1 Compressed Suffix Arrays

We will show in this section that $O(n \log \sigma)$ bits suffice also for representing $A$. The price of this compressed suffix array is that the time for retrieving an entry from $A$ is not constant any more, but rises from $O(1)$ to $O(\log \varepsilon n)$, for some arbitrarily small constant $0 < \epsilon \leq 1$.

1.1 Recommended Reading


1.2 The $\psi$-Function

The most important component of the compressed suffix array (abbreviated as CSA henceforth) is a function $\psi$ that allows us to “jump one character forward” in the suffix array.

**Definition 1.** Define $\psi: [1,n] \to [1,n]$ such that $\psi(i) = j \iff A[j] = A[i] + 1$, where position $n + 1$ is interpreted as the first position in $T$ (read text circularly!).

**Example 1.**

$$
\begin{align*}
&1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \\
&T = C \ A \ C \ A \ T \ A \ C \ A \ T \ A \ T \ A \ C \ $ \\
&A = 16 \ 4 \ 14 \ 2 \ 7 \ 12 \ 5 \ 9 \ 15 \ 3 \ 1 \ 8 \ 13 \ 6 \ 11 \ 10 \\
&\psi = 11 \ 7 \ 9 \ 10 \ 12 \ 13 \ 14 \ 16 \ 1 \ 2 \ 4 \ 8 \ 3 \ 5 \ 6 \ 15
\end{align*}
$$

Note the similarity of the $\psi$-function to suffix links in suffix trees: both “cut off” the first character of the corresponding substring.

Function $\psi$ is increasing in areas where the corresponding suffixes start with the same character. For instance, in Ex. 1 we have that all suffixes from $A[2, 9]$ start with letter A; and indeed, $\psi[2, 9] = [7, 9, 10, 12, 13, 14, 16]$ is increasing. This is summarized in the following lemma.

**Lemma 1.** If $i < j$ and $T_{A[i]} = T_{A[j]}$, then $\psi(i) < \psi(j)$.

This lemma will be used in Sect. 1.6 to store $\psi$ in a space-efficient form.
1.3 The Idea of the Compressed Suffix Array

We now present the general approach to store $A$ in a space-efficient form. Instead of storing every entry in $A$, in a new bit-vector $B_0[1,n]$ we mark the positions in $A$ where the corresponding entry in $A$ is even:

$$B_0[i] = 1 \iff A[i] \equiv 0 \pmod{2}.$$  

Bit-vector $B_0$ is prepared for $O(1)$ \textsc{rank}-queries. We further store the $\psi$-values at positions $i$ with $B_0[i] = 0$ in a new array $\psi_0[1, \lceil \frac{n}{2} \rceil]$. Finally, we store the even values of $A$ in a new array $A_1[1, \lfloor \frac{n}{2} \rfloor]$, and divide all values in $A_1$ by 2.

Example 2.

\begin{align*}
T &= \text{C A C A A T A C A T T A T A C }$
A &= 16 4 14 2 7 12 5 9 15 3 1 8 13 6 11 10
B_0 &= 1 1 1 1 0 1 0 0 0 0 1 0 1 0 1
\psi_0 &= 12 14 16 1 2 4 3 6 8 2 7 1 6 4 3 5
A_1 &= 8 2 7 1 6 4 3 5
\end{align*}

Now, the three arrays, $B_0$, $\psi_0$ and $A_1$, completely substitute $A$: to retrieve value $A[i]$, we first check if $B_0[i] = 1$. If so, we know that $A[i]/2$ is stored in $A_1$, and that the exact position in $A_1$ is given by the number of 1-bits in $B_0$ up to position $i$. Hence, $A[i] = 2A_1[\text{rank}_1(B_0, i)]$.

If, on the other hand, $B_0[i] = 0$, we follow $\psi(i)$ in order to get to the position of the $(A[i]+1)$st suffix, which must be even (and is hence stored in $A_1$). The value $\psi(i)$ is stored in $\psi_0$, and its position therein is equal to the number of 0-bits in $B_0$ up to position $i$. Hence, $A[i] = A[\psi_0(\text{rank}_0(B_0, i))] - 1$, which can be calculated be the mechanism of the previous paragraph.

