On an integer alphabet, when $m \leq w$:

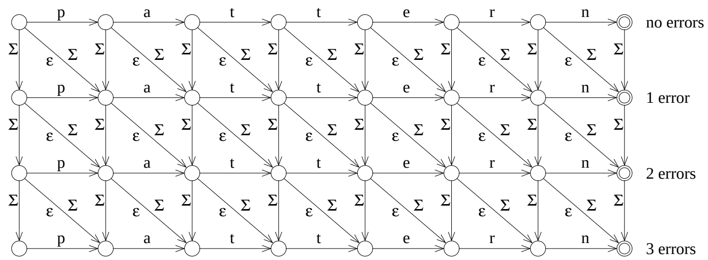
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

When $m > w$, we can store each bit vector in $\lceil m/w \rceil$ machine words:

- The worst case search time is $\mathcal{O}(n\lceil m/w \rceil)$.
- Using Ukkonen's cut-off heuristic, it is possible reduce the average case search time to $\mathcal{O}(n\lceil k/w \rceil)$.

There are also algorithms based on bitparallel simulation of a nondeterministic automaton.

Example 2.22: $P = \texttt{pattern}$, $k = 3$



- The algorithm of Wu and Manber uses a bit vector for each row. It can be seen as an extension of Shift-And. The search time complexity is $\mathcal{O}(kn\lceil m/w \rceil)$.
- The algorithm of Baeza-Yates and Navarro uses a bit vector for each diagonal, packed into one long bitvector. The search time complexity is $\mathcal{O}(n\lceil km/w \rceil)$.

Baeza-Yates–Perleberg Filtering Algorithm

A *filtering algorithm* for approximate strings matching searches the text for factors having some *property* that satisfies the following conditions:

1. Every approximate occurrence of the pattern has this property.
2. Strings having this property are reasonably rare.
3. Text factors having this property can be found quickly.

Each text factor with the property is a *potential occurrence*, and it is *verified* for whether it is an actual approximate occurrence.

Filtering algorithms can achieve linear or even sublinear *average case time* complexity.

The following lemma shows the property used by the *Baeza-Yates–Perleberg algorithm* and proves that it satisfies the first condition.

Lemma 2.23: Let $P_1 P_2 \ldots P_{k+1} = P$ be a partitioning of the pattern P into $k + 1$ nonempty factors. Any string S with $ed(P, S) \leq k$ contains P_i as a factor for some $i \in [1..k + 1]$.

Proof. Each single symbol edit operation can change at most one of the pattern factors P_i. Thus any set of at most k edit operations leaves at least one of the factors untouched. $\square$

The algorithm has two phases:

Filtration: Search the text T for exact occurrences of the pattern factors P_i. This is done in $\mathcal{O}(n)$ time using the Aho–Corasick algorithm for multiple exact string matching, which we will see later during this course.

Verification: An area of length $\mathcal{O}(m)$ surrounding each potential occurrence found in the filtration phase is searched using the standard dynamic programming algorithm in $\mathcal{O}(m^2)$ time.

The worst case time complexity is $\mathcal{O}(m^2 n)$, which can be reduced to $\mathcal{O}(mn)$ by combining any overlapping areas to be searched.

Let us analyze the *average case* time complexity of the verification phase.

- The best pattern partitioning is as even as possible. Then each pattern factor has length at least $r = \lfloor m/(k+1) \rfloor$.
- The expected number of exact occurrences of a random string of length r in a random text of length n is at most n/σ^r.
- The expected total verification time is at most
$$\mathcal{O}\left(\frac{m^2(k+1)n}{\sigma^r}\right) \leq \mathcal{O}\left(\frac{m^3 n}{\sigma^r}\right) .$$
This is $\mathcal{O}(n)$ if $r \geq 3 \log_\sigma m$.
- The condition $r \geq 3 \log_\sigma m$ is satisfied when $(k+1) \leq m/(3 \log_\sigma m + 1)$.

