CSCI 230 -- Homework 2

Nero Li

Chapter 11

- R-11.8

We go with a n-times loop. For each loop:

1. Pop out one element from each sequence, compare them to find bigger element “a” and smaller element “b”.
2. If “a == b”, choose one and push that element into set if the top and bottom of the set is the not the same as that element.
3. Check the result set’s top “top” and bottom “bot”:
    a. If “a = top”, waived that “a”, else push the element “a” into top of the set.
    b. If “b = bot”, waived that “b”, else push the element “b” into bottom of the set.

This loop will go n-times to pop all the elements from both A and B, that means out running time will become O(n).

- R-11.23

Algorithm pathCompression(Start, End):

Input: The start iterator and the end iterator

Output: Modified tree-based partition union/find structure

```
Locator *a = find(Start)
For itr from Start + 1 to End:
    Locator *b = find(itr)
        If a != b:
            If a->size > b->size:
                b->parent = a
                a->size += b->size
            Else:
                a->parent = b
                b->size += a->size
```

- R-11.25

Algorithm quickSelect(S, left, right, k):

    Input: Sequence S, the left index and the right index that we are looking for inside the sequence

    Output: The kth smallest element of S

    If left == right:

        return S[left]

    pivot = left

    If S[k] == S[pivot]

        return S[k]

    Else if S[k] < S[pivot]

        return quickSelect(S, left, pivot – 1, k)

    Else

        return quickSelect(S, pivot + 1, right, k)

- C-11.3

Algorithm computeAxorB(A, B):

    Input: Two sequences A and B

    Output: Sequence Ans

    While A is not empty or B is not empty:

        If A is empty:

            Ans.push(B.pop())

        Else if B is empty:

            Ans.push(A.pop())

        Else:

            If A .end() == B.end():

                A.pop()

            Else:

                If A.end() < B.end():

                    Ans.push(A.pop())

                Else:

                    Ans.push(B.pop())

- R-12.3

Prefix of P:

1. a
2. aa
3. aaa
4. ~~aaab~~
5. ~~aaabb~~
6. ~~aaabba~~
7. ~~aaabbaa~~
8. aaabbaaa

Suffix of P:

1. a
2. aa
3. aaa
4. ~~baaa~~
5. ~~bbaaa~~
6. ~~abbaaa~~
7. ~~aabbaaa~~
8. aaabbaaa

Hence, the couple 1, 2, 3 and 8 characters of prefix or suffix are completely the same.

- R-12.4

T = aaabaadaabaaa

P = aabaaa

Loop 1: aaabaa != aabaaa, ++i

Loop 2: aabaad != aabaaa, ++i

Loop 3: abaada != aabaaa, ++i

Loop 4: baadaa != aabaaa, ++i

Loop 5: aadaab != aabaaa, ++i

Loop 6: adaaba != aabaaa, ++i

Loop 7: daabaa != aabaaa, ++i

Loop 8: aabaaa == aabaaa, return i = 7

- R-12.5

T = aaabaadaabaaa

P = aabaaa

Loop 1: i = 5, T[5] == P[5], T[4] == P[4], T[3] != P[3], ++i

Loop 2: i = 6, T[6] != P[5], i += P.size()

Loop 3: i = 12, T[12 – 7] == P[5 – 0], return i - 5 = 7

- R-12.6

T = aaabaadaabaaa

P = aabaaa

Failure function:

j	0	1	2	3	4	5
P[j]	a	a	b	a	a	a
f(j)	0	1	0	1	2	2

Searching:

Loop 1:

a	a	a	b	a	a	d	a	a	b	a	a	a
a	a	b	a	a	a							

Collision at P[2], f[1] = 1

Loop 2:

a	a	a	b	a	a	d	a	a	b	a	a	a
	a	a	b	a	a	a						

Collision at P[5], f[4] = 2

Loop 3:

a	a	a	b	a	a	d	a	a	b	a	a	a
				a	a	b	a	a	a			

Collision at P[2], f[1] = 1

Loop 4:

a	a	a	b	a	a	d	a	a	b	a	a	a
					a	a	b	a	a	a		

Collision at P[1], f[0] = 0

Loop 5:

a	a	a	b	a	a	d	a	a	b	a	a	a
						a	a	b	a	a	a	

Collision at P[0], move forward 1

Loop 6:

