Competitive Programming and Mathematics Society

Programming Workshop #1

Stack Hacks

Angus Ritossa

Today's Workshop

What is a stack?

CPMSOC

- Simple data structure
- Supports three operations
  - Add an item to the top of the stack
  - Remove an item from the top of the stack
  - View the item at the top of the stack
- Each operation can be supported in O(1) time (e.g. with an array, or with a linked list)

Stacks in C++

CPMSOC

```
#include <stack>                                           // Include this to use std::stack
#include <cstdio>                                          // The same as stdio.h, gives printf/scanf
using namespace std;                                       // Allows us to type stack<int> instead of std::stack<int>
int main() {
    stack<int> s;                                          // Declares a stack called s
    s.push(3);                                             // Push 3 onto the stack
    printf("top of stack: %d\n", s.top());                 // prints 3
    printf("size of stack %d\n", (int)s.size());           // prints 1
    s.push(1);
    s.push(2);
    printf("top of stack: %d\n", s.top());                 // prints 2
    printf("size of stack %d\n", (int)s.size());           // prints 3
    s.pop();                                               // Remove the top element from the stack
    printf("top of stack: %d\n", s.top());                 // prints 1
    printf("size of stack %d\n", (int)s.size());           // prints 2
    s.push(4);
    printf("top of stack: %d\n", s.top());                 // prints 4
    printf("size of stack %d\n", (int)s.size());           // prints 3
    s.pop();
    printf("top of stack: %d\n", s.top());                 // prints 1
    printf("size of stack %d\n", (int)s.size());           // prints 2
    s.pop();
    printf("top of stack: %d\n", s.top());                 // prints 3
    printf("size of stack %d\n", (int)s.size());           // prints 1
    s.pop();
    printf("size of stack %d\n", (int)s.size());           // prints 0
}
```

Problem: Bracket Matching

CPMSOC

You are given a string consisting of '(', ')', '[', ']', '{' and '}'. A bracket sequence is valid if every opening bracket matches a closing bracket of the same type. More formally:

- The empty string is a valid bracket sequence
- If A is a valid bracket sequence, (A), [A] and {A} are all valid bracket sequences
- If A and B are valid bracket sequences, AB is a valid bracket sequence

Determine whether the given bracket sequence is valid.
You are guaranteed that $N \leq 100\,000$, where N is the size of the bracket sequence.

Sample input 1: `([()()]{([]())})[]`
Sample output 1: `YES`

Sample input 2: `([)]`
Sample output 2: `NO`

Problem: Bracket Matching

- If a bracket sequence is valid, we should be able to find the match of each bracket
- Informal observation: we assign each closing bracket to the closest available opening bracket.
- We can formalise this by processing the string one bracket at a time, and considering what we should do with each bracket as we get it.
    - An opening bracket will match with some closing bracket in the future.
    - A closing bracket will match with the nearest unmatched opening bracket.
    - We can facilitate these with a stack. Push opening brackets onto a stack, and match closing brackets with the top opening bracket on the stack.

Problem: Bracket Matching

- When might we get a `NO` result?
  1. We match different types of brackets together
  2. There is no unmatched opening bracket for a closing bracket (stack is empty when we reach a closing bracket)
  3. There are unmatched opening brackets at the end (stack is not empty at the end)
- Time complexity? We do $O(1)$ work for each of the N brackets, so its $O(N)$ overall.

Problem: Bracket Matching

```
#include <stack>                        // Include this to use std::stack
#include <cstdio>                       // The same as stdio.h, gives printf/scanf
#include <cstring>                      // The same as string.h, gives strlen
#include <cstdlib>                      // The same as stdlib.h, Gives exit
using namespace std;
#define MAXN 100010
char brackets[MAXN];
void NO() {
    // Prints no and ends the program
    printf("NO\n");
    exit(0);                            // exit(0) terminates the program without any errors
}
```

Problem: Bracket Matching

CPMSOC

```
int main() {
    scanf(" %s", brackets);
    int n = strlen(brackets);
    stack<char> s;
    for (int i = 0; i < n; i++) {
        if (brackets[i] == '(' || brackets[i] == '[' || brackets[i] == '{') {
            s.push(brackets[i]);
        } else {
            if (s.empty()) {
                NO();                   // No opening bracket to match us with
            } else if (brackets[i] == ')' && s.top() != '(') {
                NO();                   // Matched the wrong types of brackets together!
            } else if (brackets[i] == ']' && s.top() != '[') {
                NO();                   // Matched the wrong types of brackets together!
            } else if (brackets[i] == '}' && s.top() != '{') {
                NO();                   // Matched the wrong types of brackets together!
            }
            s.pop();                    // Remove the opening bracket from the stack, as it is now matched
        }
    }
    if (!s.empty()) {
        NO();                           // There are unmatched opening brackets!
    }
    printf("YES\n");
}
```

Problem: Sculpture II

There are N LEGO blocks, which will create a structure on a line.