Theorem 2.24: The average case time complexity of the Baeza-Yates–Perleberg algorithm is $\mathcal{O}(n)$ when $k \leq m/(3 \log_\sigma m + 1) - 1$.

Many variations of the algorithm have been suggested:

- The filtration can be done with a *different multiple exact string matching algorithm*:
  - The first algorithm of this type by Wu and Manber used an extension of the Shift-And algorithm.
  - An extension of BDM achieves $\mathcal{O}(nk(\log_\sigma m)/m)$ average case search time. This is sublinear for small enough k.
  - An extension of the Horspool algorithm is very fast in practice for small k and large σ.
- Using a technique called *hierarchical verification*, the average verification time for a single potential occurrence can be reduced to $\mathcal{O}((m/k)^2)$.

A filtering algorithm by Chang and Marr has average case time complexity $\mathcal{O}(n(k + \log_\sigma m)/m)$, which is optimal.

3. Sorting and Searching Sets of Strings

Sorting algorithms and *search trees* are among the most fundamental algorithms and data structures for sets of objects. They are also closely connected:

- A sorted array is an *implicit* search tree through *binary search*.
- A set can be sorted by inserting the elements in a search tree and then traversing the tree in order.

Another connecting feature is that they are usually based on *order comparisons* between elements, and their time complexity analysis counts the *number of comparisons* needed.

These algorithms and data structures work when the elements of the set are *strings* (pointers to strings, to be precise). However, comparisons are no more constant time operations in general, and the number of comparisons does not tell the whole truth about the time complexity.

In this part, we will see that algorithms and data structures *designed specifically for strings* are often faster.

Sorting Strings

Let us first define an order for strings formally.

Definition 3.1: Let $A[0..m)$ and $B[0..n)$ be two strings on an ordered alphabet Σ. We say that A is *lexicographically* smaller than B, denoted $A < B$, if and only if either

- $m < n$ and $A = B[0..m)$ (i.e., A is a proper prefix of B) or
- $A[0..i) = B[0..i)$ and $A[i] < B[i]$ for some $i \in [0..\min\{m,n\})$.

Determining the order of A and B needs $\Theta(\min\{m,n\})$ *symbol comparisons* in the worst case.

On the other hand, the *expected* number of symbol comparisons for two random strings is $\mathcal{O}(1)$.

$\Omega(n \log n)$ is a well known *lower bound* for the number of comparisons needed for sorting a set of n objects by any comparison based algorithm. This lower bound holds both in the worst case and in the average case.

There are many algorithms that match the lower bound, i.e., sort using $\mathcal{O}(n \log n)$ comparisons (worst or average case). Examples include quicksort, heapsort and mergesort.

If we use one of these algorithms for sorting a set of n strings, it is clear that the number of *symbol comparisons* can be more than $\mathcal{O}(n \log n)$ in the *worst case*.

What about the *average case* when sorting a set of *random strings*?

The following theorem shows that we cannot achieve $\mathcal{O}(n \log n)$ symbol comparisons in the average case (when $\sigma = n^{o(1)}$).

Theorem 3.2: Let $\mathcal{A}$ be an algorithm that sorts a set of objects using only comparisons between the objects. Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over an ordered alphabet of Σ. Sorting $\mathcal{R}$ using $\mathcal{A}$ requires $\Omega(n \log n \log_\sigma n)$ symbol comparisons.

- If σ is considered to be a constant, the lower bound is $\Omega(n(\log n)^2)$.
- An intuitive explanation for this result is that the comparisons made by a sorting algorithm are *not random*.

Proof of Theorem 3.2. Let $k = \lfloor (\log_\sigma n)/2 \rfloor$. For any string $\alpha \in \Sigma^k$, let $\mathcal{R}_\alpha$ be the set of strings in $\mathcal{R}$ having α as a prefix. Let $n_\alpha = |\mathcal{R}_\alpha|$.

