Summary

We have seen text compression techniques from three broad categories:

- Context-based compression
  - – Slow compression and decompression, because they require a complex model for good compression.
  - + Highest compression rates because of the best entropy models. Models are easy to tune as they operate directly on the text.
- Dictionary compression
  - + Very fast and simple decompression.
  - – Often not the best compression. Better compression would require better entropy models on phrases, which are less intuive. Besides complex models could destroy the main advantage.
- Burrows-Wheeler compression
  - + Fairly fast and simple decompression.
  - + Good compression with relatively simple models.
  - – Fairly complex and slow BW transform algorithm.

Definition 2.25: A text compression algorithm is called *coarsely optimal* if the compressed size of a text T of length n over an alphabet of size σ is bounded by

$$nH_k(T) + o(n \log \sigma)$$

for any $k = o(\log_\sigma n)$.

Many algorithms in each of the three categories can be shown to be coarsely optimal. Algorithms that are not include:

- LZ77 with constant window size.
- Context-based and Burrows-Wheeler compressors that use Huffman coding or constant precision arithmetic coding. (On the other hand, $\log n$ bit precision is enough.)
- For Re-Pair a weaker bound, $2nH_k(T) + o(n \log \sigma)$, has been proven.

Coarse optimality has its limitations as a measure of compression:

- Coarse optimality implies a certain kind of optimality *in the limit*, i.e., for large enough n, but there is no bound on what is large enough.
- Coarse optimality does not guarantee good compression of highly compressible texts.

3. Compressed Data Structures

A data structure is a method of storing data in a form that supports efficient execution of certain kinds of operations on the data. Often additional functionality is achieved by storing additional data such as pointers.

In data compression, the goal is to reduce the amount of stored data, often at the cost of reduced functionality. Indeed, most compression methods are designed to support only one operation on the compressed data: decompression.

In this section, we will see how one can

- add functionality with minimal addition of data
- compress data without losing functionality.

Compressed data structures are often slower than uncompressed ones, but if compression helps the data to fit in a faster memory, it can speed up algorithms.

Bit vectors

The work horse of compressed data structures is the bit vector. Let $B[0..u)$ be a bit vector with n 1-bits and $u - n$ 0-bits. A basic operation on B is

$$\text{access}_B(i) = B[i].$$

Using a standard representation of bitvectors, this is a trivially constant time operation.

We can compress B close to $uH_0(B)$ bits using arithmetic coding with probability n/u for 1's and $(u - n)/u$ for 0's, but then $\text{access}_B(i)$ cannot be implemented efficiently.

We will see in a moment that there are other ways of compressing B that support fast, even constant time access.

Rank and select

Two useful operations on bit vectors are

$$\text{rank-1}_B(i) = |\{j \in [0..i) \mid B[j] = 1\}| \quad \text{for } i \in [0..u]$$
$$\text{select-1}_B(j) = \max\{i \in [0..n] \mid \text{rank-1}_B(i) = j\} \quad \text{for } j \in [0..n]$$

select-$1_B(j)$ is the position of the $(j+1)$st 1-bit. Note also that

$$\text{rank-1}_B(\text{select-1}_B(j)) = j.$$

Sometime we also need rank-0 and select-0 defined symmetrically.

Example 3.1:

i	0	1	2	3	4	5	6	7
$B[i]$	0	1	1	1	0	0	0	
rank-$1_B(i)$	0	0	1	2	3	3	3	3
rank-$0_B(i)$	0	1	1	1	1	2	3	4
select-$1_B(i)$	1	2	3	7				
select-$0_B(i)$	0	4	5	6	7			

Applications of rank and select

Before showing, how to implement rank and select operations, let us see some of their applications.

Set of integers. Let S be a set of n integers from the universe $U = [0..u)$. We can represent S using its *characteristic bit vector* $1_S[0..u)$ defined as

$$1_S[i] = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{if } i \notin S \end{cases}$$

Using this representation, we can implement basic operations on S:

$$\text{contains}_S(i) = \text{access}_{1_S}(i)$$
$$\text{rank}_S(i) = \text{rank-1}_{1_S}(i) = |S \cap [0..i)|$$
$$\text{select}_S(j) = \text{select-1}_{1_S}(j)$$

Example 3.2: If $S = \{1,2,3\} \subseteq [0..7)$, then $1_S = 0111000$.

