\in A^*} \mathrm{Ex}[Z_w] \le \sum_{w\in A^*} Np_w\log(1+Np_w). \tag{3.5}$$

The expressions (3.1) and (3.3) are of course well known in the trie literature. They are usually derived using recurrence relations and generating functions and various restrictions on the random process are stated (e.g. see [4, Theorem 2]). Our derivation clearly shows that there are really no particular requirements on the random process, save that the p_w's exist.

Of course evaluating the expressions (3.1), (3.3), and (3.5) requires restrictions on the p_w's and the random process. Here we will just briefly deal with the simplest case where each symbol of the randomly generated string is with equal probability one of the a letters in the alphabet A, of course independent of previous choices. Thus we have $p_w = 1/a^{|w|}$ for all $w \in A^*$. By grouping strings according to their lengths we get

$$\mathrm{Ex}[V] = \sum_{k\ge 0} a^k\bigl(1 - (1+(N-1)/a^k)(1-1/a^k)^{N-1}\bigr) \tag{3.6}$$

$$\mathrm{Ex}[K] = N\cdot \sum_{k\ge 0}\bigl(1 - (1-1/a^k)^{N-1}\bigr) \tag{3.7}$$

$$\mathrm{Ex}[Z] \le N\cdot \sum_{k\ge 0}\log(1+N/a^k) \tag{3.8}$$

There is a standard, simple way of evaluating these sums: Split the sum at $k = \lambda = \log_a N$ and use different bounds for summands with small and large k.

In the case of $\mathrm{Ex}[V]$ we use the bound of 1 for small k, and for large k the bound of $(N-1)^2/a^k$ (obtained via the Bernoulli inequality $(1-x)^M \geq 1 - Mx$). For N a power of a this results in

$$\mathrm{Ex}[V] \leq \frac{a+1}{a-1} N\,.$$

Similarly in the case of $\mathrm{Ex}[K]$ use the bound of 1 for small k and the bound of $(N-1)/a^k$ for large k (again obtained via the Bernoulli ineuqality). For N a power of a this results in

$$\mathrm{Ex}[K] \leq (N \cdot \log N)/(\log a) + \frac{a}{a-1} N\,.$$

In the case of $\mathrm{Ex}[Z]$ for small k use the bound $\log(1+x) \leq 1/x + \log x$ (consider the area underneath the curve $y = 1/x$), and for large k use the bound $\log(1+x) \leq x$. This yields

$$\begin{aligned}\mathrm{Ex}[Z] \leq\; & N \cdot \sum_{0 \leq k < \lambda} a^k/N \\ & + N \cdot \sum_{0 \leq k < \lambda} \log(N/a^k) \\ & + N \cdot \sum_{k \geq \lambda} N/a^k\,.\end{aligned}$$

For N a power of a the first sum evaluates to $(N-1)/(a-1)$, and the third sum to $\frac{a}{a-1}N$. The second sum asymptotically dominates and is less than $(1/2)(N \log^2 N)/\log a$. Thus we have

$$\mathrm{Ex}[Z] \leq (N \log^2 N)/(2 \cdot \log a) + \frac{a+1}{a-1} N\,. \tag{3.9}$$

The constants of the leading terms in our bounds for $\mathrm{Ex}[K]$ and $\mathrm{Ex}[Z]$ turn out to be exact. For $\mathrm{Ex}[V]$ our constant is too large. You can obtain a better bound by approximating $\sum_{k \geq 0} f(k)$ by $\int_0^\infty f(k)dk$, where in our case we have

$$f(k) = a^k\big(1 - (1 + (N-1)/a^k)(1 - 1/a^k)^{N-1}\big)\,.$$

Somewhat surprisingly $f(k)$ integrates nicely with respect to k (substitute $u = 1/a^k$ and hence $dk = -u\,du/(\log a)$) and you obtain

$$\int_0^\infty f(k)dk = (N-1)/(\log a)\,.$$

However the error incurred by this approximation is difficult to estimate using the standard Euler-Maclaurin formula. But the function $f(k)$ is positive and bimodal over the positive reals (first increasing, then decreasing). For such a bimodal function it is easy to see that the following holds:

$$\sum_{k \geq 0} f(k) \leq \int_0^\infty f(k)\,dk + \max_{k \geq 0} f(k)\,.$$

In our case we have $f(k) \leq N/2$ since in a trie for N strings you can have at most $N/2$ nodes per level (a node must correspond to the prefix of at least 2 strings). Thus we get

$$\mathrm{Ex}[V] \leq N/(\log a) + N/2\,.$$

For small a this $N/2$ term can be shown to be superfluous, however for large a the $N/(\log a)$ term by itself can be much too small, since for $N - 1 = a^k$ you get $f(k) \approx N(1-2/e) > N/4$ and hence $\sum_{k \geq 0} f(k) > N/4$.

From Theorem 1.1 and inequality 3.9 we now get what was initially proven by Fill and Janson [5] for alphabet size $a = 2$:

THEOREM 3.1. *Let S be a set of n sufficiently long strings with each symbol chosen independently and uniformly at random from an alphabet of size a.*

If quicksort is used to sort S using the standard string comparison procedure the expected number of symbol comparisons performed will be $(n \log^2 n)/\log a + O(n)$.

By the remark after Theorem 1.1 we can claim the $1/\log a$ constant of the leading term to be exact.

4 Mergesort and a variant

Our aim is to prove Theorem 1.2. By Lemma 2.1 it suffices to show that mergesort is $R(t)$-faithful, with $R(t) = t \log_2 t$. This means that when sorting a set X of keys $x_1 < x_2 < \cdots < x_n$ for any $0 < t \leq n$ and for any "range" $X_{(i,i+t]} = \{x_{i+1}, \ldots, x_{i+t}\}$ of t order-consecutive elements of X the expected number of comparisons among elements of $X_{(i,i+t]}$ is at most $R(t)$.

If m of the keys in range $X_{(i,i+t]}$ participate in one of the recursive merges of the sort algorithm then during that merge there will be at most $m-1$ comparisons between keys in this range. To get all comparisons among the keys in our range we therefore have to determine for each of the recursive merges the number of keys of our range that participate in it.

For this purpose we need to specify the merge pattern that is used by mergesort. For the sake of exposition and ease of analysis we will use a slightly

unusual version of randomized mergesort: To sort X, flip for each element of X a fair coin and depending on the outcome put it into set X_0 or set X_1. These two sets are mergesorted recursively, and then merged. Thus we can associate with each $x \in X$ a random bitstring $b[x]$, and each merge is named by some bitstring w and involves exactly all $x \in X$ for which w is a prefix of $b[x]$. In other words, the merge pattern is really given by the trie for the set $\{\, b[x] \,|\, x \in X \,\}$.

In order to bound the number of comparisons made among keys in a range $\overline{X} = X_{(i,i+t]}$ of size t we proceed as follows: Consider the trie T for the set $b[\overline{X}] = \{\, b[x] \,|\, x \in \overline{X} \,\}$. Each node v in T corresponds to a merge operation, in which all $x \in \overline{X}$ participate for which w is a prefix of $b[x]$. Their number is the thickness $\vartheta(w)$ and thus we know that the number of comparisons among keys in $\overline{X}$ in that merge is $\vartheta(w) - 1$. Thus the total number of comparisons among keys in $\overline{X}$ is given by

$$\sum_{w \in T} \big(\vartheta(w) - 1\big)\,. \tag{4.10}$$

Since $b[\overline{X}]$ consists of t random binary strings we can apply the result stated in the previous section, and get that the expected value of this sum is $\mathrm{Ex}[K] - \mathrm{Ex}[V]$, which is $(t \log t - t)/\log 2 \le t \log_2 t = R(t)$. Thus mergesort is $R(t)$-faithful and we have proved Theorem 1.2.

Our version of randomized mergesort is a bit unusual. It is possible to adjust the analysis to the more customary merge pattern that is given by the trie formed by the bit strings $b'[x]$ given by the binary representation of the position of x in the input. If the input is randomly permuted than these are again random strings, although according to a slightly different random model. In this model the expected value of expression (4.10) can then be bounded by $R'(t) = t \log_2 t + 3t$.