Let us analyze the number of symbol comparisons when comparing strings in $\mathcal{R}_\alpha$ *against each other*.

- Each string comparison needs at least k symbol comparisons.
- No comparison between a string in $\mathcal{R}_\alpha$ and a string outside $\mathcal{R}_\alpha$ gives any information about the relative order of the strings in $\mathcal{R}_\alpha$.
- Thus $\mathcal{A}$ needs to do $\Omega(n_\alpha \log n_\alpha)$ string comparisons and $\Omega(kn_\alpha \log n_\alpha)$ symbol comparisons to determine the relative order of the strings in $\mathcal{R}_\alpha$.

Thus the total number of symbol comparisons is $\Omega\left(\sum_{\alpha \in \Sigma^k} kn_\alpha \log n_\alpha\right)$ and

$$\sum_{\alpha \in \Sigma^k} kn_\alpha \log n_\alpha \geq k(n - \sqrt{n}) \log \frac{n - \sqrt{n}}{\sigma^k} \geq k(n - \sqrt{n}) \log(\sqrt{n} - 1)$$
$$= \Omega(kn \log n) = \Omega(n \log n \log_\sigma n) \ .$$

The first step uses the facts that $\sum_{\alpha \in \Sigma^k} n_\alpha > n - \sqrt{n}$ and that $\sum_{\alpha \in \Sigma^k} n_\alpha \log n_\alpha > (n - \sqrt{n}) \log((n - \sqrt{n})/\sigma^r)$ (see exercises). $\square$

The preceding lower bound does not hold for algorithms *specialized for sorting strings*. To derive a lower bound for any algorithm based on *symbol comparisons*, we need the following concept.

Definition 3.3: Let S be a string and $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ a set of strings. The *distinguishing prefix* of S in $\mathcal{R}$ is the shortest prefix of S that separates it from the (other) members $\mathcal{R}$. Let $dp_{\mathcal{R}}(S)$ denote the length of the distinguishing prefix, and let $DP(\mathcal{R}) = \sum_{T \in \mathcal{R}} dp_{\mathcal{R}}(T)$ be the total length of distuinguishing prefixes in $\mathcal{R}$.

Example 3.4: Distuingishing prefixes:

```
a akkoselliseen
järjesty kseen
järjestä
m erkkijonot
n ämä
```


Theorem 3.5: Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings. Sorting $\mathcal{R}$ into the lexicographical order by any algorithm based on symbol comparisons requires $\Omega(DP(\mathcal{R}) + n \log n)$ symbol comparisons.

Proof. The algorithm must examine every symbol in the duistinguishing prefixes. This gives a lower bound $\Omega(DP(\mathcal{R}))$.

On the other hand, the general sorting lower bound $\Omega(n \log n)$ must hold here too.

The result follows from combining the two lower bounds. □

- Note that for a random set of strings $DP(\mathcal{R}) = \mathcal{O}(n \log_\sigma n)$ on average. The lower bound then becomes $\Omega(n \log n)$.

We will next see that there are algorithms that match this lower bound. Such algorithms can sort a random set of strings in $\mathcal{O}(n \log n)$ time.

String Quicksort (Multikey Quicksort)

Quicksort is one of the fastest general purpose sorting algorithms in practice.

Here is a variant of quicksort that partitions the input into three parts instead of the usual two parts.

Algorithm 3.6: TernaryQuicksort(R)

Input: (Multi)set R in arbitrary order.
Output: R in increasing order.
(1) *if* $|R| \leq 1$ *then return* R
(2) select a pivot $x \in R$
(3) $R_< \leftarrow \{s \in R \mid s < x\}$
(4) $R_= \leftarrow \{s \in R \mid s = x\}$
(5) $R_> \leftarrow \{s \in R \mid s > x\}$
(6) $R_< \leftarrow$ TernaryQuicksort($R_<$)
(7) $R_> \leftarrow$ TernaryQuicksort($R_>$)
(8) *return* $R_< \cdot R_= \cdot R_>$

In the normal, *binary* quicksort, we would have two subsets $R_\leq$ and $R_\geq$, both of which may contain elements that are equal to the pivot.