As we shall see later, $\psi_0$ can be stored very efficiently (basically using $O(n \log \sigma)$ bits). Hence, we have almost halved the space with this approach (from $n \log n$ bits for $A$ to $\frac{n}{2} \log \frac{n}{2}$ for $A_1$).

1.4 Hierarchical Decomposition

We can use the idea from the previous section recursively in order to gain more space: instead of representing $A_1$ plainly, we replace it with bit-vector $B_1$, array $\psi_1$ and $A_2$. Array $A_2$ can in turn be replaced by $B_2$, $\psi_2$, and $A_3$, and so on. In general, array $A_k[1,n_k]$, with $n_k = \frac{n}{2^k}$, implicitly represents $T$’s suffixes that are a multiple of $2^k$, in the order as they appear in the original array $A_0 := A$. 

Example 3.

\[ T = C A C A A T A C A T A C \]
\[ A = 16 4 14 2 7 12 5 9 15 3 1 8 13 6 11 10 \]
\[ \psi_0 = \begin{bmatrix} 12 & 14 & 16 & 1 & 2 & 4 & 3 & 6 \end{bmatrix} \]
\[ B_0 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \]
\[ A_1 = 8 2 7 1 6 4 3 5 \]
\[ \psi_1 = \begin{bmatrix} 1 & 2 & 6 & 5 \end{bmatrix} \]
\[ B_1 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix} \]
\[ A_2 = 4 1 3 2 \]
\[ \vdots \quad \text{etc.} \]

\[ A_k \] can be seen as a suffix array of a new string \( T^k \), where the \( i \)'th character of \( T^k \) is the concatenation of \( 2^k \) characters \( T_{i2^k \ldots (i+1)2^k-1} \) (we assume that \( T \) is padded with sufficiently enough \$\)-characters). This means that the alphabet for \( T^k \) is \( \Sigma^{2^k} \), i.e., all \( 2^k \)-tuples from \( \Sigma \).

Example 4. \( A_2 = [4, 1, 3, 2] \) can be regarded as the suffix array of

\[ T^2 = (AATA) (CATT) (ATAC) (\\ldots) \]

This way, on level \( k \) we only store \( B_k \) and \( \psi_k \). Only on the last level \( h \) we store \( A_h \). We choose \( h = \lceil \log \log \frac{n}{\log n} \rceil \) such that the space for storing \( A_h \) is

\[ O(n_h \log n_h) = O(n_h \log n) = O \left( \frac{n \log \sigma}{\log \frac{n}{\log n}} \log n \right) = O(n \log \sigma) \text{ bits}. \]

However, storing \( B_k \) and \( \psi_k \) on all \( h \) levels would take too much space. Instead, we use only a constant number of \( 1 + \frac{1}{\epsilon} \) levels, namely 0, \( h \epsilon, 2 h \epsilon, \ldots, h \) (constant \( 0 < \epsilon \leq 1 \)).
Example 5.

\[ T = \text{CACACAATACTATAC} \]
\[ A_0 = 16 \ 4 \ 14 \ 2 \ 7 \ 12 \ 5 \ 9 \ 15 \ 3 \ 8 \ 13 \ 6 \ 11 \ 10 \]
\[ n = 16 \]
\[ h = 4 \]
\[ \epsilon = \frac{1}{2} \]

\[ \psi_0 = 9 \ 10 \ 12 \ 14 \ 16 \ 1 \ 2 \ 4 \ 3 \ 5 \ 6 \ 15 \]
\[ B_0 = 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \]
\[ A_2 = 4 \ 1 \ 3 \ 2 \]
\[ \psi_2 = 4 \ 1 \ 3 \]
\[ B_2 = 1 \ 0 \ 0 \ 0 \]
\[ A_4 = 1 \]

Hence, bit-vector \( B_k \) has a '1' at position \( i \) iff \( A_k[i] \) is a multiple of \( 2^{h\epsilon + k} \).

