# CGJ008

## Big-O Quiz

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **f(n)** |  | **O()** |
| A | 2n+1 | 2n | O |
| B | 2n | 2n | O |
| C |  |  | O |
| D |  |  | O |
| E | 1 | 1 | O |
| F |  |  | O |
| G | 2n+2 | 2n | O |
| H |  |  | O |
| I |  | 2n | O |
| J |  |  | O |
| K |  |  | O() |
| L |  |  | O |
| M |  |  | O |
| N |  |  | O |
| O |  |  | O |
| P |  |  | O |
| Q |  |  | O |
| R |  |  | O |
| S |  |  | O |
| T |  |  | O |
| U |  |  | O |
| V |  |  | O |
| W |  |  | O |
| X |  |  | O |
| Y |  |  | O |
| Z |  |  | O |

Note: In the answer to functionM f(n), the first part is valid for n=0 and up. The alternative is valid for n=1 and up.

## Union Find

## c)

In this implementation (using an integer array, id[]), it is not possible. We need to set the lesser root of p and q as root of the combined set, to keep track of the oldest account. Which excludes the option to always choose the tallest tree as root.

If we were to solve the problem using node objects in a tree structure, one could perform weighted union, swapping the values of the root nodes if the lesser root is the top of the shorter tree.

### d)

### Scheme

Assuming we are allowed to make “stupid” requests, like union(p, q) where p = q.

Calling find(p) while p is root of itself, costs **1** array access

Calling find(p) while p is not root of itself costs **2h-1**, where h is the height of the tree (counting height as number of elements in the tree)

Calling union(p, q) where p is not a root, but q is root of itself (single element), costs 2h-1 for find(p) and 1 for find(q), and another for the actual union, resulting in **2h+1**

Calling union(p, q) where p = q and they are an element at the bottom of a worst-case tree (which we will create) costs find(p) = 2h-1, find(q) = 2h-1, and 1 for the union action. Result: 2(2h-1)+1 = **4h-1**

Now, the scheme would be to make a worst-case tree chaining elements, making the height equal to the number of elements. Example: For the 10 elements {0 to 9}, start with

|  |  |
| --- | --- |
| Union(0,1) | 1 <- 0 |
| Union(1,2) | 2 <- 1 <- 0 |
| Union(2,3) | 3 <- 2 <- 1 <- 0 |
| …until union(8,9) ? | 9 <- 8 <- … <- 1 <- 0 |

Now, at some point we would get a higher total number of array accesses if we stopped chaining elements, but rather called union(p, q) where p = q and it is the bottom of the tree, because this costs **4h-1**.

Example: after creating a worst-case tree of the elements 0 to 8

Union(8,9) = 2h+1 = **19**

Union(8,8) = 4h-1 = **35**

## Analysis

Assuming (to begin with) we want to make calls to union(p, q), where p = 0, q=1 and both increase by 1 each step (creating a worst-case tree). Then calls to union(p, q) where p = q = x.

This can be expressed as:

Using i as height for the first sum, because it increases with elements added, x as height in the second sum as we are no longer increasing the length of the tree.

The goal is to find x as a function of m.

The first sum can be written as

The second sum can be written as

Analysis of the following function shows that for any valid is at its maximum when

## Conclusion

Chaining elements in a worst-case tree until element creates the optimal situation to make the maximum number of array accesses with the remaining actions, .

Now considering the case where , the explanation above will be valid until . We can’t add any more elements to the tree when we have added all n elements. Hence, for , we must add elements to the tree until n, and then for the remaining actions, call union(p, q) where p = q = the bottom of the tree.

Tilde approximation:

for

for

Big-O notation

for

for