1 2 3 4 5 6 7

☆ Concurrent Container

Consider the following implementation of a container that will be used in a concurrent environment. The container is supposed to be used like an indexed array, but provide thread-safe access to elements.

```
struct concurrent_container
{
    // assume it's called for any new instance once before it's ever used
    void construct(int N)
    {
        init_mutex(&lock);
        resize(N);
    }

    // returns element by its index.
    int get(int index)
    {
        lock.acquire();
        if (index < 0 || index > size) {
            return -INT_MAX;
        }
        int result = data[index];
        lock.release();

        return result;
    }

    // sets element value by its index.
    void set(int index, int value)
    {
        lock.acquire();
        if (index < 0 || index > size) {
            return;
        }
        data[index] = value;
        lock.release();
    }

    // extend maximum capacity of the container so we can add more elements
    void resize(int N)
    {
        lock.acquire();
        size = N;
        data = (int *)malloc(size);
        // Ensure we store 0 for elements that have never been set
        for (int i = 0; i < size; ++i)
            data[i] = 0;
        lock.release();
    }

    // assume it's only called once when the container object is destroyed and no one is going to use it anymore
    void destroy()
    {
        free(data);
        data = NULL;
        size = 0;
        destroy_mutex(&lock);
    }

// Assume nobody (except the member methods) is ever going to access these fields directly
    mutex lock;
    int size;
    int * data;
};
```

Which of the following problems do you think are present in the implementation? You may select multiple options.

Pick the correct choices

- [x] The implementation can cause a deadlock.
- [x] The implementation can cause a resource leak.
- [ ] The implementation doesn't prevent multiple threads from accessing data at the same time.
- [ ] The implementation can cause a memory corruption.
- [x] The implementation can result in unaligned memory access, which is forbidden on some platforms.
- [x] The implementation doesn't do proper handling of invalid input.
- [ ] The implementation is not protected from integer overflow errors.
- [ ] The implementation is actually correct and thread-safe.

Clear selection

Cat Pictures Backup

Complete the blanks in the following question with the appropriate answer.

Luke has just bought 7 disk drives and grouped them together into a single logical drive so he could back up all funny cat pictures from the internet. Luke's drive usage can be described as a sustained workload of 200,000 write operations per second with following distribution:

1. 512 bytes - 20% of all operations
2. 1024 bytes - 30% of all operations
3. 4096 bytes - 40% of all operations
4. 16384 bytes - 10% of all operations

Each write operation appends sequentially to an individual drive, and the load is evenly distributed among all drives. There is no compression, garbage collection, or redundancy elimination involved.

Assuming each drive has a capacity of exactly 513 GiB (1 GiB = 1024 MiB, 1 MiB = 1024 KiB, 1 KiB = 1024 bytes), how soon will Luke's array be filled from completely empty to 86% (at that point, Luke should probably consider buying few more disks)? Please round the answer to minutes.

Answer: 75 minutes

DFS in Binary Tree

Complete the blanks in the following question with the appropriate answer.

Jim wrote the following program to perform a depth-first search (DFS) in a binary tree;

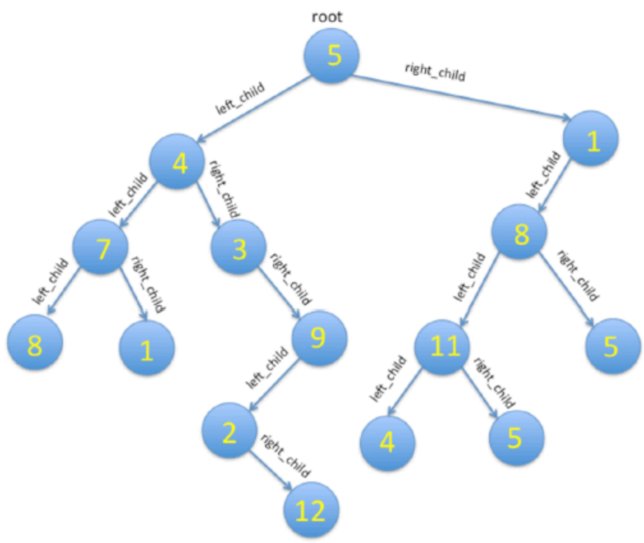
```
struct Node
{
    int value;
    Node * parent;
    Node * left_child;
    Node * right_child;
    bool   visited;
};

Node * dfs(Node * node, int target)
{
    printf("%d ", node->value);
    node->visited = true;

    if (node->value == target) {
        return node;
    }

    Node * nodes[3] = {node->right_child,
                       node->parent,
                       node->left_child};
    for (int i = 0; i < 3; ++i) {
        if (nodes[i] && !nodes[i]->visited) {
            Node * result = dfs(nodes[i], target);
            if (result) {
                return result;
            }
        }
    }

    return NULL;
}
```



