Membership sign up:

Week 4

Practice Contests Leaderboard

-

JUNIOR EXEC APPLICATIONS OPEN

Go to link below or scan QR code to apply.
The deadline is OCTOBER 31st at midnight.

https://forms.gle/8dkGw1Zezfm4BVnRA

SCAN ME

Week 3 Solutions

Card Trick

Goal:

Determine the permutation of cards such that applying the procedure in the problem description will yield the cards in order.

Solution:

Simulate the procedure in reverse using a deque. I.e. add the cards in reverse order and cycle the deck backwards

```cpp
#include <iostream>
#include <deque>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(NULL);

    int tcs, n;
    std::cin >> tcs;
    while (tcs--) {
        std::cin >> n;
        std::deque<int> deck;
        for (int i = n; i > 0; i--) {
            deck.push_front(i);
            for (int j = 0; j < i; j++) {
                deck.push_front(deck.back());
                deck.pop_back();
            }
        }
        for (int i = 0; i < n - 1; i++) {
            int card = deck.front();
            deck.pop_front();
            std::cout << card << ' ';
        }
        std::cout << deck.front() << std::endl;
    }
    return 0;
}
```

Numbers on a Tree

Goal:

Given a string of L's and R's determine what node you would arrive at in a balanced binary tree by traversing in the order of the string

Notes:

- Re-numbering the nodes by #nodes - i structures the nodes like indices of a binary heap
- Starting from the root node indices in a binary heap double + 1 when moving left and double + 2 moving right

Solution:

Start with i = 1. For every "L" double i and for every "R" double i and add one. Return #nodes - i to revert back to the correct ordering

```cpp
#include <iostream>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(NULL);

    int n;
    std::cin >> n;
    n = 2 << n;
    char c;
    int i = 0;
    while (std::cin >> c) {
        i <<= 1;
        i++;
        if (c == 'R') i++;
    }
    std::cout << n - i - 1 << std::endl;

    return 0;
}
```

Circuit Math

Goal:

Given a set of variable assignments and a boolean expression (in postfix) evaluate the expression

Solution:

Use a stack to store the expression as we see it. If we see a variable, add the assignment to the stack. If we see an operation, perform the operation on the top elements on the stack (top 1 element for negation, top 2 for + and * operations) and put the result back on the stack. When no more operations Return the top of the stack

```cpp
#include <iostream>
#include <vector>
#include <stack>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(NULL);

    int n;
    std::cin >> n;
    std::vector<bool> vals;
    vals.resize(n);
    for (int i = 0; i < n; i++) {
        char j;
        std::cin >> j;
        vals[i] = j == 'T';
    }
    std::stack<bool> stack;
    char c;
    while (std::cin >> c) {
        if (c >= 'A' && c <= 'Z') stack.push(vals[c - 'A']);
        else if (c == '-') {
            bool a = stack.top();
            stack.pop();
            stack.push(!a);
        } else {
            bool a = stack.top();
            stack.pop();
            bool b = stack.top();
            stack.pop();
            stack.push((c == '+') ? (a || b) : (a && b));
        }
    }
    std::cout << (stack.top() ? 'T' : 'F') << std::endl;
    return 0;
}
```

Chopping Wood

Goal:

Given an ordered list of nodes that were connected to leaves while dismantling the tree (knowing that the lowest leaf will be removed first) determine which leaves were removed in order

Notes:

- The error cases occur when the last node in v is not the largest node
- Nodes not in v are definitely leaf nodes in the original tree

Solution:

Consider nodes in v as “unresolved”, meaning we don’t know what node they are connected to, and nodes not in v as “resolved”. By going through the unresolved nodes and resolving them by pairing them with the lowest currently resolved node we can reconstruct the tree. Note that if an unresolved node appears multiple times in v it has to be resolved as many times as it appears before becoming fully resolved. The order at which we add resolved nodes gives us u

```cpp
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(NULL);

    int n, v;
    std::unordered_map<int, int> resolves;
    std::queue<int> unresolved;
    std::priority_queue<int> available;
    std::cin >> n;
    // Store v column and element frequency
    for (int i = 0; i < n; i++) {
        std::cin >> v;
        resolves[v]++;
        unresolved.push(v);
    }
    // Unresolvable if the last v value is not the largest node
    if (unresolved.back() != n + 1) {
        std::cout << "Error" << std::endl;
        return 0;
    }
    // Add the resolved nodes (not unresolved) to available pq
    for (int i = 1; i <= n + 1; i++)
        if (resolves.find(i) == resolves.end())
            available.push(-i);
    // Resolve each v entry and if finished resolving a value
    // add it to the available pq
    while (!unresolved.empty()) {
        v = unresolved.front();
        unresolved.pop();
        std::cout << -available.top() << std::endl;
        available.pop();
        if (!--resolves[v]) available.push(-v);
    }
    return 0;
}
```

