QUESTION 2

```
> ntab :: (Int -> a) -> (Int -> a)
> ntab f = ntf
>         where ntf n = tabf !! n
>
>               tabf = [f n | n <- [0..]]
```

(a) Assuming lazy evaluation, the ntab function, given a function f, tabulates all the values of f n, for given numbers n and it is useful for repeated calls of f on the same values. The time-efficiency of ntab f n is O(n) because of the (!!) function, but after the first call of a specific n for ntab f n, the value of f n need not be calculated again. For ntab there is also important to specify that it needs O(n) memory.

The definition of ntab could not have been written ntab f n = ... because it would lose the property of tabulating the values of n and on subsequent calls over the same n, the complexity will also include the calculation of f.

f would be preferable to ntab f when we only need to calculate f n once for any n and for small n if f is easy to calculate and fast.

(b)

```
> ztab :: (Int -> a) -> (Int -> a)
> ztab f = ztf
>         where ztf n = if (n >= 0) then tabf !! n else ztabf !! (-n-1)
>
>               tabf = [f n | n <- [0..]]
>
>               ztabf = [f n | n <- [-1,-2,...]]
```

(c)

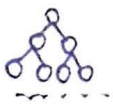
```
> data Tree a = Fork (Tree a) (Tree a) | Leaf a

> tab :: (Int -> a) -> Int -> Int -> (Int -> a)
> tab f low high = ftab
>                  where ftab n = extract tree low high n
>
>                        tree = make_tree f low high
```

```
> make_tree :: (Int -> a) -> Int -> Int -> Tree a
> make_tree f low high
>     | low == high = Leaf (f low)
>     | otherwise = Fork (make_tree f low mid) (make_tree f (mid+1) high)
>       where mid = (low + high) `div` 2

> extract :: Tree a -> Int -> Int -> Int -> a
> extract (Leaf x) _ _ _ = x
> extract (Fork left right) low high n
>     | n <= mid = extract left low mid n
>     | otherwise = extract right (mid+1) high n
>       where mid = (low + high) `div` 2
```

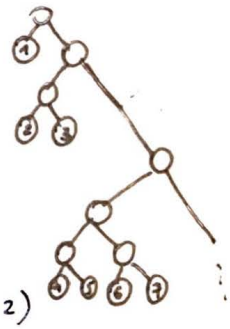
We create a tree in which all the values of f in the interval [a,b) are, so the search for a value needs log(b-a) steps



(d)

```
> tab_inf :: (Int -> a) -> Int -> (Int -> a)
> tab_inf f low = ftab
>                 where ftab n = extract_inf tree low 1 n
>                       tree = make_tree_inf f low 1

> make_tree_inf :: (Int -> a) -> Int -> Int -> Tree a
> make_tree_inf f low x = Fork left_son right_son
>                         where left_son = make_tree f low (low + x - 1)
>                               right_son = make_tree_inf f (low + x) (x * 2)
```



↑ the infinite tree

```
> extract_inf :: (Tree a) -> Int -> Int -> Int -> a
> extract_inf (Fork left_son right_son) low x n
>     | n < low + x = extract left_son low (low + x - 1) n
>     | otherwise = extract_inf right_son (low + x) (x*2) n
```

The time complexity for tab_inf f n is O(log n) as we need log n steps to find in which region of type $[2^K, 2^{K+1}-1)$ n is and another log n steps to find the leaf containing f n.