CS 401: Homework 3

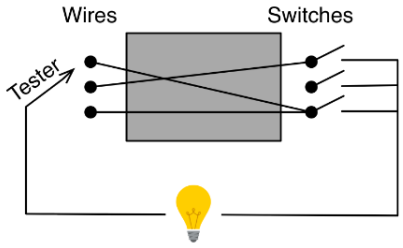
Due on February 26, 2024 at 11:59pm

Professor Sidiropoulos 9:30am

Ryan Magdaleno
rmagd2@uic.edu

Problem 1: Wires and switches.

You are given a black box that connects n wires to n switches. Each wire is connected to exactly one switch. Each switch is connected to zero or more wires. Your goal is to determine which wire is connected to each switch, without opening the box. You can perform a sequence of the following two operations: (1) Turn some switch j on or off. (2) Test some wire i. When testing a wire, if the switch it is connected to is on, then a lamp lights up.



It is easy to discover all the connections using $O(n^2)$ operations, as follows.

```
Initially, turn all the switches off (n operations)
For i = 1 to n
    For j = 1 to n
        Turn switch j on
        Test wire i
        If the lamp lights up
            we know that the wire i is connected to switch j
        Turn switch j off
```

Give an algorithm which discovers all the connections using $O(n \log n)$ operations.

Problem 1 continued on next page...

Solution
High Level Algorithm Explanation

I will use a divide and conquer technique to get a $O(n \log n)$ time complexity for my algorithm. Assuming we can have multiple switches on at once when testing a single wire, my algorithm is as follows:

1. Turn off all switches initially. (n operations), also return if $n = 0$.
2. Initialize some hashmap/map that takes in a wire's id and returns the switch it is connected to, initially it is empty, this is the final result to look through after the algorithm executes. (1 operation)
3. Initialize an array container that stores all the wires from 0 to $n - 1$. (n operations)
4. Randomly assign wires to switches, switches can have, 0, 1, or multiple wires connected to it, but wires can only connect to one (n operations).
5. Do the following until only 1 switch is left to check for remaining wires in that function call: ($\log n$)
   (a) Turn on half the switches in the current range (initially $start = 0$, $end = n - 1$) (n/c operations, c is the current division step, grows by $*2$ for every recursive depth layer, 2, 4, 8, 16, ...)
   (b) Intialize two new arrays that will store whether a wire is connected to the half of the wires that are turned on and the other array for not turned on wires.
   (c) For every wire in function's given array, check if it's in the on group or the off group (test wire against the half of the turned on switches), and place in corresponding array. (n operations)
   (d) Turn off the switches that were turned on in step (5a) (n/c operations)
   (e) Do two function calls on this same function, divide the range of switches to check for the wires, the on group wires must be in some wire between the current $start$ and mid, so for the on group do a recursive call on the new range of $start$ – mid, where $mid = \frac{(start+end)}{2}$, do the same for the off group function call where the new range of switches is $mid + 1$ – end.
6. The remaining switch must be connected to all the wires in the given array, therefore in our hashmap indicate that every wire in the given array is connected to this remaining switch, then return to continue for other recursive calls. (n operations)
7. When the algorithm returns to main the map will be populated with the correct wire/switch pairings.

Note: If for some reason I am not allowed to move a wire object, then my algorithm would store the wire's index in the onGroup and offGroup arrays instead of the wire object itself.

Algorithm Pseudo-Code

```
1: function makeConnections(connections, wires, switches, start, end)
2:     if start ≡ end then
3:         for each wireID in wires do
4:             connections[wireID]← switches[start]
5:         end for
6:         return
7:     end if
8:
9:     mid ← (start + end)/2
10:    onGroup ← empty array
11:    offGroup ← empty array
12:
13:    for i ← start, i to mid do
14:        Turn switch i on
15:    end for
16:
17:    for each wire in wires do
18:        Test wire
19:        if the lamp turns on then
20:            onGroup.push_back(wire)
21:        end if
22:        if the lamp does not turn on then
23:            offGroup.push_back(wire)
24:        end if
25:    end for
26:
27:    for i ← start, i to mid do
28:        Turn switch i off
29:    end for
30:
31:    makeConnections(connections, onGroup, switches, start, mid)
32:    makeConnections(connections, offGroup, switches, mid + 1, end)
33: end function
34:
35: function main(n)
36:    if n ≤ 0 then
37:        return
38:    end if
39:    connections ← dynamic map<wireID, switchID>
40:    wires ← array of size n, contains wireIDs from 0 to n − 1
41:    switches ← array of size n, contains switchIDs from 0 to n − 1
42:    makeConnections(connections, wires, switches, 0, n − 1)
43:    // connections should now have a connection from every wire to a switch.
44: end function
```

Time Complexity Justification

Let's analyze this algorithm to see if it is $O(n \log n)$.

