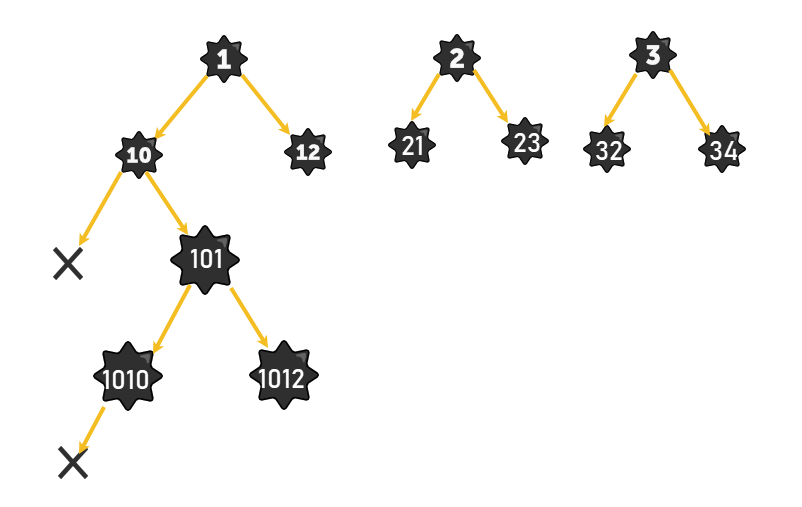
## BINARY SEARCH

1. Range Related Problems (Search a value in ordered range).
2. Optimization Answer in a Range (Binary Search on Answer).
3. Find Median of Two Sorted Arrays.
4. Problems on Sorted Rotated Arrays (Search for Min, Find a Number in logN).
5. Given an array and [A, B] count the number of values between A and B.
   1. [4, 5, 9, 11, 22,22, 33, 33,45, 67, 88, 99]
   2. A = 20, B = 89

Note: All below data structures do not give us random access.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | STACK | QUEUE | DEQUE(DECK) | MINPQ |
| INSERT | O(1) | O(1) | insert\_begin inset\_end O(1) | O(logN) |
| DELETE | O(1) | O(1) | delete\_begin delete\_end O(1) | O(logN) |
| GET | O(1) | O(1) | front() back() O(1) | O(1) |
| SIZE | O(1) | O(1) | O(1) | O(1) |
| EMPTY | O(1) | O(1) | O(1) | O(1) |

[**PROBLEM**] Print all jumping numbers from [1, N]. N = 200

* Jumping numbers are the numbers where its adjacent digit difference is 1.
* All single digit numbers are jumping numbers.
* Example:
  + 2123
  + 789

**APPROACH 1:**

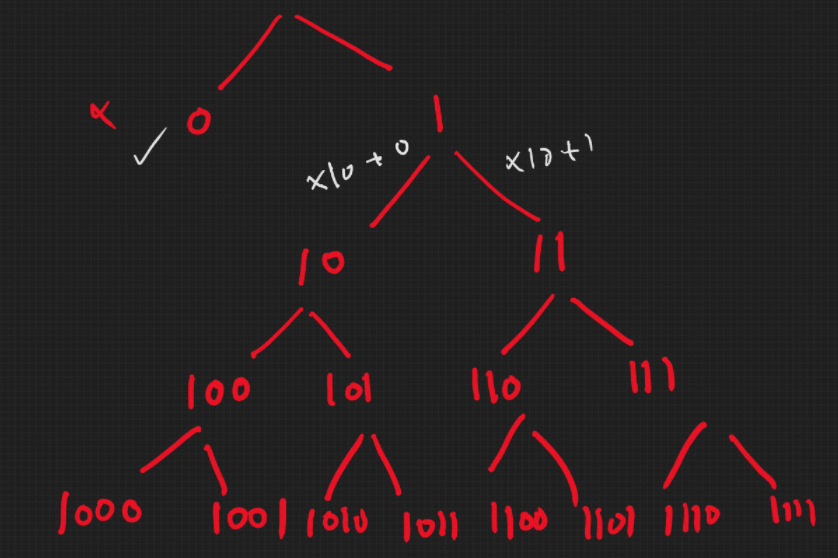
1. For each number [1, N]
   1. if(isJumping(i))
      1. print(i)

