PAT - Assignment 2

Mikkel Willén
bmq419

May 17, 2024

Task 1

After running the code with the initial value

$$\text{text}[10] = \{3, 3, 7, 7, 7, 7, 5, 5, 5, 0\}$$

the array code[20] looks like this:

$$\text{code}[20] = [3\ 2\ 7\ 4\ 5\ 3\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$

and the text[10] array looks like like the following:

$$\text{text}[10] = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$$

and the value of i and j are $i = 0$ and $j = 0$.

The encode iterates through the input array text, and summarizes sequences of identical values into pairs of (value, count), and saves that in the code array. It halts when it encounteres a terminator (0) in the text array. In the end the function decrements i and j, j by minus 2 at a time, and i by the indices of code$[j+1]$ until i is zero, which means it counts back the number of numbers read for each distinct number, so both i and j end up being zero.

Task 2

```
runLengthEncode(int text[], int code[], int* i, int* j) {
    while (text[&i] != 0) {
        int value = text[&i];
        int numberOfOccurences = 0;
        while (text[&i] == value) {
            numberOfOccurences++;
            &i++;
        }
        code[&j++] = value;
        code[&j++] = numberOfOccurences;
    }
    code[&j] = 0;
    &i = 0;
    &j = 0;
}
```

The main difference is, that the Janus program is reversible in every computation step, which means that the state before and after the execution of the program can be derived from each other, where the steps in a C-like language are not designed to be reversible. Also, the last loop in the janus program, which decrements i and j is replaced with just setting both numbers to zero, which has the same effect.

Task 3

```
procedure decode(int text[], int arc[])
    local int i = 0
    local int j = 0
    from i = 0 do
        i += arc[j + 1]
        j += 2
    until arc[j] = 0
    from text[i] = 0 loop
        j -= 2
        from arc[j] != text[i] do
            i -= 1
            arc[j + 1] -= 1
        loop
            text[i] += arc[j]
        until arc[j + 1] = 0
        text[i] += arc[j]
        arc[j] -= text[i]
    until i = 0 && j = 0
    delocal int j = 0
    delocal int i = 0
```

Task 4

$\text{Inv}(x+=e) = x-=e$
$\text{Inv}(x-=e) = x+=e$
$\text{Inv}(x <=> y) = y <=> x$
$\text{Inv}(\text{from } e_1 \text{ do } s_1 \text{ loop } s_2 \text{ until } e_2) = \text{from } e_2 \text{ do } s_2 \text{ loop } s_1 \text{ until } e_1$
Loops and conditional branches in the original program is reversed. E.g. if the original program iterates through an array by incrementing, the inverse program decrements.
Each arithmetic operation like addition and subtraction is replaced with the inverse. So addition becomes subtraction and vice versa.
If the program performs a swap operation, this swap must also occur in the reverse program at the corresponding step, thus ensuring that the state is restored to its original form.
All operations are performed in reverse. If there are multiple operations occuring in a sequence in a program, they are executed in reverse order in the reverse program.

Task 5

```
decode(int code[], int text[], int* i, int* j) {
    while (code[j] != 0) {
        int value = code[&j];
        int numberOfOccurences = code[&j + 1];
        for (int k = 0; k < numberOfOccurences; k++) {
            text[&i++] = value;
        }
        &j += 2;
    }
    text[&i] = 0;
    &i = 0;
    &j = 0;
}
```

The encoder terminates when it encounters a zero in the text array, as it means end of data. The decoder also needs to handle this. The biggest issue is memory management, since the size of the output array code in the encoder is dependent on the input data, and not just the same size. In the encode this has an upper bound, as the worst case is that all numbers in the text array are distinct, thus meaning the code array must be double the length of the text array. For the decoder, what can make a similar rule, since the number of occurrences in data pairs in the code array, could be any number, thus to circumvent this, we need to add up all number of occurrences in the code array first, to then be able to allocate sufficient space.

Task 6

a)

```
procedure fall
  from t = 0 loop
    v += 10
    h -= v
    h += 5
    t += 1
  until t = tend
```

b)

store	t	v	h	tend
init	0	0	176	3
from t = 0 loop	0	0	176	3
v += 10	0	10	176	3
h -= v	0	10	166	3
h += 5	0	10	171	3
t += 1	1	10	171	3
until t = tend	1	10	171	3
v += 10	1	20	171	3
h -= v	1	20	151	3
h += 5	1	20	156	3
t += 1	2	20	156	3

```
until t = tend              2   20   156   3
    v += 10                 2   30   156   3
    h -= v                  2   30   126   3
    h += 5                  2   30   131   3
    t += 1                  3   30   131   3
until t = tend              3   30   131   3
```

Task 7

```
store                       t   v    h   tend
init                        4   40   0    4
from t = end loop           4   40   0    4
    t -= 1                  3   40   0    4
    h -= 5                  3   40   -5   4
    h += v                  3   40   35   4
    v -= 10                 3   30   35   4
until t = 0                 3   30   35   4
    t -= 1                  2   30   35   4
    h -= 5                  2   30   30   4
    h += v                  2   30   60   4
    v -= 10                 2   20   60   4
until t = 0                 2   20   60   4
    t -= 1                  1   20   60   4
    h -= 5                  1   20   55   4
    h += v                  1   20   75   4
    v -= 10                 1   10   75   4
until t = 0                 1   10   75   4
    t -= 1                  0   10   75   4
    h -= 5                  0   10   70   4
    h += v                  0   10   80   4
    v -= 10                 0   0    80   4
until t = 0                 0   0    80   4
```

So the tower is 80 meters high. We could also run the program in reverse, but since we just need Δh, it does not really matter.