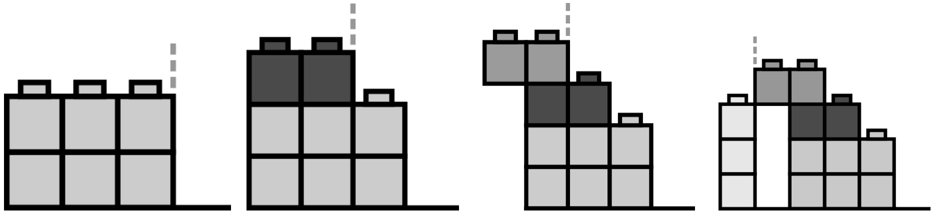
The i-th block is w_i cm wide and h_i cm tall. After t_i seconds, i-th the block will magically fall from the sky, with its right end $t_i$cm along the line (measured from the right). If a block lands on top of another, it will stick to it.

What is the height of the highest point on the sculpture?

Problem: Sculpture II

For example, imagine there are four blocks

1. The first block has $t_i = 1$, $w_i = 3$ and $h_i = 2$.
2. The second block has $t_i = 2$, $w_i = 2$ and $h_i = 1$.
3. The third block has $t_i = 3$, $w_i = 2$ and $h_i = 1$.
4. The fourth block has $t_i = 5$, $w_i = 1$ and $h_i = 3$.



The height of the sculpture is 4

Problem: Sculpture II

Bounds

- $N \leq 100\,000$
- $w_i, t_i \leq 1\,000\,000$ for all i
- $h_i \leq 1\,000$ for all i
- $t_i < t_{i+1}$ (that is, the blocks are sorted by time, and times are unique)

Subtasks

1. All values $\leq 1\,000$
2. $w_i = w_j$ and $h_i = h_j$ for all i and j. That is, all blocks have the same dimensions.
3. Full

Source: Australian Informatics Olympiad 2016

Sculpture II: Solution

- Observation: each block will sit atop at most one other block
- In other words, when the *i*-th block is placed there is a single ‘stack’ of blocks which it falls on top of
- Consider this ‘stack’: does it have the properties of a stack?
    - When a new block is added, it is added to the top of the stack.
    - If the t_i of the current block is larger than the $t_j + w_j$ of the top of the stack, the new block will not land on the old top of the stack, and so we should pop the top of the stack
    - A block in the middle of the stack will remain in the stack for at least as long as the block on top of it does. Hence blocks are only removed from the top of the stack.

Sculpture II: Solution

This motivates the following solution

- Process the blocks in order, keeping a stack. Maintain the current height of all the blocks in the stack combined, and the highest stack we have seen so far.
- When we reach a block *i*, pop the top of the stack *j* while $t_i \geq t_j + w_j$ (that is, block *i* will not land on top of block *j*)
- Push block *i* onto the stack. Update the highest stack we have seen so far, if our current stack exceeds the old highest.

Time complexity? We process each block once. For each block, we pop some number of things off the stack, and then push one thing onto the stack. Because each block is popped off the stack once throughout the entire solution, the overall complexity is $O(N)$.

Sculpture II: Code

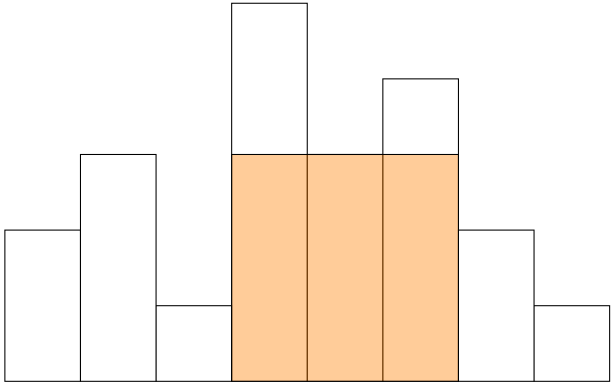
```
#include <stack>                               // Include this to use std::stack
#include <cstdio>                              // The same as stdio.h, gives printf/scanf
#include <algorithm>                           // Gives us std::max
using namespace std;
#define MAXN 100010
int n, t[MAXN], w[MAXN], h[MAXN];
int curr_hei;                                  // the height of the current stack
int ans;                                       // the height of the highest stack we have seen so far
stack<int> s;                                  // will contain the indicies of the blocks on the current stack
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d%d%d", &t[i], &w[i], &h[i]);

        // this is the condition for popping from the solution slide
        while (!s.empty() && t[i] >= t[s.top()]+w[s.top()]) {
            curr_hei -= h[s.top()];            // we remove the block from the stack, so decrement the current stack height
            s.pop();
        }

        curr_hei += h[i];                      // we add the block to the stack, so increment the current stack height
        ans = max(ans, curr_hei);              // update the answer if our stack height is the highest so far
        s.push(i);
    }
    printf("%d\n", ans);
}
```

