# Largest Rectangle

*Skyline Real Estate Developers* is planning to demolish a number of old, unoccupied buildings and construct a shopping mall in their place. Your task is to find the largest solid area in which the mall can be constructed.

There are a number of buildings in a certain two-dimensional landscape. Each building has a height, given by $h[i]$ $\textbf{where}$ $i \in [1,\ n]$. If you join $k$ adjacent buildings, they will form a solid rectangle of area $k \times min(h[i], h[i+1], \ldots, h[i+k-1])$.

For example, the heights array $h = [3, 2, 3]$. A rectangle of height $h = 2$ and length $k = 3$ can be constructed within the boundaries. The area formed is $h \cdot k = 2 \cdot 3 = 6$.

**Function Description**

Complete the function `largestRectangle` int the editor below. It should return an integer representing the largest rectangle that can be formed within the bounds of consecutive buildings.

largestRectangle has the following parameter(s):

- *h*: an array of integers representing building heights

**Input Format**

The first line contains $n$, the number of buildings.
The second line contains $n$ space-separated integers, each representing the height of a building.

**Constraints**

- $1 \leq n \leq 10^5$
- $1 \leq h[i] \leq 10^6$

**Output Format**

Print a long integer representing the maximum area of rectangle formed.

**Sample Input**

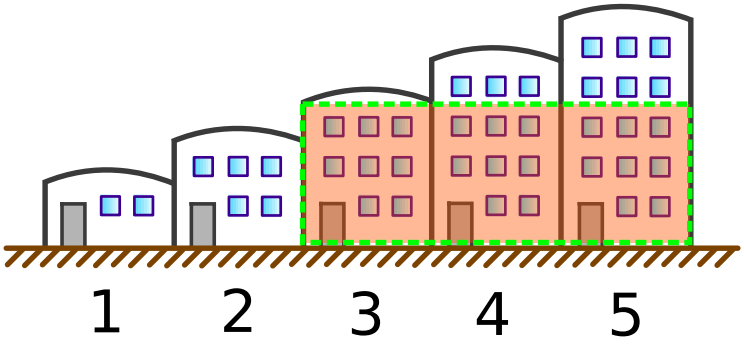
```
5
1 2 3 4 5
```

**Sample Output**

```
9
```

**Explanation**

An illustration of the test case follows.


1
2
3
4
5

# Largest Rectangle ★



Editorial by bishop15

This problem is a classic one in the context of Algorithmic Programming Contests. It asks for the area of the largest rectangle which can be segmented from a given Histogram with bars of different heights but fixed width.

There are many nice approaches to solving this problem. We will review two of those. For simplicity, consider each bar has a unit width. We will use a notion of a **Rectangle** for each bar and **two indices**, $j$ and $k$.

## Naive Approach:

Let's think about the most naive approach first. This will help us to understand a more efficient solution.

What we can do: for each bar, we can find the rectangle with the height of it and also containing it.

To do this for a bar, let's say $i^{th}$ bar, we go to left from $i$ and stop at the first bar which is smaller than the $i^{th}$ bar. Say we have stopped at $j^{th}$ position.

Then, we move to the right of $i$ and stop at the bar which is smaller than $i^{th}$ bar. Say we have stopped at $k^{th}$ bar.

**So, the rectangle containing $i^{th}$ bar and of its height will have width from position $j$ to $k$(exclusive). The width will be $k - j - 1$.**

We can calculate the area of this rectangle for each bar and take the maximum. That will be our answer.

**Time Complexity:** For each bar we are considering a linear search. Thus it is $O(n^2)$ in total.

## $O(n)$ Approach:

The objective is same, i.e. for each bar, we will calculate the same rectangle we did in the naive approach but of course efficiently.

Just restating: to find the width of the desired rectangle for any $i^{th}$ bar, we need the position of first smaller element left to $i$ (was denoted by $j$ in naive approach) and the position of the first smaller element right to $i$(denoted by $k$).

Here comes the magic of stack!

Consider the code:

```
while x < stack.top():
    stack.pop()
stack.push(x)
```

This operation will create an increasing stack with a fascinating property!

For $stack[i]$ the first smaller element left to it will be $stack[i-1]$ and the first element which will make $stack[i]$ pop will be the first smaller element right to $stack[i]$.

We see that this stack will keep track of the first smaller element to the left and right of each element. By using this property, we can calculate our desired rectangle for each bar in just $O(n)$!