a	a	a	b	a	a	d	a	a	b	a	a	a
							a	a	b	a	a	a

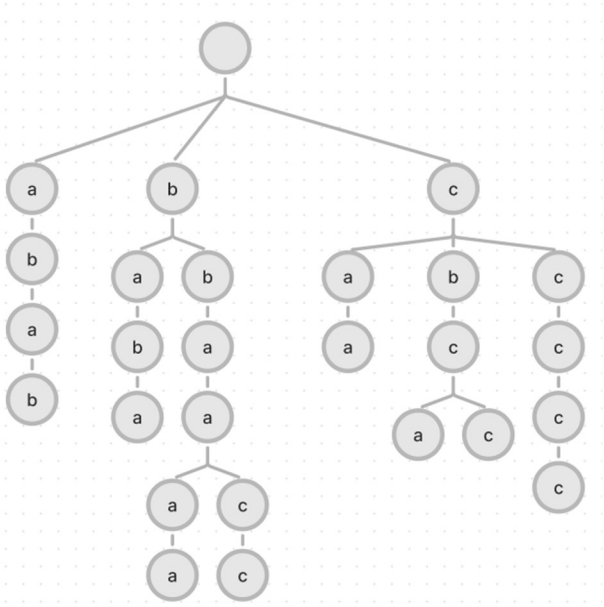
Match, return i = 7

- R-12.9

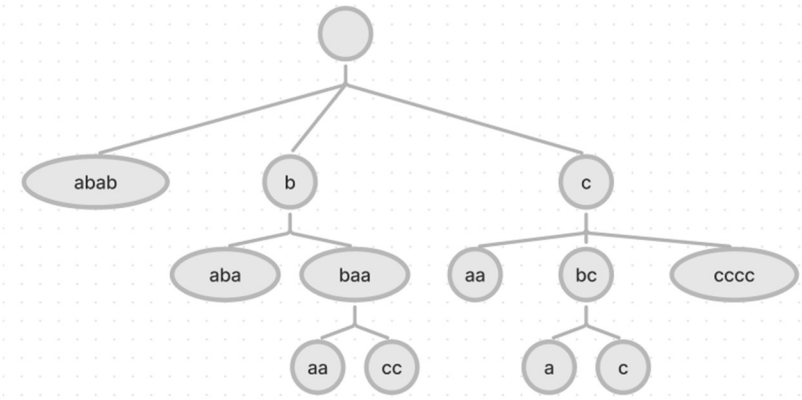
P = cgtacgttcgtac

j	0	1	2	3	4	5	6	7	8	9	10	11	12
P[j]	c	g	t	a	c	g	t	t	c	g	t	a	c
f(j)	0	0	0	0	1	2	3	0	1	2	3	4	5

- R-12.10



- R-12.11

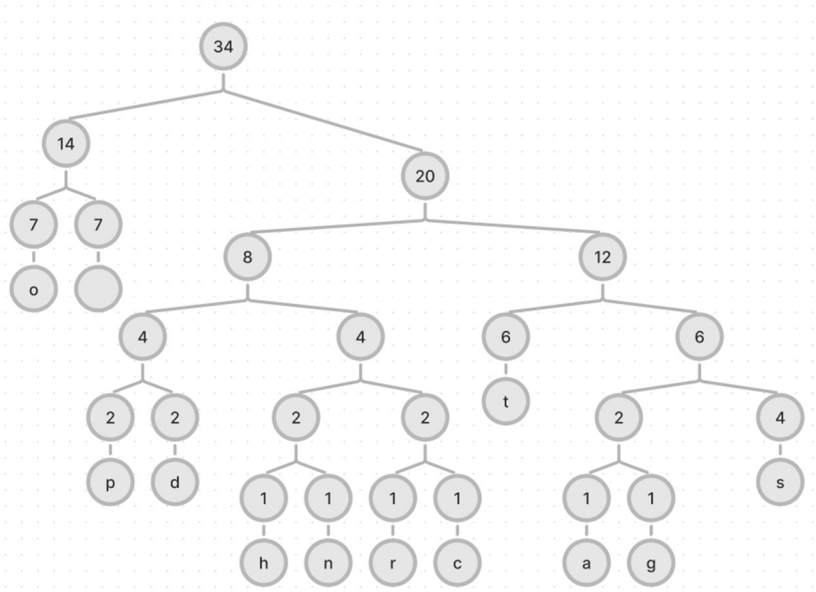


- R-12.13

“cgtacg” is the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of this string.

- R-12.14

	a	c	d	g	h	n	o	p	r	s	t
7	1	1	2	1	1	1	7	2	1	4	6



- C-12.1

Algorithm longestTrip():

While Anatjari didn’t reach the destination:

If distance to the watering holes D < K:

Find the hole that has the closest D compared with K

Set a checkpoint to that hole

Else:

Return impossible to reach the destination

For the reason why my algorithm is correct, we always get the water when our bottle is close to empty, that means we made the best efficiency for using the water inside bottle each time. Hence, we don’t always need to refill our bottle unless we reach the limit on using that.

- C-12.3

If we have **{25, 10, 1}** and our requirement is **30**, the greedy should give us the result with a 25 coin and five 1 coins. However, the correct result should be three 10 coins.