Asymptotics and Big O

Time/Space Complexity

- **Time Complexity:** The time/number of operations an algorithm takes based on the input size
- **Space Complexity:** The amount of storage/memory used by an algorithm based on the input size

When analyzing an algorithm we often “don’t care” about the specific time complexity/runtime (often denoted T(n)) but rather how it scales as the input size grows. This helps us understand the underlying algorithm rather than a specific implementation. For example changing the implementing language or computer running the algorithm could speed up/slow down the algorithm but that doesn’t tell us about how the algorithm behaves.

Asymptotic/Big O/Big Θ Notation

Given the runtime T(n) of an algorithm we want to determine how the runtime scales with respect to the input size. To do this we switch to Asymptotic notation such as Big O or Big Θ notation.

- T(n) is said to be in the set **O(**f(n)**)** iff T(n) grows **no faster** than f(n)
  - Formally $\exists\, c > 0: \exists\, n_0 > 0: \forall\, n > n_0: c \cdot f(n) \geq T(n)$
  - T(n) is no larger than f(n) times some positive constant factor given sufficiently large n
  - ex. $n + 3 \in O(n^2)$, $n + 3 \in O(n)$
- T(n) is said to be in the set **Θ(**f(n)**)** iff T(n) grows **as fast** as f(n)
  - Formally $\exists\, c_1 > 0: \exists\, c_2 > 0: \exists\, n_0 > 0: \forall\, n > n_0: c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$
  - There exists some positive constant factors c_1 and c_2 such that for sufficiently large n T(n) is between $c_1 \cdot f(n)$ and $c_2 \cdot f(n)$
  - ex. $n + 3 \in \Theta(n)$, $2n^2 - 3n + 1 \in \Theta(n^2)$

- Given a polynomial time complexity $T(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots + c_1 n + c_0$, T(n) is in big O and big Θ of n^k

The following ordering ranks complexities best to worst:
$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^k) < O(k^n) < O(n!) < O(n^n)$

Application of Big O

If we assume that the judge (kattis) computer can perform $\mathbf{10^9}$ **ops/s** we can use Big O (or Big θ) to estimate whether or not a solution will run in time given the following:

let **N** be the max input size, **s** be the number of seconds allowed (the question will/should provide both), and **f(n)** be a function bounding the runtime of your algorithm (i.e. $T(n) \in O(f(n))$) then your algorithm will (most likely) run in time if $\mathbf{f(N) \leq s \cdot 10^9}$ then

Greedy Algorithms

Greedy Algorithms

- For some problems, we can find the global optimum by repeatedly choosing a local optimum (using a *Greedy Heuristic*)
- Takes some creativity to find the right local optimum
  - The right local optimum can provably "stay ahead" of any competing strategy at each step
  - Greedy approaches can be thought of as extending partial solutions to a final solution. If each extension is no worse than one from an optimal solution then the final solution will be an optimal one as well
  - Greedy algorithms usually take linear O(n) or linearthmic/log-linear O(n log n) time (in the case of optimizing using a priority queue) so watch the input bounds!
    - Often using a pre-sort (which can be done in O(n log n)) followed by an iteration with constant operations resulting in O(n log n + n) = O(n log n) or using a priority queue and an iteration with logarithmic operations resulting in O(n log n) as well

Greedy Algorithms Example

- The Dragon of Loowater

  Problem:

  We have *n* knights and a dragon with *m* heads. Each knight has a height and can only cut off a single head and said head must have a diameter equal or smaller to the knights height. The knight will then require payment from the king in proportion to their height Determine the minimum cost for the king to defeat the dragon (if possible)

  - Greedy Heuristic: match the smallest remaining dragon head with the shortest available knight that can cut it off
  - Key insight: by sorting in increasing order, either the next shortest knight can chop off the next smallest head (which minimizes the cost for that head) or the next shortest knight can't chop off *any* remaining heads (because we've sorted both)

This Week's Contest

https://open.kattis.com/contests/z2pn7m

(or look up "CPC Fall 2023 Practice Contest Week 4" in the Kattis contest list)

Feel free to ask questions until 7pm, and then throughout the week on Discord!

COMPETITIVE PROGRAMMING CLUB