- Binary quicksort is slightly faster in practice for sorting sets.
- Ternary quicksort can be faster for sorting multisets with many duplicate keys (exercise).

The time complexity of both the binary and the ternary quicksort depends on the *selection of the pivot*:

- The optimal choice is the *median* of R, which can be computed in linear worst case time. Then the time complexity is $\mathcal{O}(n \log n)$ in the worst case.
- If we choose the pivot *randomly*, the expected time complexity is $\mathcal{O}(n \log n)$.
- A practical choice is the median of a few elements.

In the following, we assume an optimal pivot selection.

String quicksort is similar to ternary quicksort, but it partitions using a single character position.

Algorithm 3.7: StringQuicksort$(\mathcal{R}, \ell)$

Input: Set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: R in increasing lexicographical order.

(1) *if* $|\mathcal{R}| \leq 1$ *then return* $\mathcal{R}$
(2) $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
(3) select pivot $X \in \mathcal{R}$
(4) $\mathcal{R}_< \leftarrow \{S \in \mathcal{R} \mid S[\ell] < X[\ell]\}$
(5) $\mathcal{R}_= \leftarrow \{S \in \mathcal{R} \mid S[\ell] = X[\ell]\}$
(6) $\mathcal{R}_> \leftarrow \{S \in \mathcal{R} \mid S[\ell] > X[\ell]\}$
(7) $\mathcal{R}_< \leftarrow$ StringQuicksort$(\mathcal{R}_<, \ell)$
(8) $\mathcal{R}_= \leftarrow$ StringQuicksort$(\mathcal{R}_=, \ell + 1)$
(9) $\mathcal{R}_> \leftarrow$ StringQuicksort$(\mathcal{R}_>, \ell)$
(10) return $\mathcal{R}_\perp \cdot \mathcal{R}_< \cdot \mathcal{R}_= \cdot \mathcal{R}_>$

In the initial call, $\ell = 0$.

Example 3.8: A possible partitioning, when $\ell = 2$.

al	p	habet
al	i	gnment
al	l	ocate
al	g	orithm
al	t	ernative
al	i	as
al	t	ernate
al	l	

$\Longrightarrow$

al	i	gnment
al	g	orithm
al	i	as
al	l	ocate
al	l	
al	p	habet
al	t	ernative
al	t	ernate

Theorem 3.9: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(DP(\mathcal{R}) + n \log n)$ time.

- Thus string quicksort is an optimal symbol comparison based algorithm.
- String quicksort is also fast in practice.

Proof of Theorem 3.9. The time complexity is dominated by the symbol comparisons on lines (4)–(6). We charge the cost of each comparison either on a single symbol or on a string depending on the result of the comparison:

$S[\ell] = X[\ell]$**:** Charge the comparison on the symbol $S[\ell]$.

- Now the string S is placed in the set $\mathcal{R}_=$. The recursive call on $\mathcal{R}_=$ increases the common prefix length to $\ell + 1$. Thus $S[\ell]$ cannot be involved in any future comparison and the total charge on $S[\ell]$ is 1.
- The algorithm never accesses symbols outside the distinguishing prefixes. Thus the total number of symbol comparisons resulting equality is at most $\mathcal{O}(DP(\mathcal{R}))$.

$S[\ell] \neq X[\ell]$**:** Charge the comparison on the string S.

- Now the string S is placed in the set $\mathcal{R}_<$ or $\mathcal{R}_>$. Assuming an optimal choice of the pivot X, the size of either set is at most $|\mathcal{R}|/2$.
- Every comparison charged on S halves the size of the set containing S, and hence the total charge accumulated by S is at most $\log n$.
- Thus the total number of symbol comparisons resulting inequality is at most $\mathcal{O}(n \log n)$. □