TC: NlogN

SC: 1

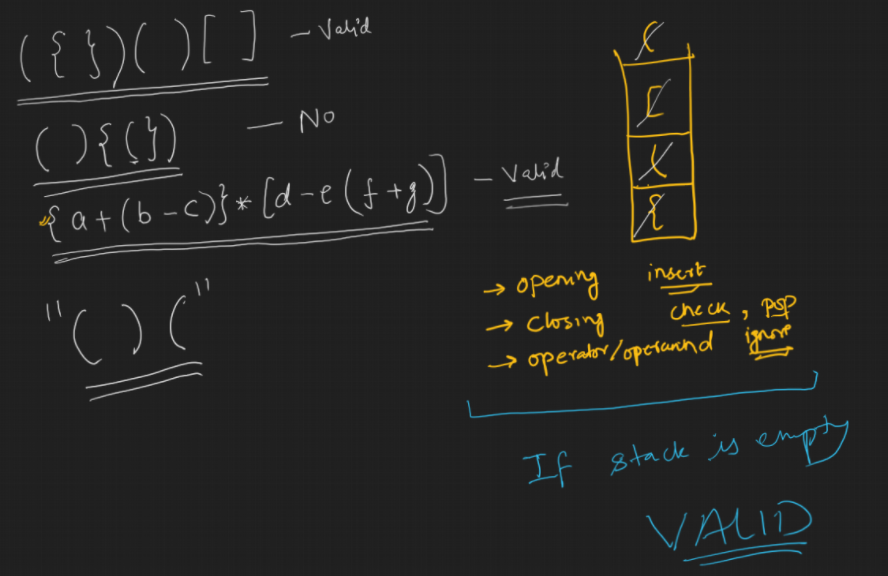
|  |
| --- |
| *vector*<int> getAllJumpingNumbers(int n) {  *queue*<int> q;  for (int i = 1; i <= *min*(n, 9); i++)  q.*emplace*(i);  *vector*<int> ans;  while (!q.*empty*()) {  int f = q.*front*();  q.*pop*();  ans.*push\_back*(f);  if (f % 10 != 0) {  int left = f \* 10 + f % 10 - 1;  if (left <= n)  q.*emplace*(left);  }  if (f % 10 != 9) {  int right = f \* 10 + f % 10 + 1;  if (right <= n)  q.*emplace*(right);  }  }  return ans;  }  int main(void) {  auto ans = getAllJumpingNumbers(200);  for (auto i : ans) {  *cout* << i << " ";  }  return 0;  } |

**[PROBLEM]** Given N, get all binary patterns of numbers from [0, N]



|  |
| --- |
| *vector*<int> getBinaryPattern(int n) {  *queue*<int> q;  *vector*<int> ans;  ans.*push\_back*(0);  if (n == 0) return ans;  q.*emplace*(1);  while (!q.*empty*()) {  int f = q.*front*();  q.*pop*();  ans.*push\_back*(f);  int left = f \* 10 + 0;  if (left <= n)  q.*emplace*(left);  int right = f \* 10 + 1;  if (right <= n)  q.*emplace*(right);  }  return ans;  }  int main(void) {  auto ans = getBinaryPattern(100000);  for (auto i : ans) {  *cout* << i << " ";  }  return 0;  } |