What will the program output be for the above tree when dfs() is called for node 11 and target is 12? 11 5 8 5 1 5 4 3 9 2 12

Efficient Memory Layout

Complete the blanks in the following question with the appropriate answer.

Suppose you want to store the following properties per each memory page:

1. **Page status** - one of:
   a) In use (has some useful data)
   b) Dirty (stored data previously; ready to be reused)
   c) Free (never stored any data)
2. **Page index** - an integer in range [0 .. 364847]
3. **Access mode** - one of:
   a) Read-only
   b) Write-only
   c) Execute only
   d) Can read and write
   e) Can read and execute
   f) Can read, write and execute
4. **Process ID** - an integer in range [0 .. 37337] storing the process ID the page is allocated to

Assuming we need to keep the direct field access API as simple as possible (e.g. we can't have an additional compression layer), what is the minimal number of bytes we need to store a single record?

Answer: 5 bytes.

Count Palindromes

A string S is called palindome if it reads same way if spelled backwards, for example: "nolemonnomelon", "ASANtaLivedAsAdeviLatNASA".
Any non-empty string has substrings that are palindromes. For example, in the string S="hellolle", there are many of such "subparlindromes":
1) ellolle
2) ll, ll - note that these are two distinct substrings that only happen to be equal
3) lol and lloll
4) And, of course, each letter can be considered a palindrome - all 8 of them.

Write a function that, given a string S (that only consists of lowercase English letters), counts how many different ways are there to pick a palindrome substring from S.

Examples:

1. Input: hellolle
   output: 13 (the above example)
2. Input: wowpurerocks
   output: 14 (each letter + "wow" + "rer")

YOUR ANSWER

We recommend you take a quick tour of our editor before you proceed. The timer will pause up to 90 seconds for the tour. Start tour

Draft saved 03:45 pm   Original code   C++

```
/*
 * Complete the function below.
 */
int count_palindromes(string S) {
```

Lock Use Analyzer

Suppose we want to monitor how locks are used in our system. As the first step, we log moments of acquire and release for each lock in the following format:

- ACQUIRE X
- RELEASE X

where X is some integer ID (1<=X<=1,000,000) of the lock.

All locks must be released in the reverse order of acquiring them; for example, this is a correct event sequence:

1. ACQUIRE 364
2. ACQUIRE 84
3. RELEASE 84
4. ACQUIRE 1337
5. RELEASE 1337
6. RELEASE 364

However, the following sequence violates this rule, because lock 84 is still acquired while releasing lock 364:

1. ACQUIRE 364
2. ACQUIRE 84
3. **RELEASE 364**
4. RELEASE 84

It's also dangerous to leave locks acquired after application termination, as other processes in the system may be blocked while waiting on them, so such sequence is incorrect, too:

1. ACQUIRE 364
2. ACQUIRE 84
3. RELEASE 84

since lock 364 is never released

Third type of problem is lock misuse: it's never good to release a lock that has never been acquired, e.g.:

1. ACQUIRE 364
2. **RELEASE 84**

3. RELEASE 364

and it is as bad to acquire an already acquired lock (usually resulting in a deadlock):
1. ACQUIRE 364
2. ACQUIRE 84
3. **ACQUIRE 364**
4. RELEASE 364