Problem: Rectangle Under Histogram

You are given a histogram (a shape consisting of a number of bars of equal widths but various heights, which all have their base on the same line). You must find the area of the largest axis-aligned rectangle contained within the histogram.



Problem: Rectangle Under Histogram

Input

- The first line consists of a single integer N, the number of bars.
- The next N lines each contain h_i, the height of the i-th bar.

Output: You must output a single integer: the area of the largest rectangle.

Constraints

- $N \leq 100\,000$
- $h_i \leq 1\,000\,000$ for all i

Subtasks

1. $N \leq 100$
2. $N \leq 1\,000$
3. Full

Problem: Rectangle Under Histogram

Sample Input

```
8
2 3 1 5 3 4 2 1
```

Sample Output

```
9
```

Explanation: this matches the diagram on the first slide

Subtask 1: $N \leq 100$

- The bound on N allows an $O(N^3)$ solution
- Observation: the optimal rectangle always has an edge along the bottom of the histogram
- Solution: we will consider all $O(N^2)$ ranges which could form the base of our rectangle
- In each of these ranges, find the smallest bar. This takes $O(N)$ time.
- The overall time complexity is $O(N^3)$

Subtask 1: $N \leq 100$

```
// O(N^3) solution
#include <cstdio>
#include <algorithm>
using namespace std;
#define MAXN 100010
int n, hei[MAXN], ans;
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &hei[i]);
    }
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            // Consider the range [i, j]. We will find the height of the bar rectangle in this range
            int mn = hei[i];
            for (int k = i; k <= j; k++) {
                mn = min(mn, hei[k]);
            }
            // Update maximum area, if this rectangle exceeds it
            ans = max(ans, (j-i+1)*mn);
        }
    }
    printf("%d\n", ans);
}
```

Subtask 2: $N \leq 1\,000$

- The bound on N allows an $O(N^2)$ solution
- We can optimise the subtask 1 solution. In particular, we can optimise the $O(N)$ step where we find the smallest bar.
- Observe that $\min(h_i, h_{i+1}, ..., h_{j-1}, h_j) = \min(\min(h_i, h_{i+1}, ..., h_{j-1}), h_j)$.
- Consider each i (that is, the start of the range). For each i, we consider each j starting at i and increasing to n.
- As we increase j, we keep track of the smallest bar we have seen. This will correspond to the smallest bar in the range $[i, j]$.
- We have two nested for loops, with a constant amount of work inside the loops. Hence, the overall complexity is $O(N^2)$.

Subtask 2: $N \leq 1\,000$

```
// O(N^2) solution
#include <cstdio>
#include <algorithm>
using namespace std;
#define MAXN 100010
int n, hei[MAXN], ans;
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &hei[i]);
    }
    for (int i = 0; i < n; i++) {
        int mn = hei[i];
        for (int j = i; j < n; j++) {
            mn = min(mn, hei[j]);
            // Update maximum area, if this rectangle exceeds it
            ans = max(ans, (j-i+1)*mn);
        }
    }
    printf("%d\n", ans);
}
```

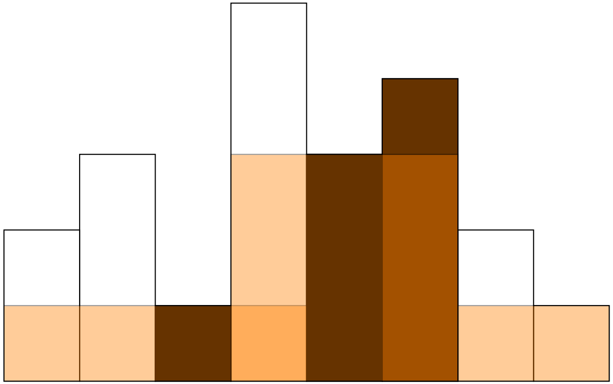
Histogram: Going for full

- We were able to optimise our $O(N^3)$ solution to an $O(N^2)$ solution because there was an unnecessary inner loop.
- In our $O(N^2)$ solution, we consider $O(N^2)$ ranges individually. As such, we cannot just ‘optimise’ the existing solution, we must adopt a new strategy.

