Derivation of loopless reflected mixed-radix Gray generation, Algorithm 7.2.1.1H, p. 300

We start with the simple problem generating all tuples $(b_{n-1}, \ldots, b_1, b_0)$ with $0 \le b_i < m_i$ in lexicographic order. (Knuth, TAOCP, Vol. 4A, Algorithm 7.2.1.1M, p. 282)

A digit b_i is *active* if $b_i < m_i - 1$.

A digit b_i is *passive* if $b_i = m_i - 1$: Such a digit waits until a higher-order digit b_j with $j > i$ changes before it starts running again.

So far, this information is redundant, because it can be read off directly from the value of the digit itself. Nevertheless, we will keep it in an array $active[i]$. We also set $active[n] = True$.

> $(b_{n-1}, \ldots, b_1, b_0) := (0, 0, \ldots, 0, 0)$
> for $i = 0, \ldots, n$: $active[i] := True$
> Main loop:
>   VISIT $(b_{n-1}, \ldots, b_1, b_0)$
>   look for the rightmost active digit $j = \rho(k)$,
>   where $k = (b_{n-1}, \ldots, b_1, b_0)$ in mixed-radix representation
>   and make all intervening digits active:
> $j := 0$
>   **while not** $active[j]$:
> $b_j := 0$
> $active[j] := True$
> $j := j + 1$
>   **if** $j = n$: TERMINATE
> $b_j := b_j + 1$
>   **if** $b_j = m_j - 1$:
> $active[j] := False$

First extension: Gray code

We simultaneously produce the Gray code for all tuples $(a_{n-1}, \ldots, a_1, a_0)$ with $0 \le a_i < m_i$. The delta sequence $\rho(k)$ $(k = 1, 2, \ldots)$ of the Gray code is place until which the carry propagates in lexicographic generation. It equals the ruler function.
Since each digit a_i goes alternatively up and down, we need direction variables $d_i \in \{+1, -1\}$ for $i = 0, \ldots, n - 1$.

> $(b_{n-1}, \ldots, b_1, b_0) := (0, 0, \ldots, 0, 0)$
> $(a_{n-1}, \ldots, a_1, a_0) := (0, 0, \ldots, 0, 0)$
> for $i = 0, \ldots, n$: $active[i] := True$
> for $i = 0, \ldots, n - 1$: $d_i := +1$.
> Main loop:
>   VISIT $(b_{n-1}, \ldots, b_1, b_0)$ / VISIT $(a_{n-1}, \ldots, a_1, a_0)$
>   look for the rightmost active digit $j = \rho(k)$, and make all intervening digits active:
> $j := 0$
>   **while not** $active[j]$:
> $b_j := 0$
> $active[j] := True$
> $j := j + 1$
>   **if** $j = n$: TERMINATE
> $b_j := b_j + 1$
> $a_j := a_j + d_j$
>   **if** $b_j = m_j - 1$: (OR EQUIVALENTLY:)
>   **if** $a_j = m_j - 1$ **or** $a_j = 0$: (alternative test)
> $active[j] := False$
> $d_j := -d_j$

Second extension: Skip pointers

The goal is to eventually avoid the loop that searches for the rightmost active digit $j = \rho(k)$. Thus we establish *skip pointers* $f[i]$, $i = 0, \ldots, n-1$, which have the following meaning. If $b_j, b_{j-1}, \ldots, b_i$, for $j \geq i$, is a maximal block of consecutive passive digits, then $f[i] = j+1$. All other skip pointers point to themselves: $f[i] = i$.
These pointers allow us to find the next active digit quickly.

> $(b_{n-1}, \ldots, b_1, b_0) := (0, 0, \ldots, 0, 0)$
> for $i = 0, \ldots, n$: $\mathit{active}[i] := \mathit{True}$
> for $i = 0, \ldots, n$: $f[i] := i$.
> Main loop:
>> VISIT $(b_{n-1}, \ldots, b_1, b_0)$
>> look for the rightmost active digit $j = \rho(k) = f[0]$, and make all intervening digits active:
>> $j := 0$
>> **while** $j < f[0]$:
>>> $b_j := 0$
>>> $\mathit{active}[j] := \mathit{True}$
>>> $j := j + 1$
>>
>> $f[0] := 0$ (This may be redundant.)
>> **if** $j = n$: TERMINATE
>> $b_j := b_j + 1$
>> **if** $b_j = m_j - 1$:
>>> $\mathit{active}[j] := \mathit{False}$
>>> $f[j] := f[j+1]$
>>> $f[j+1] := j + 1$ (This may be redundant.)

This did not make the program faster, because we still have to set each digit b_j to zero while increasing j. (But we could eliminate $\mathit{active}[j]$ now. It is implicitly given by the f pointers.)

Combining the two extensions

Now we combine the two extensions:

> $(b_{n-1}, \ldots, b_1, b_0) := (0, 0, \ldots, 0, 0)$
> $(a_{n-1}, \ldots, a_1, a_0) := (0, 0, \ldots, 0, 0)$
> for $i = 0, \ldots, n$: $\mathit{active}[i] := \mathit{True}$
> for $i = 0, \ldots, n-1$: $d_i := +1$.
> for $i = 0, \ldots, n$: $f[i] := i$.
> Main loop:
>> VISIT $(b_{n-1}, \ldots, b_1, b_0)$ / VISIT $(a_{n-1}, \ldots, a_1, a_0)$
>> look for the rightmost active digit $j = \rho(k) = f[0]$, and make all intervening digits active:
>> $j := 0$
>> **while** $j < f[0]$:
>>> $b_j := 0$
>>> $\mathit{active}[j] := \mathit{True}$
>>> $j := j + 1$
>>
>> $f[0] := 0$ (This may be redundant.)
>> **if** $j = n$: TERMINATE
>> $b_j := b_j + 1$
>> $a_j := a_j + d_j$
>> **if** $b_j = m_j - 1$: (OR EQUIVALENTLY:)
>> **if** $a_j = m_j - 1$ **or** $a_j = 0$: (alternative test)
>>> $\mathit{active}[j] := \mathit{False}$
>>> $d_j := -d_j$
>>> $f[j] := f[j+1]$
>>> $f[j+1] := j + 1$ (This may be redundant.)

If we are only interested in the Gray code and not in the counter $b_{n-1}, \ldots, b_1, b_0$, we don't need the inner loop: we can replace it by $j := f[0]$. This results in a very compact loopless algorithm for the reflected Gray code.

> $(a_{n-1}, \ldots, a_1, a_0) := (0, 0, \ldots, 0, 0)$
> for $i = 0, \ldots, n-1$: $d_i := +1$
> for $i = 0, \ldots, n$: $f[i] := i$
> Main loop:
>> VISIT $(a_{n-1}, \ldots, a_1, a_0)$
>> $j := f[0]$
>> $f[0] := 0$ (This may be redundant.)
>> **if** $j = n$: TERMINATE
>> $a_j := a_j + d_j$
>> **if** $a_j = m_j - 1$ **or** $a_j = 0$:
>>> $d_j := -d_j$
>>> $f[j] := f[j+1]$
>>> $f[j+1] := j + 1$ (This may be redundant.)

In the binary case, when all $m_j = 2$, the program can be simplified. The directions d_j are not needed, and we simply flip a bit by setting $a_j := 1 - a_j$. The test "$a_j = m_j - 1$ **or** $a_j = 0$" is always true.