Write a program that, given a list of **N (0<=N<=1,000,000)** lock acquire and release events (counting from 1), checks if there were any problems (acquire-release order violation, dangling acquired lock, acquiring a lock twice or releasing a free lock), and if so, tells the earliest time that could be detected. Note that there's no limit on how many nested locks may be acquired at any given moment.
More formally, you are given an array of strings where each string is either "ACQUIRE X" or "RELEASE X", where all **X**s are integers in the range [1..1000000].
Return:
- **0**, if there were no lock-related problems even after program termination
- **N+1**, if the only issue after program termination were dangling acquired locks
- **K**, in case event number **K** violated any of the principles (release a lock not acquired previously, acquire an already held lock OR violate lock acquire-release ordering).

Examples:
Input:
1. ACQUIRE 364
2. ACQUIRE 84
3. RELEASE 84
4. RELEASE 364

Output: 0 (nothing bad happened)

Input:
1. ACQUIRE 364
2. ACQUIRE 84
3. RELEASE 364
4. RELEASE 84

Output: 3 (lock 84 should have been released before releasing 364)

Input:
1. ACQUIRE 123
2. ACQUIRE 364
3. ACQUIRE 84
4. RELEASE 84
5. RELEASE 364
6. ACQUIRE 456

Output: 7 (upon terminating, not all locks were released, namely 123 and 456, but we can't know that until actually exiting)

Input:
1. ACQUIRE 123
2. ACQUIRE 364
3. ACQUIRE 84
4. RELEASE 84
5. RELEASE 364
6. ACQUIRE 789
7. RELEASE 456
8. RELEASE 123

Output: 7 (releasing a lock not acquired before)

Input:
1. ACQUIRE 364
2. ACQUIRE 84
3. ACQUIRE 364
4. RELEASE 364

Output: 3 (acquiring an already held lock)

YOUR ANSWER

We recommend you take a quick tour of our editor before you proceed. The timer will pause up to 90 seconds for the tour. Start tour

☆ Breaking Binary Search

Bob is trying to write a program that performs a binary search for an element within an array (sorted in ascending order). To get feedback from his friends, he's written it in multiple languages. Here's his program - all versions work the same way, so feel free to look at the language you're most comfortable with:

C / C++ version:
```
int sorted_search(int * elements, int size, int target)
{
    if (size <= 0 || !elements) return -1;

    int left = 0, right = size - 1;
    while (left < right) {
        int middle = (left + right + 1) / 2;

        if (elements[middle] > target) {
            right = middle - 1;
        } else {
            left = middle + 1;
        }
    }

    if (elements[right] == target) return right;
    return -1;
}
```

Java version:
```
int sorted_search(int[] elements, int target)
```

```
{
    if (elements == null || elements.length <= 0) return -1;

    int left = 0, right = elements.length - 1;
    while (left < right) {
        int middle = (left + right + 1) / 2;

        if (elements[middle] > target) {
            right = middle - 1;
        } else {
            left = middle + 1;
        }
    }

    if (elements[right] == target) return right;
    return -1;
}
```

Python version:

```
def sorted_search(elements, target):
    if not elements or len(elements) <= 0:
        return -1

    left = 0
    right = len(elements) - 1
    while left < right:
        middle = (left + right + 1) / 2

        if elements[middle] > target:
            right = middle - 1
        else:
            left = middle + 1

    if elements[right] == target:
        return right
    return -1
}
```

Is Bob's program correct? If not, what **valid** input will lead to an incorrect result?

Write a program that either:

1. prints "CORRECT" (without quotes) if Bob's binary search implementation is correct.
2. prints a sorted array and a target number that makes Bob's implementation return the wrong result.
   In this case, your program should output exactly 3 lines:
   a) first line should contain the number of elements in the input array
   b) second line should contain space-delimited elements of the input array
   c) third line should contain the target number.

We recommend you take a quick tour of our editor before you proceed. The timer will pause up to 90 seconds for the tour. Start tour ×

Original code | C++

```
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
```