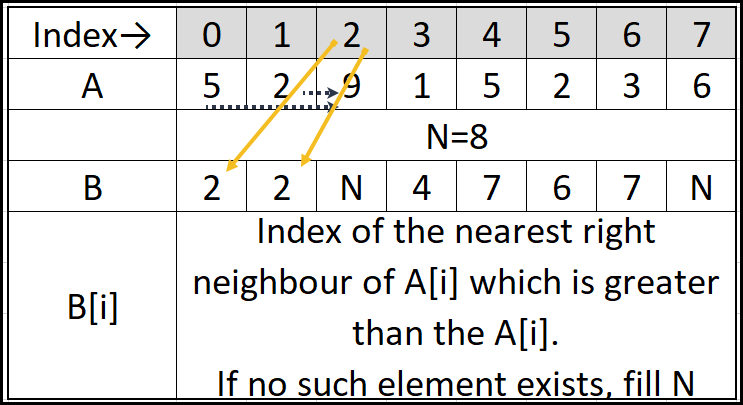
[**PROBLEM**] Given a string of parentheses of 3 types. ( , [, {. Tell if the given pattern is balanced or not.



|  |
| --- |
| #include <iostream>  #include <stack>  #include <vector>  #include <string>  using namespace *std*;  char getClosing(char c) {  if (c == '(') return ')';  else if (c == '[') return ']';  else if (c == '{') return '}';  else return '\0';  }  bool isBaIancedParenthesis(*string* s) {  *stack*<char> stk;  for (auto c : s) {  if (c >= 'a' && c <= 'z') continue;  if (c >= 'A' && c <= 'Z') continue;  if (c == '+' || c == '-' || c == '\*' || c == '/' || c == '%')continue;  if (c == '(' || c == '[' || c == '{')  stk.*push*(getClosing(c));  else if (c == ')' || c == ']' || c == '}') {  if (stk.*empty*()) return false;  if (stk.*top*() != c) return false;  stk.*pop*();  }  }  return stk.*empty*();  }  int main(void) {  *vector*<*string*> s;  s.*push\_back*("");  s.*push\_back*("({})(){}[]");  s.*push\_back*("()({}[)]");  s.*push\_back*("{a+(b-c)}\*[d-e(f+g)]");  for (auto i : s)  if (isBaIancedParenthesis(i)) *cout* << "YES\n";  else *cout* << "NO\n";  return 0;  } |

**[PROBLEM]** For each element in A, fill index in B such that

B[i] index of nearest greater element of A[i] in array A.



## BRUTE FORCE:

|  |
| --- |
| #include <iostream>  #include <stack>  #include <vector>  using namespace *std*;  *vector*<int> nearestGreaterToTheRight(*vector*<int> a) {  *vector*<int> b(a.*size*());  int len = a.*size*();  for (auto i = 0; i < len; i++) {  int idx = len;  for (auto j = i + 1; j < len; j++) {  if (a[j] > a[i]) {  idx = j;  break;  }  }  b[i] = idx;  }  return b;  }  int main(void) {  *vector*<int> a = { 5, 2, 9, 1, 5, 2, 3, 6 };  auto b = nearestGreaterToTheRight(a);  for (auto i : b)  *cout* << i << " ";  return 0;  } |

## STACK BASED SOLUTION

TC: **2N**

SC: N

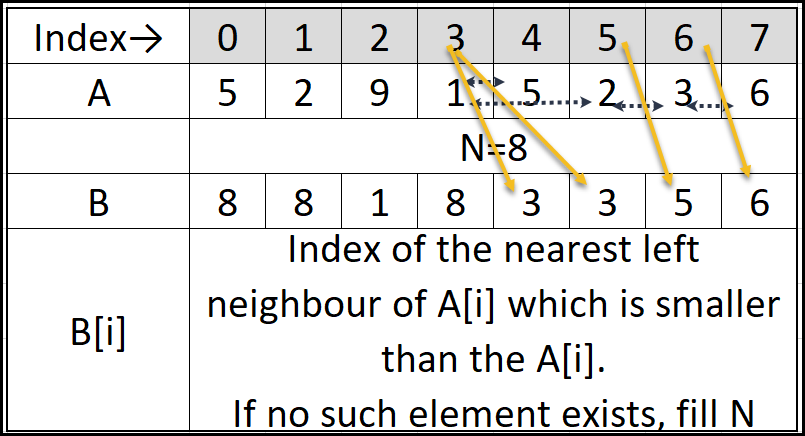
Time Complexity: Count total number of PUSH and POP operations being carried out.