Initializing the wire and switch containers, along with randomizing the wires to the switches is a linear operation so this is $O(3n) = O(n)$.

At each recursive call we do the following: turn on half the switches in the current function call's range, we turn them off beofre the function returns, this is a linear amount of operations against n, $O(n)$.

We also test every wire that is in the function's wire array against the half turned on switches, this is $O(n)$ operations.

Now as for where the $\log n$ term comes from, at each recursive call we are essentially performing a binary search iteration where if some *wire* is tested against the half turned on switches, it gets placed in the on or off group depending on the state of the bulb after testing the *wire*. We then do two function calls for the on and off group and split the range in two, for the on group those wires must be in the front half of switches considering the bulb turned on, for the off group, they must be in the latter half. At each recursive step, we divide the range of switches to check in two every step, until there is only one switch left. One switch left indicates that the wires still in this function call's wire array must be connected to the remaining switch. So at every step we divide the range in two, essentially making a smaller and smaller subproblem until we are left with a single switch (base case). Therefore the recursive depth grows in $\log_2(n) = \log(n)$.

During the base case we also make the map pairs from the wires array, this is a linear operation, $O(n)$.

Combining everything together we are left with the following:

$$
\begin{aligned}
T(n) &= O(3n + (n + n + n) \cdot \log_2(n)) \\
T(n) &= O(3n + 3n \cdot \log_2(n)) \\
T(n) &= O(3n \cdot \log_2(n)) \\
T(n) &= \boxed{O(n \cdot \log(n))}
\end{aligned}
$$

Real Code Implementation

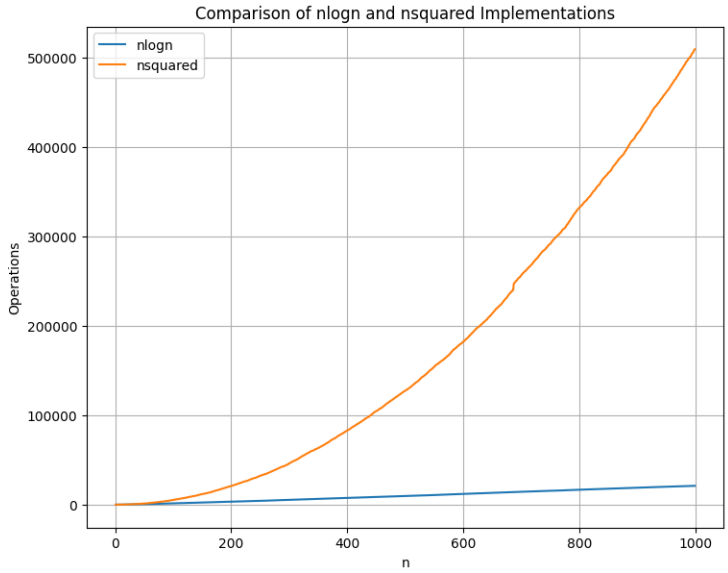
```cpp
/* P1.cpp */
/* g++ -std=c++23 -O2 -Wall P1.cpp -o P1.exe */
#include <map>       // Use unordered_map for faster operations if you'd like.
#include <iostream>  // Console IO
#include <vector>    // Array-like containers.
#include <algorithm> // for std::shuffle
#include <random>    // for std::default_random_engine and std::uniform_int_distribution
#include <ctime>     // for std::time

typedef unsigned long long ull;
typedef std::map<ull, ull> Connections;
struct Wire   { ull wireID, switchID;    };
struct Switch { ull switchID; bool isOn; };
```

