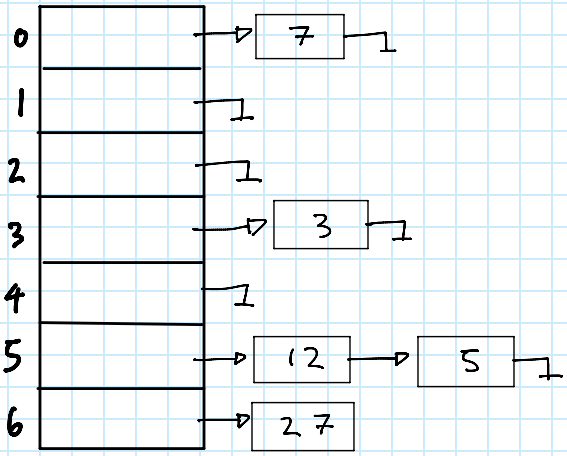
# Assignment 3

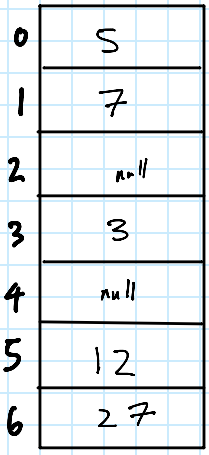
Tuesday, October 25, 2022 7:42 PM

1.



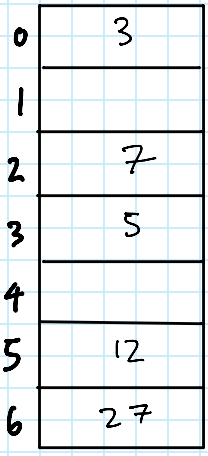

$h(12) = 12 \bmod 7 = 5$
$h(27) = 27 \bmod 7 = 6$
$h(5) = 5 \bmod 7 = 5$
$h(3) = 3 \bmod 7 = 3$
$h(7) = 7 \bmod 7 = 0$

2.




$h(12) = 12 \bmod 7 = 5$
$h(27) = 27 \bmod 7 = 6$
$h(5) = 5 \bmod 7 = 5$
$h(3) = 3 \bmod 7 = 3$
$h(7) = 7 \bmod 7 = 0$

3.




$h(12) = 12 \bmod 7 = 5$
$h(27) = 27 \bmod 7 = 6$
$h(5) = 5 \bmod 7 = 5$
$h(3) = 3 \bmod 7 = 3$
$h(7) = 7 \bmod 7 = 0$

$h'(5) = 5 - (5 \bmod 5) = 5$
$h'(3) = 5 - (3 \bmod 5) = 2$
$h'(7) = 5 - (7 \bmod 5) = 3$

4. $f(1) = C_0$

$f(n) = f(n-1) + C_1 n + C_2$, for $n > 0$.
$f(n) = f(n-2) + C_1(n-1) + C_2 + C_2$
$f(n) = f(n-3) + C_1(n-2) + C_1(n-1) + C_2 + C_2 + C_2$
$f(n) = f(n-4) + C_1(n-3) + C_1(n-2) + C_1(n-1) + C_2 + C_2 + C_2 + C_2$

$f(n-1) = f(n-2) + C_1(n-1) + C_2$
$f(n-2) = f(n-3) + C_1(n-2) + C_2$
$f(n-3) = f(n-4) + C_1(n-3) + C$

$f(n) = f(n-2) + C_1(n-1) + C_2 + C_2$
$f(n) = f(n-3) + C_1(n-2) + C_1(n-1) + C_2 + C_2 + C_2$
$f(n) = f(n-4) + C_1(n-3) + C_1(n-2) + C_1(n-1) + C_2 + C_2 + C_2 + C_2$

$f(n-2) = f(n-3) + C_1(n-2) + C_2$
$f(n-3) = f(n-4) + C_1(n-3) + C$

$f(n) = f(n-k) + C_1(n-(k-1)) + C_1(n-(k-2)) + \ldots + C_1 n + kC_2$

Assume: $n-k=0$

$n=k$

$f(n) = f(n-n) + C_1(n-n+1) + C_1(n-n+2) + \ldots + C_1 n + nC_2$
$= f(0) + C_1 + 2C_1 + 3C_1 + \ldots + nC_1 + nC_2$
$= f(0) + C_1(1 + 2 + 3 + \ldots + (n-1)(n)) + nC_2$

$f(n) = C_0 + C_1\left(\frac{n(n+1)}{2}\right) + C_2 n$

Sum of first n natural numbers $= \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$f(n) = C_0 + C_1\left(\frac{n^2+n}{2}\right) + C_2 n$

∴ the time complexity of $f(n)$ is $O(n^2)$.

Steps:

1. I first repeatedly substituted the recurrence equation until I noticed a pattern.
2. I then rewrote the equation for all values of $n > 0$.
3. I then set $n-k$ equal to zero because:
   $f(1) = f(1-1) + C_1 + C_2$
   $= f(0) + C_1 + C_2$
   $= C_0$
4. After doing this, I saw that the equation was summing the first n natural numbers so I substituted it for the given formula
5. Lastly I used this last equation to calculate the time complexity of the algorithm.

5. a)

```
Algorithm: totalLeaves(r, level)
In: root of a (sub) tree, integer value level
Out: sum of all levels of all the leaves of the tree

        if r.isLeaf(), then return level {
        } else {
                for each child c of r do {
                        sum += totalLeaves(c, ++level)
                }
                return sum
        }
```

b)

```
Algorithm: totalLeaves(r, level)
In: root of a (sub) tree, integer value level
Out: sum of all levels of all the leaves of the tree

        if r.isLeaf(), then return level {
        } else {
                for each child c of r do {
                        sum += totalLeaves(c, ++level)
                }
                return sum
        }
```

$C_1$ $C_2$ $C_3$

# iterations = # children = degree(r)

① Complexity without recursive calls
$C_1$ if r is a leaf
$C_2 + C_3$ degree(r)

② # of calls

$n$ calls (one call per node)

③ $f(n) = \sum_{\text{leaves}} c_1 + \sum_{\text{internal}} c_2 + c_3$ degree(r)

$f(n) = c_1 \times$ # leaves $+ c_2 \times$ # internal $+ c_3 \times$ # nodes

$\therefore$ $f(n)$ is $O(\underbrace{\text{\# leaves} + \text{\# internal}}_{n} + n) = O(2n) = O(n)$

Steps I used:

1. I figured out which sections of the algorithm were constant & how many calls would be made.
2. I developed a rough function to represent the recursive function in terms of the number of nodes.
3. I simplified it & removed the constants.
4. Used the degree of the resulting function to compute time complexity

6.

**Algorithm** algo$(A, B, n)$
**In:** Arrays $A$ and $B$ of size $n$

$c_1$
$i \leftarrow 0$
$j \leftarrow 0$
**while** $j < n$ **do** {

$c_2$ (× $n$)
$B[i] \leftarrow A[j]$
**if** $A[i] = B[j]$ **then** $i \leftarrow n - i$
**else** $i \leftarrow i + 1$
$j \leftarrow i + 1$

}

$f(n) = c_1 + n c_2$

$\therefore$ $f(n)$ is $O(n)$

Steps
1. Designate constant time operations
2. Write function to represent time complexity of algorithm.
3. Use the degree of the function to calculate the time complexity of the function