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.

Compressed bit vectors

The rank and select data structures we just saw consists of the bit vector B of u bits and additional data structures of $o(u)$ bits. To achieve compression, we can replace B with a compressed representation of B, and keep the other data structures as is.

- Divide B into blocks of size $b = (\log u)/2$, and consider the result $B_0, B_1, \ldots, B_{n/b-1}$ as sequence of length u/b over an alphabet of size $2^b \approx \sqrt{u}$.
- Encode the sequence of blocks using Huffman coding.
- Add a prefix sum data structure to provide random access to the blocks.

With this representation of B, we can access any block of size b in constant time, which is sufficient for implementing rank and select as we just saw. In addition, access queries can be answered in constant time too.

Let us analyze the Huffman coded representation.

- Let us assign the probability $p = n/u$ to 1-bits and $1 - p$ to zero bits. The encoding of B using exact arithmetic coding has size at most $uH_0(B) + 2$ bits.
- We can extend the probabilities to the blocks: a block of size b with ℓ 1-bits has the probability $p^\ell(1-p)^{b-\ell}$. The encoding of the sequence of blocks using exact arithmetic coding has size at most $uH_0(B) + 2$ bits too.
- If we replace arithmetic coding with Huffman coding, the size increases by less than one bit per symbol. For the sequence of blocks, this adds at most $u/b = \mathcal{O}(u/\log u)$ bits.
- If we use the actual frequencies of the blocks, the size of the encoding can only go down. Thus the Huffman coded sequence of blocks uses at most $uH_0(B) + \mathcal{O}(u/\log u)$ bits.
- Using the prefix sums data structure, we can easily find the length of each code word. Assuming canonical Huffman code, we can then decode each codeword with a single table lookup. The total size of the decoding tables is $\mathcal{O}(\sqrt{u}\log u)$.

Finally, let us analyze the prefix sum data structure.

- We need to compute prefix sums over the sequence $L[0..u/b)$ of block code word sizes. As we have seen, this can be implemented with a bit vector B_L of length $uH_0(B) + \mathcal{O}(u/\log u) = \mathcal{O}(u)$ with exactly u/b 1-bits.
- We will only need the sum operation, not the search operation. Thus we need to implement select-1 over B_L.
- Using the simple solution of storing all select values in a table $S[0..u/b)$, we need $\mathcal{O}((u/b)\log u) = \mathcal{O}(u)$ bits.
- To reduce space, divide B_L into blocks so that each block contains $\log u$ 1-bits. Since no Huffman code is longer than $\mathcal{O}(\log u)$, no block is longer than $\mathcal{O}((\log u)^2)$.
- We can store full select values at the block boundaries using $\mathcal{O}(u/\log u)$ bits. A select value relative to the beginning of the block needs $\mathcal{O}(\log\log u)$ bits, so we can store all relative select values in $\mathcal{O}((u/b)\log\log u) = \mathcal{O}(u(\log\log u)/\log u) = o(u)$ bits.

Theorem 3.5: A bit vector $B[0..u)$ can be stored in $uH_0(B) + o(u)$ bits so that the operations access, rank-1, select-1, rank-0 and select-0 can be answered in constant time.

In the data structure we just described, the term $o(u)$ is more precisely $\mathcal{O}(u(\log\log u)/\sqrt{\log u})$, but it can be reduced further:

- The best theoretical result is Patrascu's Succincter data structure using $uH_0(B) + \mathcal{O}(u/(\log u)^k)$ for any constant k.
- The RRR data structure by Raman, Raman and Rao using $uH_0(B) + \mathcal{O}(u(\log\log u)/\log u)$ bits has been implemented and works well in practice.
- Some techniques can get below $uH_0(B)$ if the bit vector has some context dependecies, but not much research exists in this direction.
- A special case is select-1 for sparse bit vectors with few 1-bits. The simple solution of storing all the select values in an array takes just $\mathcal{O}(n\log u)$ bits and can be reduced further as we did on the previous slide. This is not less than $uH_0(B)$, but for small n, it is less than the $o(u)$ term of many data structures.

All the data structures use similar techniques:

- Divide data into blocks, superblocks, and sometimes even more levels of blocks.
- Store data at the block boundaries and handle the smallest blocks with lookup tables.