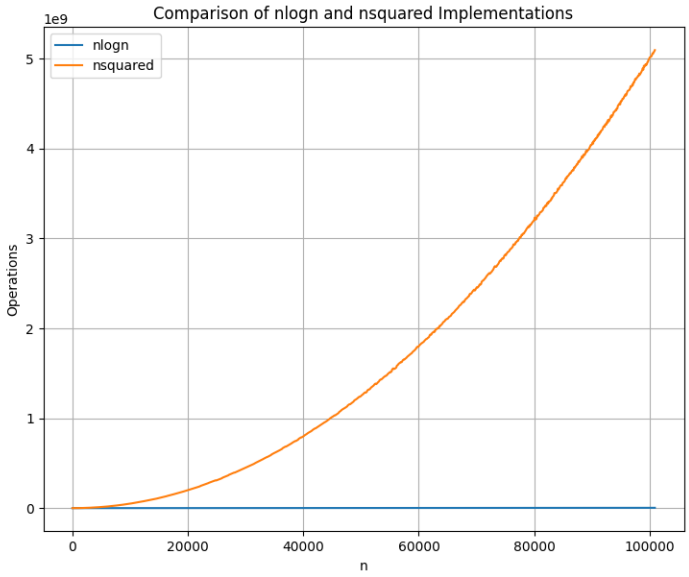
```
void makeConnections(
    Connections& c,          // Resultant map from Wire IDs to correct switch IDs.
    std::vector<Wire>& w,    // Container for current call's wires.
    std::vector<Switch>& s,  // Container for all switches.
    ull start,               // Start index for range.
    ull end)                 // End index for range.
{
    // Base case: only one switch, n step: add connection to resultant container.
    if (start == end) {
        for (const Wire& wire : w) {
            c[wire.wireID] = start;
        }
        return;
    }
    std::vector<Wire> onGroup, offGroup;
    ull mid = (start + end) / 2;

    // n/c step: Turn on front half of switches in current step.
    // c is current division step constant, depth 1 = 2, depth 2 = 4, ...
    for (ull i = start; i <= mid; ++i) {
        s[i].isOn = true; // Redundant, but this is part of the algorithm.
    }

    // n step: Check each wire if bulb turns on.
    for (const Wire& wire : w) {
        // This is how I'm checking if a bulb is on, in the real
        // diagram it's simply a black box bulb check, after turning the switches on.
        // Simply check if each wire's "solution" was in the range of bulbs that were on.
        if (wire.switchID >= start && wire.switchID <= mid) {
            onGroup.push_back(wire);
        } else {
            offGroup.push_back(wire);
        }
    }
    // n/c step: Turn off front half of switches in current step.
    // c is current division step, depth 1 = 2, depth 2 = 4, ...
    for (ull i = start; i <= mid; ++i) {
        s[i].isOn = false; // Redundant, but this is part of the algorithm.
    }

    // Divide range in half for onGroup wires:
    makeConnections(c, onGroup, s, start, mid);
    // Divide range in half for offGroup wires:
    makeConnections(c, offGroup, s, mid + 1, end);
}
```

```cpp
// Randomly assign wires to switches (0/1 to 1, many wires to 1)
void randomAssign(std::vector<Wire>& w, std::vector<Switch>& s, ull n)
{
    std::uniform_int_distribution<ull> distribution(0, s.size() - 1);
    std::default_random_engine generator(std::time(0));
    for (ull i = 0; i < n; ++i)
    {
        w[i] = {i, distribution(generator)};  // wireID, switchID
        s[i] = {i, false};                    // switchID, isOn
    }

    for (const auto& wire : w) {
        std::cout <<
           "Wire " << wire.wireID << " connected to switch " << wire.switchID << '\n';
    }
}


int main()
{
    ull n = 100;
    if (n <= 0) { return 0; }
    auto wires       = std::vector<Wire>(n);
    auto switches    = std::vector<Switch>(n);
    auto connections = Connections();
    randomAssign(wires, switches, n);
    makeConnections(connections, wires, switches, 0, n - 1);

    // Validate:
    std::cout << std::endl;
    for (const auto& [wireID, switchID] : connections) {
        std::cout <<
            "Wire " << wireID << " connected to switch " << switchID << '\n';
        if (wires[wireID].switchID != switchID) {
            std::cerr << "Mismatch: " << wires[wireID].switchID << ' ' << switchID;
            return 1;
        }
    }
    std::cout << "All good!" << "\n\n";
    return 0;
}
```

Testing the Algorithm

I ran my $n \log n$ algorithm against the given n^2 algorithm, using code instrumentation I acquired the approximate operation count for both, the input n was from 1–1000, then 1–100000, incrementing by 1000.