Histogram: Going for full

CPMSOC

- Idea: in each range, it was the smallest bar which determined the height of the rectangle. Instead of fixing the base of the rectangle, lets fix the smallest bar in it.

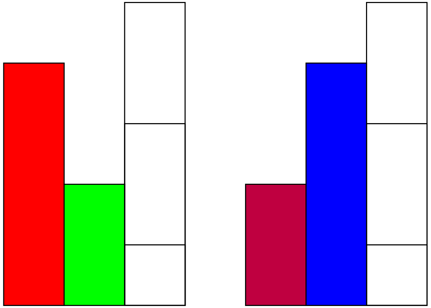


Histogram: Going for full

- New solution: for each i, assume it is the smallest bar in the rectangle.
- We want to find the first bar to the left of bar i that is smaller than bar i (call this the left-limiting bar). Do the same thing for the right.
- We can find these in $O(N)$ time with a simple loop, giving another $O(N^2)$ solution.

Sorted Stack

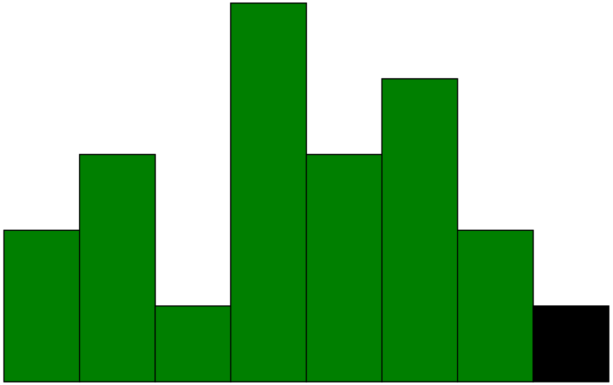
- New problem: for each bar, find its left-limiting bar



- The red bar will never be a left-limiting bar, as the green bar is to the right of it and smaller than it.
- The blue and purple bars could both be left-limiting bars.

Sorted Stack

- Idea: process bars from left to right, and keep track of all bars which could be left-limiting bars in the future.



Sorted Stack

- Problem: for each bar, find its left-limiting bar
- Solution: Process bars from left to right, and maintain a stack.
- When we are processing some bar i, we pop the top of the stack while $h_{\text{top of stack}} \geq h_i$. The new top of the stack is the left-limiting bar. We then push i onto the stack.
- Each bar is pushed and popped from the stack once, so the overall complexity is $O(N)$.

Sorted Stack: Code

```cpp
// n and h are the input
// left_limit is the array which we will use to store the output
void find_left_limits(int n, int h[], int left_limit[]) {
    stack<int> s;                                            // Will contains indices of the bars on the sorted stack
    for (int i = 0; i < n; i++) {
        while (!s.empty() && h[s.top()] >= h[i]) {  // Pop a bar if it can never again be a left-limiting bar
            s.pop();
        }
        if (s.empty()) {
            // The bar does not have a left limit
            left_limit[i] = -1;
        } else {
            left_limit[i] = s.top();
        }
        s.push(i);
    }
}
```

Histogram: Full Solution

- We will use the sorted stack technique to solve the problem
- For each bar, we find the left and right limiting bars (we can use exactly the same technique in reverse to find the right limiting bar)
- It takes $O(N)$ to find the left/right limiting bars, and $O(N)$ to use these to find the answer. Hence, the overall time complexity is $O(N)$.

Histogram: Code

```
#include <cstdio>
#include <algorithm>                          // Gives us reverse
using namespace std;
#define MAXN 100010
typedef long long ll;                         // The maximum possible answer is 10^5 * 10^6 = 10^11, so we need long longs
void find_left_limits(int n, int h[], int left_limit[]); // Omitted - see code from sorted stack slide
int n, hei[MAXN], left_limit[MAXN], right_limit[MAXN];
int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &hei[i]);
    }
    find_left_limits(n, hei, left_limit);
    reverse(hei, hei+n);                              // Reverse the array
    find_left_limits(n, hei, right_limit);            // The right limits (in reverse order) are now stored in right_limit
    reverse(hei, hei+n);
    reverse(right_limit, right_limit+n);
    for (int i = 0; i < n; i++) {
        right_limit[i] = n-1-right_limit[i];          // Because we reversed the array, need to "reverse" every index
    }
    ll ans = 0;
    for (int i = 0; i < n; i++) {
        ll len = right_limit[i]-left_limit[i]-1;      // The width of the rectangle with its lowest bar at bar i
        ll area = len*(ll)hei[i];
        ans = max(ans, area);                         // Check if we improve the answer
    }
    printf("%lld\n", ans);
}
```