|  |
| --- |
| #include <iostream>  #include <stack>  #include <vector>  using namespace *std*;  *vector*<int> nearestGreaterToTheRight(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk; // Store only indices  for (auto i = 0; i < len; i++) {  while (!stk.*empty*() && a[stk.*top*()] < a[i]) {  b[stk.*top*()] = i;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = a.*size*();  stk.*pop*();  }  return b;  }  int main(void) {  *vector*<int> a = { 5, 2, 9, 1, 5, 2, 3, 6 };  auto b = nearestGreaterToTheRight(a);  for (auto i : b)  *cout* << i << " ";  return 0;  } |

## Stack based problems

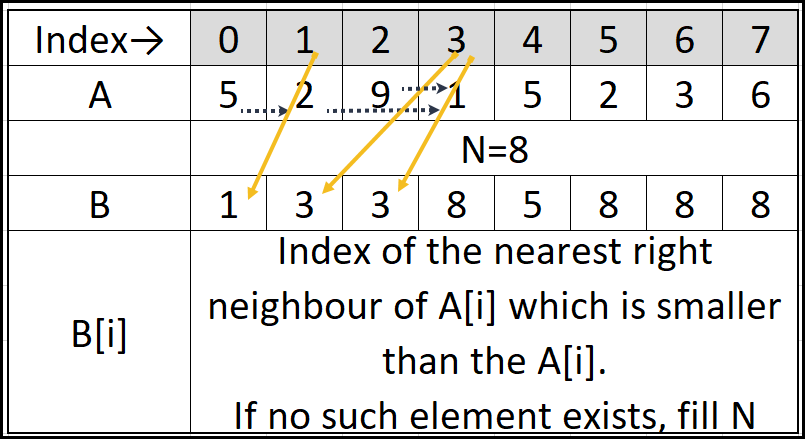
1. NSL (Nearest Smaller to Left)
2. NSR (Nearest Smaller to Right)
3. NGR (Nearest Greater to Right)
4. NGL (Nearest Greater to Left)
5. Largest Rectangular Area of Histogram

## NSL (Nearest Smaller to Left)



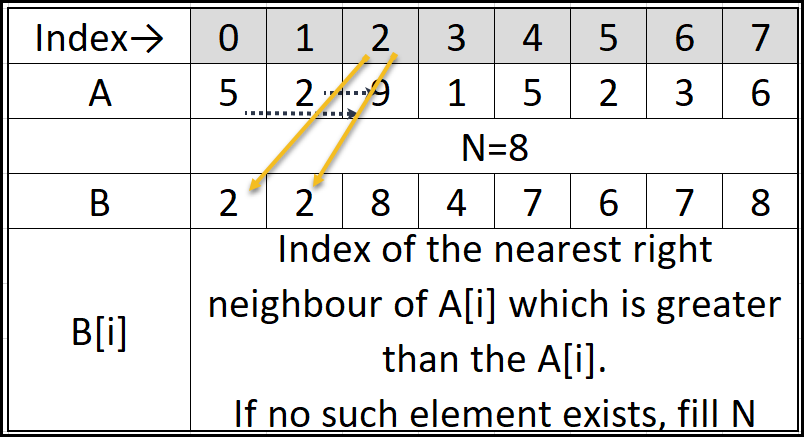
|  |
| --- |
| *vector*<int> nearestSmallerToTheLeft(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk;  for (auto i = len - 1; i >= 0; i--) {  while (!stk.*empty*() && a[stk.*top*()] > a[i]) {  b[stk.*top*()] = i;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = len;  stk.*pop*();  }  return b;  } |