Problem 2: Fraudulent accounts.

A large online social networking company has recently discovered that it may have a problem with a number of accounts that are spreading false information through the site. The company has a collection of n accounts that they've been monitoring over the last several months, suspecting them of being used for such activity. Each account has the following data associated with it: a user id, some possibly fraudulent information (such as the name, etc.), and a unique encrypted number generated by the site when the account was created. A user can have more than one account on the site. The company will not disclose the identity of the account holders directly without a subpoena, but will tell if two accounts belong to the same user.

The government agency's question is the following: among the collection of n accounts, is there a set of more than $n/2$ of them that are all associated with the same user? Assume that the only feasible operation the government investigators can do with the accounts is to choose two of them and ask the company if they belong to the same user (and the company people get more and more reluctant to answer each time). Show how to decide the answer to the agency's question with only $O(n \log n)$ requests to the social networking company.

Solution
High Level Algorithm Explanation

I will use a divide and conquer technique to get a $O(n \log n)$ time complexity for my algorithm.
My algorithm is as follows:

1. Do the following until we are down to one account to check.
   (a) Split the input accounts into halves.
   (b) Do a recursive call on each halve to effectively split input accounts down logarithmically.
2. After the divide phase we should now only be at a single account, return from the base case call.
3. Do the following until we exit out of the initial function call.
   (a) Compare the majority account from the returned left call with the right call's returned majority account, if they are the same user then add their counts together and return this new majority account. If they are different users then return the majority share account and discard the minority share account, if they are not the same user but have the same share, return the left, then right[1].
4. Validate the two returned account shares and check across the entire account base for their true share.
5. Make the final account the one with the higher share of the two.
6. Check if this final account indeed has a $> n/2$ share across the accounts, if so then it satisfies the government agency's question that there is a user with a majority share.

Note: This assumes that we are allowed to associate a count with a set of accounts.
[1] What I mean by this is we will run two slightly different algorithms, one where if the userIDs are different, but same count, we return the left half in one main call, and the right halve in the other main function call.
Side-Note: I call count and share interchangeably throughout my explanations and code.

Algorithm Pseudo-Code

```
1: function majorityFind(Accounts, leftHalve)
2:     if len(Accounts) ≡ 1 then
3:         return (Accounts[0].userID, 0)
4:     end if
5:     mid ← len(Accounts)/2
6:     left = empty container, right = empty container
7:     Split Accounts in half, put left half in left and the right half in right.
8:     leftMajority ← majorityFind(left, leftHalve)
9:     rightMajority ← majorityFind(right, leftHalve)
10:    if leftMajority is the same user as rightMajority then
11:        return merge share into one account.
12:    end if
13:    if leftMajority has a higher count than rightMajority then
14:        return leftMajority
15:    end if
16:    if rightMajority has a higher count than leftMajority then
17:        return rightMajority
18:    end if
19:    return leftHalve ? leftMajority : rightMajority
20: end function
21:
22: function main(n)
23:    if n ≤ 0 then
24:        return
25:    end if
26:    Accounts ← dynamic array of type Account
27:    FinalLeft ←majorityFind(Accounts, true)
28:    FinalRight ←majorityFind(Accounts, false)
29:
30:    // Validate:
31:    countLeft ← 0, countRight ← 0
32:    for i ← 0; i to n do
33:        if Accounts[i] is the same user as FinalLeft then
34:            countLeft ← i + 1
35:        end if
36:        if Accounts[i] is the same user as FinalRight then
37:            countRight ← i + 1
38:        end if
39:    end for
40:    finalAccount ← countLeft > countRight ? FinalLeft : FinalRight
41:    finalCount ← countLeft > countRight ? countLeft : countRight
42:    return FinalCount > n/2
43: end function
```

Time Complexity Justification

Let's analyze this algorithm to see if it is $O(n \log n)$.

My recursive function *majorityFind* splits the passed input array into two halves namely $leftHalve$ and $rightHalve$, $O(n)$ operations. We then recursivly call *majorityFind* on both split halves until we are down to an array size of one, which also means one Account is left. In our base case we return this sole account upwards with a count of one. *majorityFind* then combines the sum of the counts if the user is the same, if different it returns the max count Account between the two, in total there is a single check to see if they are the same user.