If the property is confusing then let's look into a **proof**:

It is kind of evident that the first element which will make any $stack[i]$ pop is the first smaller element right to $stack[i]$.

Now let's think about $stack[i-1]$ and $stack[i]$. $stack[i-1]$ is smaller than $stack[i]$, this is for sure. But say, it is not the nearest smaller element. Let, $S_x$ be the first smaller element left to $stack[i]$.

If $S_x < stack[i-1]$:

- Then $stack[i-1]$ must be popped by $S_x$.

STATISTICS

| | |
|---|---|
| Difficulty: | Medium |
| Time Complexity: | O(n), O(nlgn) |
| Required Knowledge: | Stack, Largest Rectangle in Histogram, Divide and Conquer |
| Publish Date: | May 05 2015 |

This is a Practice Challenge

NEED HELP?

View discussions

View top submissions

If $S_x \geq stack[i-1]$:

- Then obviously $S_x$ will be pushed to stack and will be right to $stack[i-1]$.

So, either case, $S_x$ will be between $stack[i-1]$ and $stack[i]$. That implies, $S_x = stack[i-1]$

The above idea is very neatly implemented in the solution code. If you are still confused about the properties, let's simulate the stack operations for $3, 2, 5, 6, 4$ and $5$.

**Step1:** $3$

The stack is empty so push $3$.

3

**Step2:** $2$

Two is less than the top element. Pop it and then push $2$.

2

Notice that for $3$, **there is no element in it's left which is smaller than it** and $2$ is the nearest smaller element to it's right. The area = $3 \times 1 = 3$.

**Step3+4:** $5, 6$

Push $5$ and then $6$.

6
5
2

**Step5:** $4$

$4 < 6$. Pop 6.

Notice, our $j$ is the position of $5$(which is the immediate previous element of $6$ in the stack) and $k$ is the position of $4$ (the element which make $6$ pop from the stack).

Area for $6 = 6 \times (k - j - 1) = 6 \times (4 - 2 - 1) = 6$.(considering $0$ based index).

$4 < 5$. Pop 5.

$j$ is the position of $2$ and $k$ is the position of $4$.

Area for $5 = 10$.

$4 > 2$. Push $4$

4
2

Notice, $2$'s position is the position of $j$ for $4$, i.e.,. $2$ is the nearest smaller element of $4$ to it's left.

**Step6:** $5$

$5 > 4$. Push $5$.

5
4
2

There is no element left and stack is not empty. That means, **for $2, 4$ and $5$ every element right to it is greater or equal!**.

To handle this scenario automatically, we can append a $0$ to the height list before starting. Why? Figure this one out by yourself :).

## Divide and Conquer Approach:

Amongst all other approaches, I find this method the easiest to understand.

Consider the full histogram: $[start, end]$. Find the position of the minimum value in $[start, end]$. Let it be $x$.

If the optimal subpart contains the $x^{th}$ bar. Then it's area would be: $Height[x] \times (end - start + 1)$.

If the optimal subpart doesn't contain the $x^{th}$ bar then, either it will be in $[start, x-1]$ or in $[x+1, end]$, that's our

divide and conquer!

$Area(start, end)$ = max($Height[x] \times (end - start + 1)$, $Area(start, x-1)$, $Area(x+1, end)$); $x$ is the position of the smallest bar in $[start, end]$.

This idea can very easily be implemented using a segment tree.

**Editorialist's Code:**

C++

```
#include <bits/stdc++.h>
using namespace std;

typedef long long           lli;

#define pb push_back


vector < lli > height;
vector < int > s;
lli Histogram(vector<lli> &height)
{
    s.clear();
    height.push_back(0);

    lli sum = 0;
    int i = 0;
    while(i < height.size())
    {
        if(s.empty() || height[i] > height[s.back()])
        {
            s.push_back(i);
            i++;
        }
        else
        {
            int t = s.back();
            s.pop_back();

            sum = max(sum, height[t] * (s.empty() ? i : i - s.back() - 1));
        }
    }

    return sum;
}

int main(void)
{
    int i,j,k,kase=0;

    int n;
    while( scanf("%d",&n)==1 )
    {
        height.assign(n, 0);
        for(i=0; i<n; i++) scanf("%lld",&height[i]);
        printf("%lld\n",Histogram(height));
    }
    return 0;
}
```

### Feedback

Was this editorial helpful?

Yes No