Sparse array. Let $A[0..u)$ be an array of arbitrary element type, where $u-n$ entries contain a *null* value. We can implement A using a bit vector $1_A[0..u)$ and an array $A'[0..n)$ as follows:

$$1_A[i] = \begin{cases} 1 & \text{if } A[i] \text{ is non-null} \\ 0 & \text{if } A[i] \text{ is null} \end{cases}$$
$$A'[j] = A[\text{select-1}_{1_A}(j)]$$

Then

$$A[i] = \begin{cases} A'[\text{rank-1}_{1_A}(i)] & \text{if access}_{1_A}(i) = 1 \\ \text{null} & \text{if access}_{1_A}(i) = 0 \end{cases}$$

Searchable prefix sums (searchable partial sums). Let $L[0..n)$ be a sequence of n positive integers summing up to u. (In the exercises, this result is extended to non-negative integers.)

We want to have the following operations on L:

$$\text{sum}_L(j) = \sum_{i \in [0..j)} L[i] \quad \text{for } j \in [0..n]$$
$$\text{search}_L(i) = \max\{j \in [0..n] \mid \text{sum}_L(j) \leq i\} \quad \text{for } i \in [0..u]$$

We can implement L using a bit vector $B_L[0..u)$ as follows:

$$B_L[i] = \begin{cases} 1 & \text{if sum}_L(j) = i \text{ for some } j \\ 0 & \text{otherwise} \end{cases}$$

Then

$$\text{sum}_L(j) = \text{select-1}_{B_L}(j)$$
$$\text{search}_L(i) = \text{rank-1}_{B_L}(i+1) - 1$$

Random access to variable length encoding. Let $T[0..n)$ be a text over an alphabet Σ. Let C be variable length binary code for Σ and let $u = |C(T)|$ be the length of the code for T.

We want to store the encoded representation $C(T)$ but still support fast access to any symbol. For this we need the operation $\mathsf{pos}_{C(T)}(j)$ that returns the position of the first bit of the code for $T[j]$. Then

$$C(T[j]) = C(T)\big[\mathsf{pos}_{C(T)}(j)..\mathsf{pos}_{C(T)}(j+1)\big)$$

and we can quickly decode $T[j]$.

The pos operation can be implemented using searchable prefix sum data structure for the sequence $L[0..n)$ defined as

$$L[j] = |C(T[j])|.$$

Then

$$\mathsf{pos}_{C(T)}(j) = \mathsf{sum}_L(j).$$

The reverse mapping from a position in $C(T)$ to a position in T can be implemented using search_L.

Succinct rank

Now we will see how to implement rank and select operations on a bit vector $B[0..u)$. Let us start with rank-1. We will describe a data structure that consists of B itself and additional data of $o(u)$ bits. Such a data structure using a sublinear amount of additional space is called *succinct*.

There are two trivial implementations of rank-1:

- Store an array $R[0..u]$ with $R[i] = \text{rank-1}_B(i)$. This supports constant time rank-1 but needs $\Omega(u \log n)$ bits of space.
- Store only B using u bits. To compute $\text{rank-1}_B(i)$ we have to scan $B[0..i)$ (or $B[i..u)$) and count bits.

We can speed up bit counting using the fact that one machine word contains multiple bits:

- Store the bit counts for all combinations of $(\log u)/2$ bits into a lookup table of $2^{(\log u)/2} = \mathcal{O}(\sqrt{u})$ entries and $\mathcal{O}(\sqrt{u}\log\log u)$ bits. Then a scan of k bits can be performed in $\mathcal{O}(1 + k/\log u)$ time.
- In practice, it is better to do bit counting using hardware instructions if available or certain bit tricks.

We can combine the trivial solutions as follows:

- Divide B into *blocks* of size b. For simplicity, assume that u is a multiple of b. Then the blocks are $B_i = B[ib..(i+1)b)$, $i \in [0..n/b)$.
- Store ranks at the the block boundaries in an array $R[0..n/b]$: $R[i] = \text{rank-1}_B(ib)$.
- Then
$$\text{rank-1}_B[i] = R[k] + \text{bitcount}(B[kb...i))$$
where $k = \lfloor i/b \rfloor$. The bit count term is computed by scanning using lookup table of $\mathcal{O}(\sqrt{u}\log\log u)$ bits.