Let us now turn to our "lazy mergesort" variant and the proof of Theorem 1.5. We want to sort a set S of n strings. As in the proof of Theorem 1.4 we will classify symbol comparisons by their outcome as wins or ties and analyze them separately. The number of wins is of course again exactly the same as the number of key comparisons performed by mergesort giving the $M(n)$ bound.

Let w be some prefix of positive length k and let S_w be the strings in S with prefix w. We have $|S_w| = \vartheta(w)$, considering the trie $T(S)$. Also note that the string in S_w will form a contiguous subrange of the sorted output. Here is a simple observation: After string $s \in S_w$ has been compared to some other string $s' \in S_w$ and has been found to be smaller in some merge (i.e. s succeeds s' in a sorted sublist generated by the merge) in all subsequent comparisons between s and other strings of S_w there will be no symbol comparison between the k-th symbols of the strings. Why? Because from then on s will always share a common prefix of length k with its predecessor, whichever list it happens to be in, and this fact will be recorded by $\ell[s] \ge k$, which makes further comparisons in the k-th symbol between s and some other string in S_w unnecessary.

This means each string in S_w can "lose" at most 1 tie in the k-th position to other strings in S_w and since the smallest string in S_w does not ever "lose" such a tie we get that the number of comparisons in the k-th symbol between strings in S_w is at most $|S_w| - 1 = \vartheta(w) - 1$. This completes the proof of Theorem 1.5.

We have implemented mergesort and this variant in order to check our results on the number of symbol comparisons. Our experiments confirm the results of Theorems 1.2 and 1.5. The experiments also show that using the $\ell[]$-values it is possible to reduce the number of wins substantially. However, we have not yet completed a sound analysis of this phenomenon.

5 Lower Bounds

Let S be a set of n distinct keys $s_1 < s_2 < \cdots < s_n$ and let A be a key-comparison based sorting algrithm. The set S is presented as input to algorithm A in order given by some permutation π. Let $C_A(\pi, S)$ denote the number of comparisons performed by A when S is input under permutation π. Let R be some order contiguous subrange of S, i.e. $R = R_{[i,j]} = \{s_i, s_{i+1}, \ldots, s_j\}$ for some $1 \le i \le j \le n$. Define $C_A(\pi, R)$ to be the number of comparisons between keys in R performed by algorithm A when faced with input S permuted by π. If the keys in S are strings inducing trie $T(S)$ and S is presented to A under permutation π and A realizes key comparisons by the standard string comparison procedure, then the total number of symbol comparisons performed will be

$$\sum_{w \in T(S)} C_A(\pi, S_w)$$

where S_w is again the contiguous subrange of strings in S with common prefix w. For every π we obviously have $C_A(\pi, S_w) \ge |S_w| - 1 = \vartheta(w) - 1$, since every one except for the largest string in S_w must lose at least one comparison to some other string in S_w. Thus to prove the first part of Theorem 1.3 it suffices to

show that there is a permutation π with $C_A(\pi, S_w) \geq L(\vartheta(w)) - \lg n$ for each S_w.

Consider some subrange $S_w = R_{[i,j]}$ and consider the decision tree D induced by algorithm A. Consider the set of permutations U that map all indices outside $[i, j]$ to an arbitrary sequence of fixed values. Thus $|U| = |S_w|! = \vartheta(w)!$. Each leaf of D corresponds to a permutation π. Take the leaves that correspond to permutations in U along with their rootpaths. They induce a subtree of D. Contracting all chains in this tree results in a tree D_U that represents a valid sorting algorithm for S_w. Since D_U is a binary tree with $\vartheta(w)!$ leaves, for any $k > 0$ there can be at most a $(1/2^k)$-fraction of these leaves that have distance less than $L(\vartheta(w)) - k = \lceil \log_2 \vartheta(w)! \rceil - k$ from the root. In other words, at most a $(1/2^k)$-fraction of the permutations π in U are k-bad for S_w in the sense that $C_A(\pi, S_w) < L(\vartheta(w)) - k$. Since this is true for any which way the permutation values outside $[i, j]$ were fixed we get that for any $k > 0$ the fractions of all permutations that are k-bad for S_w is at most $1/2^k$.