Avoiding redundancy is important for minimizing the space:

- The Huffman encoded sequence of blocks contains all the information of bit vector. All the information in the additional data structures is fully redundant.
- The best data structures store the bit vector in a form that omits data that is already in the other data structures.
- It is also common to implement only some of the operations directly using additional data structures. The other operations can be implemented through calls to the directly implemented ones (exercise).

Rank and select for larger alphabets

Rank and select are useful operations on sequences over larger alphabets too.

Let $S[0..n)$ be a sequence over an alphabet $\Sigma = [0..\sigma)$. Define

$$\begin{aligned} \mathrm{access}_S(i) &= S[i] \quad \text{for } i \in [0..n) \\ \mathrm{rank}_S(c,i) &= |\{j \in [0..i) \mid S[j] = c\}| \quad \text{for } c \in \Sigma,\ i \in [0..n] \\ \mathrm{select}_S(c,j) &= \max\{i \in [0..n] \mid \mathrm{rank}_S(c,i) = j\} \quad \text{for } c \in \Sigma,\ j \in [0..\mathrm{rank}_S(c,n)] \end{aligned}$$

Example 3.6:

i	0	1	2	3	4	5	6	7	8	9	10	11
$S[i]$	a	b	r	a	c	a	d	a	b	r	a	
$\mathrm{rank}_S(\mathtt{a},i)$	0	1	1	1	2	2	3	3	4	4	4	5
$\mathrm{select}_S(\mathtt{a},i)$	0	3	5	7	10	11						

Application: Permutations

A permutation is a bijective mapping $\pi : [0..n) \to [0..n)$. We can represent the permutation as a sequence $\pi[0..n)$: $\pi[i] = \pi(i)$. Then $\pi(i) = \mathrm{access}_\pi(i)$ and $\pi^{-1}(i) = \mathrm{select}_\pi(i, 0)$.

Let $S[0..n)$ be a sequence over an alphabet $\Sigma = [0..\sigma)$ such that the permutation π performs a stable sorting of S. π itself is such a sequence but not the only one. S may have a smaller alphabet or even be compressible.

Let $L[0..\sigma)$ be an array, where $L[a]$ is the number of occurrences of a in S. Then

$$\begin{aligned} \pi(i) &= \mathrm{sum}_L(c) + \mathrm{rank}_S(c,i) \quad \text{where } c = \mathrm{access}_S(i) \\ \pi^{-1}(i) &= \mathrm{select}_S(c, i - \mathrm{sum}_L(c)) \quad \text{where } c = \mathrm{search}_L(i) \end{aligned}$$

Example 3.7:

π	3	7	0	1	4	2	5	6
S	b	c	a	a	b	a	b	b

	a	b	c
L	3	4	1

$$\begin{aligned} \pi(4) &= \mathrm{sum}_L(\mathtt{b}) + \mathrm{rank}_S(\mathtt{b}, i) = 3 + 1 = 4 \\ \pi^{-1}(4) &= \mathrm{select}_S(\mathtt{b}, 4 - \mathrm{sum}_L(\mathtt{b})) = \mathrm{select}_S(\mathtt{b}, 1) = 4 \end{aligned}$$

Wavelet tree

The most common data structure for large alphabet rank and select is the wavelet tree.

A wavelet of a sequence S over an alphabet $\Sigma = [0..\sigma)$ is defined as follows:

- Let $\mathcal{T}$ be a binary tree with σ leaves labelled by the symbols of the alphabet Σ. For each $a \in \Sigma$, let $C(a)$ be the binary string representing the path from the root to the leaf labelled by a. Note that C is a prefix code, and that for any prefix code there exist the matching binary tree.
- For each node v in $\mathcal{T}$, let Σ_v be the subset of symbols in the subtree rooted at v, and let S_v be the subsequence of S consisting of all the symbols of Σ_v. In other words, S_v is obtained from S by removing all symbols that are not in Σ_v.
- For each internal node v in $\mathcal{T}$, let B_v be a bit vector with the same lenght as S_v defined as