Given all this, we have the following algorithm to compute \( A[i] \), to be invoked with \( \text{lookup}(i, 0) \).

**Algorithm 1: function \text{lookup}(i, k)**

```
if \( k = h \) then
  return \( A_h[i] \);
end
if \( k = \omega_k \) then
  return \( n_k \);
end
if \( B_k[i] = 1 \) then
  return \( 2^{h\epsilon} \text{lookup} \left( \text{RANK}_1(B_k, i), k + h\epsilon \right) \);
else
  return \( \text{lookup} \left( \psi_k(\text{RANK}_0(B_k, i), k) \right) - 1 \);
end
```

Here, \( \omega_k \) stores the position of the last suffix, i.e., \( A_k[\omega_k] = n_k \). Checking if \( i = \omega_k \) is necessary in order to avoid following \( \psi_k \) from the last suffixes to the first, because this would give incorrect results.

**Example 6.** \( A[15] = \text{lookup}(15, 0) = \text{lookup}(\psi_0(11), 0) - 1 = \text{lookup}(6, 0) - 1 = 2^2 \text{lookup}(3, 2) - 1 = 2^2(\text{lookup}(\psi_2(2), 2) - 1) - 1 = 2^2(\text{lookup}(1, 2) - 1) - 1 = 2^2(4 - 1) - 1 = 11 \)

To analyze the running time of the \text{lookup}-procedure, we first note that on every level \( k \), we need to follow \( \psi_k \) at most \( 2^{h\epsilon} \) times until we hit a position \( i \) with \( B_k[i] = 1 \) (second case of the last if-statement). Because the number of “implemented” levels, \( 1 + \frac{1}{\epsilon} \), is constant (remember \( \epsilon \) is constant!), the total time of the \text{lookup}-procedure is

\[
O \left( 2^{h\epsilon} \right) = O \left( \left( 2^{\log_{\log_\sigma n} n} \right)^{\epsilon} \right) = O \left( \log^\epsilon n \right)
\]

which is sub-logarithmic for \( \epsilon < 1 \).
1.5 Elias-Codes

For coding the $\psi$-values in a space efficient form, we will use Elias-$\gamma$ and Elias-$\delta$ codes, which we present in this section. Let us write $(x)_2$ for the binary representation of integer $x \geq 1$. Also $(x)_1$ denotes the unary representation of $x$, which consists of $x - 1$ 0’s, followed by a single 1. For example, $(5)_2 = 101$ and $(5)_1 = 00001$.

The Elias-$\gamma$ code of a number $x$, denoted by $(x)_\gamma$, is defined as follows: first, write the length of the binary representation of $x$ in unary, i.e., write bits $(|x|_2)_1$. Then append the bits from $(x)_2$, with the first (leftmost) '1' being omitted. For example, the first five $\gamma$-codes (representing the numbers 1, 2, ..., 5) are 1, 010, 011, 00100 and 00101.

The $\delta$-code is obtained in a similar manner, but instead of encoding $|x|_2$ in unary, we encode it with the $\gamma$-code. That is, we first write $(|x|_2)_\gamma$, and then append $(x)_2$, again with the trailing '1' being omitted. Examples of $\delta$-codes are shown in the following table.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$(x)_\delta$</th>
<th>$x$</th>
<th>$(x)_\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00100000</td>
<td>9</td>
<td>00100001</td>
</tr>
<tr>
<td>2</td>
<td>00100010</td>
<td>10</td>
<td>00100011</td>
</tr>
<tr>
<td>3</td>
<td>00100100</td>
<td>11</td>
<td>00100100</td>
</tr>
<tr>
<td>4</td>
<td>00100101</td>
<td>12</td>
<td>00100110</td>
</tr>
<tr>
<td>5</td>
<td>00100110</td>
<td>13</td>
<td>00100111</td>
</tr>
<tr>
<td>6</td>
<td>001010000</td>
<td>14</td>
<td>00101010</td>
</tr>
<tr>
<td>7</td>
<td>001010000</td>
<td>15</td>
<td>00101010</td>
</tr>
<tr>
<td>8</td>
<td>001010000</td>
<td>16</td>
<td>001010000</td>
</tr>
</tbody>
</table>