Now it suffices to observe that, although the trie $T(S)$ can have arbitrarily many nodes w, there are only at most $n - 1$ different sets S_w. (If a trie node w has only a single child v then $S_w = S_v$.) Thus choosing $k \geq \lg n$ ensures that there is some permutation π that is not k-bad for any S_w, i.e. for this permutation we have the desired $C_A(\pi, S_w) \geq L(\vartheta(w)) - \lg n$ for each S_w. This completes the proof of the first part of Theorem 1.3.

We only give a short sketch of the proof of the second part: Model a randomized sorting algorithm as a family $\mathcal{A}$ of deterministic algorithms from which one is then chosen at random. Fix a set S_w and choose $k = 2\lg n$. According to the arguments of the previous paragraph for any algorithm $A \in \mathcal{A}$ there is only a $1/n^2$ fraction of the permutations that are k-bad for S_w. From this conclude that there is at most a $1/n$ fraction of permutations for which more than a $1/n$ fraction of the algorithms is k-bad. For the other permutations π the average of $C_A(S_w, \pi)$ taken over all $A \in \mathcal{A}$ is at least $L(\vartheta(w)) - 3\lg n$ (with the additional negative $\lg n$ term deriving from the algorithms with π being k-bad for S_w). Thus for at most a $1/n$ fraction of the permutations π we have $\mathrm{Ex}_A[C_A(S_w, \pi)] < L(\vartheta(w)) - 3\lg n$. Considering that there are only $n - 1$ different S_w's we can again conclude that there is a permutation π for which we have $\mathrm{Ex}_A[C_A(S_w, \pi)] \geq L(\vartheta(w)) - 3\lg n$ for all S_w.

6 Acknowledgements

Some of this work was motivated by a Dagstuhl workshop on *Adaptive, Output Sensitive, Online and Parameterized Algorithms*. I would like to thank Sascha Parduhn for some implementation work.

References

[1] J.L. Bentley and R. Sedgewick Fast Algorithms for Sorting and Searching Strings. *Proc. 8th ACM-SIAM SODA* (1997) 360–369.

[2] K.L. Clarkson and P.W. Shor Applications of Random Sampling in Computational Geometry, II. *Discrete & Computational Geometry*, 4 (1989) 387–421.

[3] K.L. Clarkson, K. Mehlhorn, and R. Seidel Four Results on Randomized Incremental Constructions. *Comput. Geom. Theory and Applications*, 3 (1993) 185–212.

[4] J. Clément, P. Flajolet, and B. Vallée Dynamical Sources in Information Theory: A General Analysis of Trie Structures. *Algorithmica* 29(1) (2001) 307–369.

[5] J.A. Fill and S. Janson The Number of Bit Comparisons Used by Quicksort: an Average-Case Analysis. *Proc. 15th ACM-SIAM SODA* (2004) 300–307.

[6] P. Flajolet, X. Gourdon, and P. Dumas Mellin Transforms and Asymptotics: Harmonic Sums. *Theoretical Computer Science* 144(1–2) (1995) 3–58.

[7] D.E. Knuth **The Art of Computer Programming. Volume 3: Sorting and Searching.** Addison-Wesley (1998).

[8] H.M. Mahmoud **Evolutions of Random Search Trees.** Wiley (1992).

[9] K. Mehlhorn **Data Structures and Algorithms 1: Sorting and Searching.** Springer (1984).

[10] W. Ng and K. Kakehi Merging String Sequences by Longest Common Prefixes. *IPSJ Digital Courier* 4 (Feb. 2008) 69–78.

[11] R. Sedgewick and P. Flajolet **An Introduction to the Analysis of Algorithms.** Addison-Wesley (1996).

[12] W. Szpankowski **Average Case Analysis of Algorithms on Sequences.** Wiley (2001).

[13] M. Thorup Private Communication.

[14] B. Vallée, J. Clément, J.A. Fill, and P. Flajolet The Number of Symbol Comparisons in QuickSort and QuickSelect. *Proc. ICALP (1)*(2009) 750–763.