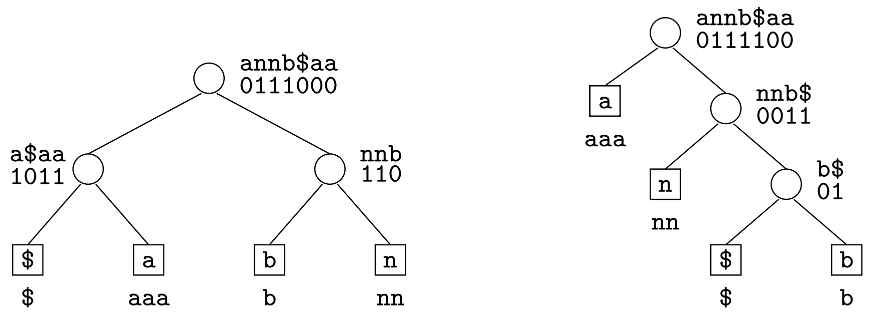
$$B_v[i] = \begin{cases} 0 & \text{if } S_v[i] \text{ is in the left subtree of } v \\ 1 & \text{if } S_v[i] \text{ is in the right subtree of } v \end{cases}$$

- The wavelet tree consists of the tree $\mathcal{T}$ with the leaf labels, and the bit vectors B_v processed to support rank and select queries.

A different binary tree / prefix code gives a different wavelet tree. The most common choices are:

- balanced tree / fixed length code
- Huffman tree / Huffman code

Example 3.8: Balanced and Huffman wavelet trees for the string $S = \mathtt{annb\$aa}$.



Given the wavelet tree of S (what ever the shape of the tree), we can implement $\text{access}_S(i)$ as follows.

WT-access(i)

```
v ← root; r ← i
while v is not a leaf do
    if access_{B_v}(r) = 0 then
        r ← rank-0_{B_v}(r)
        v ← leftchild(v)
    else
        r ← rank-1_{B_v}(r)
        v ← rightchild(v)
return label of v
```

- The final value of r is $\text{rank}_S(S[i], i)$.

The implementations of rank and select are equally simple.

WT-rank(c, i)

```
v ← root; r ← i
while v is not a leaf do
    if c is in the left subtree of v then
        r ← rank-0_{B_v}(r)
        v ← leftchild(v)
    else
        r ← rank-1_{B_v}(r)
        v ← rightchild(v)
return i
```

WT-select(c, i)

```
v ← leaf representing c; r ← i
while v is not root do
    p ← parent(v)
    if v is in the left child of p then
        r ← select-0_{B_p}(r)
    else
        r ← select-1_{B_p}(r)
    v ← p
return r
```


Let us analyze the size of the wavelet tree.

- The tree structure, the leaf labels and the pointers to the bit vectors B_v fit in $\mathcal{O}(\sigma \log n)$ bits.
- The contribution of a symbol $S[i]$ to the total length of the bit vectors is equal to the length of its code word $C(S[i])$. Thus the total length of the bit vectors is the same as the length of the code $C(S)$. This is $n\lceil \log \sigma \rceil$ for the balanced tree and less than $n(H_0(S)+1)$ for the Huffman tree.
- Adding support for select and rank changes n into $n + o(n)$, and adds $\mathcal{O}(\sigma \log n)$ bits.
- Compressed representation of the bit vectors reduces the space to $nH_0(S) + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits with the balanced tree and to $nH_0(S) + o(n(H_0(S)+1)) + \mathcal{O}(\sigma \log n)$ bits with the Huffman tree.

The time complexity of all the operations is proportional to the length of the code word for the symbol involved. No Huffman code word can be longer than $\mathcal{O}(\log n)$.

Theorem 3.9: Let $S[0..n)$ be a string over the alphabet $[0..\sigma)$.

The balanced wavelet tree of S needs

$$(n + o(n)) \log \sigma + \mathcal{O}(\sigma \log n)$$

bits using uncompressed bit vectors and

$$nH_0(S) + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$$

bits using compressed bit vectors and supports access, rank and select queries in $\mathcal{O}(\log \sigma)$ time.

The Huffman wavelet tree of S needs

$$(n + o(n))(H_0(S) + 1) + \mathcal{O}(\sigma \log n)$$

bits using uncompressed bit vectors and

$$nH_0(S) + o(n(H_0(S) + 1)) + \mathcal{O}(\sigma \log n)$$

bits using compressed bit vectors and supports access, rank and select queries in $\mathcal{O}(\log \sigma)$ worst case time and in $\mathcal{O}(H_0(S))$ average time.