The size of the $\delta$-code is

$$|(x)_\delta| = |(\log x + 1)_\gamma| + |\log x|$$

$$= (\lfloor \log (\lceil \log x \rceil + 1) \rfloor + 1) + \lfloor \log (\lceil \log x \rceil + 1) \rfloor + |\log x|$$

$$= |\log x| + 2\lfloor \log (\lceil \log x \rceil + 1) \rfloor + 1 \text{ bits.}$$

1.6 Storing $\psi$

Let us first concentrate on level 0, i.e., on storing $\psi_0$. From Lemma 1, we know that $\psi$ is piecewise increasing in areas $A[l, r]$ where the suffixes start with the same character (i.e., where $T_A[i] = T_A[j]$ for all $i, j \in [l, r]$). Let $[l, r]$ be one such area. Instead of storing $\psi_0[i, r]$ plainly, we first compute the differences $\Delta_0[i] = \psi_0[i] - \psi_0[i - 1]$ for $l < i \leq r$. This produces a list of positive integers from the range $[1, n]$, which will be encoded space-efficiently in a subsequent step. In general, we define

$$\Delta_0[i] = \begin{cases} \psi_0[i] - \psi_0[i - 1] & \text{if } T_A_0[i] = T_A_0[i - 1]; \\ \psi_0[i] & \text{otherwise.} \end{cases}$$
Example 8.

\[ \Delta_0 = 9 \ 1 \ 2 \ 2 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ 1 \ 9 \]

These \( \Delta \)-values are now encoded with Elias \( \delta \)-code; the resulting bit stream is called \( S_0 \).

Example 9.

\[ \Delta_0 = 9 \ 1 \ 2 \ 2 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ 1 \ 9 \]

\[ S_0 = 00100001 \ 1 \ 0100 \ 0100 \ 0100 \ 1 \ 1 \ 0100 \ 0101 \ 0100 \ 1 \ 0010001 \]

In general, because \( A_k \) can be regarded as the suffix array of a text \( T^k \), we can compress \( \psi_k \) on levels \( k > 0 \) by the same mechanism, i.e., by using Elias \( \delta \)-codes on the list of differences of consecutive \( \psi_k \)-values. We therefore define

\[
\Delta_k[i] = \begin{cases} 
\psi_k[i] - \psi_k[i-1] & \text{if } T^k_{A_k[i]} = T^k_{A_k[i-1]}, \\
\psi_k[i] & \text{otherwise.}
\end{cases}
\]

How can we decompress the \( \psi_k \)-values from the stream \( S_k \) of \( \delta \)-encoded \( \Delta_k \)-values? For this purpose we store \( \psi_k[i] \) explicitly if either position \( i \) marks the beginning of a new character in \( T^k \) (second case in the definition of \( \Delta_k \)), or if the length of the encoded bit-stream since the last sampled \( \psi_k \)-value exceeds \( s = \log \frac{n}{2} \) bits. To implement this, we introduce three new arrays:

1. \( D_k \) is a bit vector such that \( D_k[i] = 1 \) iff \( \psi_k[i] \) is sampled. \( D_k \) is enhanced with data structures for constant-time \( \text{rank} \) and \( \text{select} \) queries.

2. \( R_k \) is an array that stores the sampled values of \( \psi_k \). All \( \psi_k \)-values stored in \( R_k \) are removed from the bit-stream \( S_k \) (they need not to be stored twice!).

3. \( P_k \) is a bit stream of the same size as \( S_k \) and marks those positions in \( S_k \) with a ’1’ where a \( \delta \)-encoded \( \Delta_k \)-value starts. \( P_k \) is prepared for \( O(1) \) \( \text{select} \)-queries. Then \( \text{select}_1(P_k, i) \) points to the \( i \)'th \( \Delta_k \)-value \( S_k[i] \).