This solution needs $\mathcal{O}((u\log u)/b + \sqrt{u}\log\log u)$ bits in addition to B and supports rank queries in $\mathcal{O}(1 + b/\log u)$ time. The choice of b gives us a space–time tradeoff. For example:

b	additional space	time
$\log u$	$\mathcal{O}(u)$ bits	$\mathcal{O}(1)$
$(\log u)^2$	$\mathcal{O}(u/\log u)$ bits	$\mathcal{O}(\log u)$

For the final solution, we use *two levels of blocks*:

- The *superblock* size is $b_1 = (\log u)^2$. The ranks at superblock boundaries are stored in an R_1 as before, which needs $\mathcal{O}(u/\log u) = o(u)$ bits.
- Each superblock is divided into blocks of size is $b_2 = \log u$. An array R_2 stores the ranks relative to the nearest preceding superblock boundary:
$$R_2[i] = \text{rank-1}_B[ib_2] - R_1[\lfloor ib_2/b_1 \rfloor].$$
Using $\mathcal{O}(\log b_1) = \mathcal{O}(\log\log u)$ bits per entry, the array R_2 needs $\mathcal{O}((u\log\log u)/\log u) = o(u)$ bits.
- Then
$$\text{rank-1}_B[i] = R_1[k_1] + R_2[k_2] + \text{bitcount}(B[k_2b_2...i))$$
where $k_1 = \lfloor i/b_1 \rfloor$ and $k_2 = \lfloor i/b_2 \rfloor$. The bit count can be computed in constant time using a lookup table of size $\mathcal{O}(\sqrt{u}\log\log u) = o(u)$ bits.

Theorem 3.3: A bit vector $B[0..u)$ can be augmented with a data structure of $\mathcal{O}((u\log\log u)/\log u) = o(u)$ bits so that rank-1 queries can be answered in constant time.

Succinct select

Now we will see how to implement the select-1 operation on a bit vector $B[0..u)$ containing n 1-bits.

We have similar basic solutions as with rank-1:

- Store an array $S[0..n]$ with $S[j] = \text{select-1}_B(j)$. This supports constant time select-1 using $\mathcal{O}(n \log u)$ bits. Note that if $n = \mathcal{O}(u/(\log u)^2)$, this solution already matches the result for rank-1.
- Store just B. To compute $\text{select-1}_B(j)$, scan B from the beginning until the number of 1-bits reaches j.
- As with rank-1, we can scan $\log u$ bits at a time using a lookup table of size $\mathcal{O}(\sqrt{u} \log\log u)$ bits. Since this optimized scan may overshoot by up to $\log u$ bits, we need another lookup table locate the desired position within a region of $\log u$ bits. This second lookup table has $\mathcal{O}(\sqrt{u} \log u)$ entries and needs $\mathcal{O}(\sqrt{u}(\log u) \log\log u)$ bits.

As with rank-1, the solution involves two levels of blocks.

- Divide B into n/b_1 superblocks $B_0, B_1, \ldots, B_{n/b_1-1}$ so that each superblock contains exactly $b_1 = (\log u)^2$ 1-bits. Then
$$\text{select-1}_B(j) = \sum_{h\in[0..k)} |B_h| + \text{select-1}_{B_k}(j - kb_1)$$
where $k = \lfloor j/b_1 \rfloor$. We need $\mathcal{O}((n \log u)/b_1) = \mathcal{O}(n/\log u)$ bits to store the prefix sums $\sum_{h\in[0..k)} |B_h|$ for all $k \in [0..n/b_1]$.

The implementation of the local select query $\text{select-1}_{B_k}(j - kb_1)$ depends on the size of the superblock B_k:

- If $|B_k| \geq (\log u)^4$, we store an array $S_k[0..b_1)$ recording the answers to all queries using $\log u$ bits per entry. The array needs $\mathcal{O}(b_1 \log u) = \mathcal{O}((\log u)^3)$ bits. Since there are at most $\mathcal{O}(u/(\log u)^4)$ such superblocks, the total space is $\mathcal{O}(u/\log u)$ bits.
- If $|B_k| < (\log u)^4$, we divide the superblock into smaller blocks with each block containing $b_2 = \sqrt{\log u}$ 1-bits. As with superblocks, we need prefix sums and local select operations. The prefix sum arrays fit into $\mathcal{O}((n \log\log u)/b_2) = \mathcal{O}(n \log\log u/\sqrt{\log u})$ bits.

The implementation of the local select query on a block B'_k depends on the size of the block:

- If $|B'_k| \geq \log u$, we store all answers in an array $S'_k[0..b_2)$ of $\mathcal{O}(b_2 \log\log u)$ bits. Since there are at most $u/\log u$ such blocks, the total space is $\mathcal{O}(u \log\log u/\sqrt{\log u})$.
- If $|B'_k| < \log u$, we can use lookup tables of size $\mathcal{O}(\sqrt{u}(\log u)^2)$ bits to answer the local select query.

Theorem 3.4: A bit vector $B[0..u)$ can be augmented with a data structure of $\mathcal{O}((u \log\log u)/\sqrt{\log u}) = o(u)$ bits so that select-1 queries can be answered in constant time.

- Practical implementations are similar to the data structures described above, though they tend to use larger block sizes, since scanning as a sequential operation is very fast.