Because we are dividing our input array for each recursive step this results in the height of the recursion call stack being $\log_2(n)$.

I run this algorithm twice to account for both returned halves when their ID is different but their share is the same. This results in a constant 2 being multipled against our *majorityFind*'s $T(n)$.

I also do a linear check on the returned Accounts against the entire Account container to check the true total for the returned account, and take the max of the two returned halves, this results in $O(2n)$ operations.

Combining everything together we are left with the following:

$$
\begin{aligned}
T(n) &= O(2 \cdot n \log_2(n) + 2n) \\
T(n) &= O(n \log_2(n) + 2n) \\
T(n) &= O(n \log_2(n)) \\
T(n) &= \boxed{O(n \log(n))}
\end{aligned}
$$

Real Code Implementation

```cpp
/* P2.cpp */
/* g++ -std=c++23 -O2 -Wall P2.cpp -o P2.exe */
#include <iostream>  // Console IO
#include <vector>    // Array-like containers.

typedef unsigned long long ull;
struct Account {ull userID, count; };
typedef std::vector<Account> AVector;


// I ran out of time to make a  goodrandomAssigner but
// essentially if one userID dominates, then that
// userID should be the outcome of my algorithm.
void randomAssignIDs(AVector& a, ull n)
{
    for (ull i = 0; i < n/2; ++i) { a[i] = {222, 0}; }
    for (ull i = n/2; i < n; ++i) { a[i] = {i,   0}; }
}
```

```
// Recursive function that divides input array into small sub arrays then
// returns the majority user.
Account majorityFind(
    AVector& a,     // Container for current call's accounts.
    bool leftHalf)  // Whether we return the left halve if counts are equal.
{
    // Base case: Only one account:
    if (a.size() == 1) {
        return {a[0].userID, 1};
    }
    // Split input array a into two equal halves:
    ull mid = a.size() / 2;
    AVector left = AVector(), right = AVector();

    for (ull i = 0; i < mid; ++i)        { left.push_back(a[i]);  }
    for (ull i = mid; i < a.size(); ++i) { right.push_back(a[i]); }

    // Retrieve the majority accounts from the left and right half:
    auto leftHalve = majorityFind(left, leftHalf);
    auto rightHalve = majorityFind(right, leftHalf);

    // Both halves belong to same user, combine their counts:
    if (leftHalve.userID == rightHalve.userID) {
        return {leftHalve.userID, leftHalve.count + rightHalve.count};
    }
    // If different users, return the account with the higher count:
    if (leftHalve.count > rightHalve.count) {
        return leftHalve;
    }
    else if (leftHalve.count < rightHalve.count) {
        return rightHalve;
    }
    // If different users, same count, return halve based on "leftHalf":
    return leftHalf ? leftHalve : rightHalve;
}
```

Problem 2 continued on next page...

```
int main()
{
    ull n = 300;
    if (n <= 0) { return 1; }
    auto Accounts = AVector(n);
    randomAssignIDs(Accounts, n);

    auto leftCheck  = majorityFind(Accounts, true);
    auto rightCheck = majorityFind(Accounts, false);

    ull leftCount = 0, rightCount = 0;
    for (ull i = 0; i < n; ++i) {
        if (Accounts[i].userID == leftCheck.userID) {
            leftCount++;
        }
        else if (Accounts[i].userID == rightCheck.userID) {
            rightCount++;
        }
    }
    Account final  = leftCount > rightCount ? leftCheck : rightCheck;
    ull finalCount = leftCount > rightCount ? leftCount : rightCount;

    std::cout << "\n n: " << n << ", ";
    std::cout << "userID: " << final.userID << ", Count: " << finalCount;
    if (finalCount > n/2) { std::cout <<  " > n/2" << '\n'; }
    else                  { std::cout << " <= n/2" << '\n'; }
    return 0;
}
```

Problem 2 continued on next page...

Testing the Algorithm

I ran my $n \log n$ algorithm against a n^2 implementation, using code instrumentation I acquired the approximate operation count for both, the input n was from 1–1000, then 1–100000, incrementing by 1000.