Example 10. Assuming \( s = 5 \), we have the following structures:

\[
\begin{align*}
\psi_0 &= 9 \ 10 \ 12 \ 14 \ 16 \ 1 \ 2 \ 4 \ 3 \ 5 \ 6 \ 15 \\
\Delta_0 &= 9 \ 1 \ 2 \ 2 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ 1 \ 9 \\
D_0 &= 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \\
R_0 &= 9 \ 14 \ 1 \ 3 \ 15 \\
P_0 &= \begin{array}{cccccc}
11000 & 1000 & 11000 & 10001 & 00100001 \\
\end{array} \\
S_0 &= (00100001 \ 10100 \ 0100 \ 0100 \ 1 \ 10100 \ 01001 \ 00100001 \ 00100001)
\end{align*}
\]

We can decode \( \psi_k[i] \) as follows. First compute the number of sampled \( \Delta_k \)-values up to position \( i \) by \( y = \text{rank}_1(D_k, i) \). Then check if \( \Delta_k[i] \) is represented explicitly \( (D_k[i] = 1) \), and return \( R_k[y] \) in this case. Otherwise \( (D_k[i] = 0) \), compute the greatest index \( j \) such that \( \psi_k \) is sampled by \( j = \text{select}_1(D_k, y) \). The result is then \( R_k[y] (= \Delta_k[j]) \), plus the sum of the \( (i - j) \) values \( \Delta_k[j+1], \ldots, \Delta_k[i] \) that follow \( \Delta_k[j] \) in \( S_k \). Note that \( D_k[j+1] = 0 \), and that the 0’s in \( D_k \) corresponds to the 1’s in \( P_k \). As \( \Delta_k[j+1] \) is the \( z \)'th encoded \( \Delta_k \)-value in \( S_k \), with \( z = \text{rank}_0(D_k, j+ \)
1) $j + 1 - \text{RANK}_1(D_k, j + 1) = j + 1 - y$, we thus go to position \text{SELECT}_1(P_k, z) in $S_k$, from where we decode the values $\Delta_k[j+1], \ldots, \Delta_k[i]$, and return $R_k[y] + \sum_{i=j+1}^{i} \Delta_k[i]$ as the result $\psi_k[i]$. This decoding is possible because the $\delta$-code is prefix-free (no codeword is a prefix of a different codeword).

To compute this sum in $O(1)$ time, we use again the Four-Russians-Trick: in a global lookup-table $G$, for all bit-vectors $V$ of length $s$ and all positions $i \in [1, s]$, $G[V][i]$ stores the answer to $\sum_{j=1}^{i} y_j$, if we interpret $V$ as a sequence of $\delta$-encoded values $y_1, y_2, \ldots$. Note that some values in $G$ are undefined, because not at all positions $i \in [1, s]$ there ends a $\delta$-encoded value in $V$, and not all bit-vectors $V$ represent a correct sequence of $\delta$-codes, but these values will never be accessed by the algorithm.

Example 11.

<table>
<thead>
<tr>
<th>$V$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10100</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

1.7 Space Analysis

We now analyze the space requirement of the compressed suffix array. Recall that on level $k < h$, we store bit-vectors $B_k$, $D_k$, $S_k$, and $P_k$ (plus some data structures for \text{RANK} and \text{SELECT}), and array $R_k$. On level $h$, we only store $A_h$, which needs $O(n \log \sigma)$ bits. Thus it remains to be shown that an level $k < h$ the space is $O(n \log \sigma)$ bits. Then the total space on all $1 + \frac{1}{\epsilon}$ levels is $O\left(\frac{1}{\epsilon} n \log \sigma\right)$ bits.

The bit-vectors $B_k$ and $D_k$ are certainly of size $O(n)$ bits each, as they are never longer than $n$, the length of the text. Actually, the total size of all $B_k$’s can be bounded by $2n$ bits, because the length of the $B_k$-vectors is at least halved from one level to the next:

$$\sum_{k=0}^{h-1} |B_k| = \sum_{k=0}^{h-1} n_k = \sum_{k=0}^{h-1} n \frac{1}{2^k} = n \sum_{k=0}^{\infty} \frac{1}{2^k} \leq n \sum_{k=0}^{\infty} \frac{1}{2^k} = 2n.$$ 

The total size of the $D_k$’s is even smaller. Together with the data structures for constant-time \text{RANK}- and \text{SELECT}-queries, the space for all $B_k$’s and $D_k$’s can be upper bounded by $4n + o(n)$ bits in total.

Let us now analyze the space for the bit-stream $S_k$ on a fixed level $k < h$. For simplicity, we assume that $S_k$ stores all $\Delta_k$-values, also those that are stored explicitly in $R_k$ and thus deleted from $S_k$. Let $n_k^c$ denote the number of positions in $\psi_k$ corresponding to suffixes that start with the same character $c \in \Sigma^k$, and let $\Delta_k^c[1, n_k^c]$ denote the corresponding sub-array in $\Delta_k$. Thus,
by Lemma 1, $S_k$ stores at most $\sigma^{2^k}$ increasing sequences from the range $[1, n_k]$, each encoded by $\delta$-codes of the differences $\Delta_k$. Therefore, the space is

$$|S_k| = \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} (\lfloor \log \Delta^c_k[i] \rfloor + 2 \lfloor \log (\lfloor \log \Delta^c_k[i] \rfloor + 1) \rfloor + 1)$$

$$= \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} (\lfloor \log \Delta^c_k[i] \rfloor + 2 \log \log \Delta^c_k[i]) + O(n_k)$$

$$\leq \sum_{c \in \Sigma^{2^k}} \sum_{i=1}^{n_k^c} \left( \log \frac{n_k}{n_k^c} + 2 \log \log \frac{n_k}{n_k^c} \right) + O(n_k)$$

$$= \sum_{c \in \Sigma^{2^k}} n_k^c \left( \log \frac{n_k}{n_k^c} + 2 \log \log \sigma^{2^k} \right) + O(n_k)$$

$$\leq \sum_{c \in \Sigma^{2^k}} n_k^c \left( \log \sigma^{2^k} + 2 \log \log \sigma^{2^k} \right) + O(n_k)$$

$$= \left( \log \sigma^{2^k} + 2 \log \log \sigma^{2^k} \right) \sum_{c \in \Sigma^{2^k}} n_k^c + O(n_k)$$

$$= \left( 2^k \log \sigma + 2 \log 2^k \log \sigma \right) n_k + O(n_k)$$

$$= \left( 2^k \log \sigma + 2 \log 2^k \log \sigma \right) \frac{n}{2^k} + O(n_k)$$

$$= n \log \sigma + O(n \log \log \sigma) \text{ bits.}$$

Here, both inequalities follow from the fact that the sum of logarithms is largest when the values are spread evenly over the interval: if $\sum_{i=1}^{m} x_i \leq x$ for a sequence of $m$ real numbers with $x_i \geq 1$ for all $i$, then $\sum_{i=1}^{m} \log x_i \leq \sum_{i=1}^{m} \log \frac{x}{m}$.

Because $P_k$ is of the same size as $S_k$, we can upper bound the space for $P_k$ (including the data-structure for SELECT) by $O(n \log \sigma)$ bits.

Finally, the array $R_k$ of sampled values consist of

$$|R_k| = \left( \frac{\sum_{i=1}^{m} x_i}{\log n} \right) \times \frac{|S_k|}{\text{value from } \lfloor 1, n_k \rfloor} \times \log n_k$$

$$= \left( \sigma^{2^k} + \frac{n \log \sigma}{\log n} \right) \log n_k$$

$$\leq O \left( \left( \sigma^{2^k} + \frac{n \log \sigma}{\log n} \right) \log n \right)$$

$$= O \left( \left( \frac{n}{\log n} + \frac{n \log \sigma}{\log n} \right) \log n \right)$$

$$= O(n \log \sigma) \text{ bits.}$$
We summarize this section in a final theorem:

**Theorem 2.** The suffix array $A$ of a text of length $n$ over an alphabet of size $\sigma$ can be stored in $O\left(\frac{1}{\epsilon} n \log \sigma\right)$ bits such that retrieving an arbitrary entry $A[i]$ from the suffix array with $1 \leq i \leq n$ takes $O(\log^\epsilon n)$ time.