## NSR (Nearest Smaller to Right)



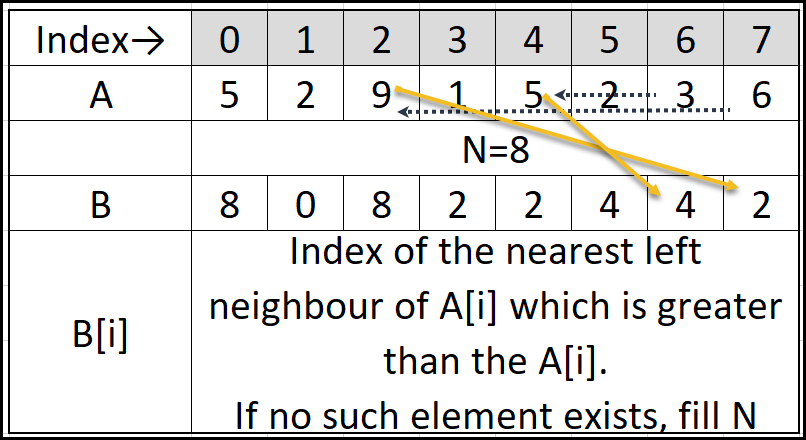
|  |
| --- |
| *vector*<int> nearestSmallerToTheRight(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk;  for (auto i = 0; i < len; i++) {  while (!stk.*empty*() && a[stk.*top*()] > a[i]) {  b[stk.*top*()] = i;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = a.*size*();  stk.*pop*();  }  return b;  } |

## NGR (Nearest Greater to Right)



|  |
| --- |
| *vector*<int> nearestGreaterToTheRight(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk; // Store only indices  for (auto i = 0; i < len; i++) {  while (!stk.*empty*() && a[stk.*top*()] < a[i]) {  b[stk.*top*()] = i;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = a.*size*();  stk.*pop*();  }  return b;  } |

## NGL (Nearest Greater to Left)

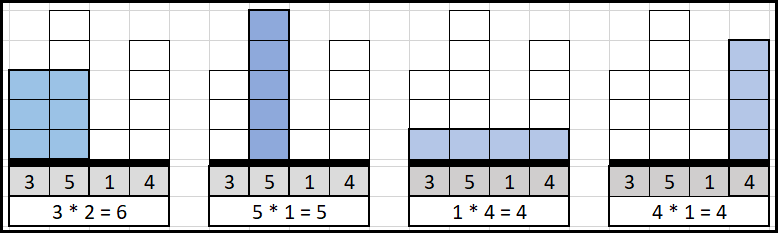


|  |
| --- |
| *vector*<int> nearestGreaterToTheLeft(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk;  for (auto i = len - 1; i >= 0; i--) {  while (!stk.*empty*() && a[stk.*top*()] < a[i]) {  b[stk.*top*()] = i;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = len;  stk.*pop*();  }  return b;  } |

## Largest Rectangular Area of Histogram

#### Observation:

* At least 1 bar is fully included in the largest rectangle.
* If largest rectangle contains at least 1 bar in full, then if we find areas of all largest rectangles for each bar being included full, then we can find the maximum rectangle area.



#### Solution Steps

1. Find largest rectangle including each bar one by one.
   1. Find the nearest left bar with height < current bar.
   2. Find the nearest right bar with height < current bar.
      1. width = (NSR - NSL + 1)
      2. height = current bar
      3. Area = width \* height
2. Take maximum of all the max areas for each bar.

|  |
| --- |
| #include <iostream>  #include <stack>  #include <vector>  using namespace *std*;  *vector*<int> nearestSmallerToTheLeft(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk;  for (auto i = len - 1; i >= 0; i--) {  while (!stk.*empty*() && a[stk.*top*()] > a[i]) {  b[stk.*top*()] = i+1;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = 0;  stk.*pop*();  }  return b;  }  *vector*<int> nearestSmallerToTheRight(*vector*<int> a) {  int len = a.*size*();  *vector*<int> b(len);  *stack*<int> stk;  for (auto i = 0; i < len; i++) {  while (!stk.*empty*() && a[stk.*top*()] > a[i]) {  b[stk.*top*()] = i-1;  stk.*pop*();  }  stk.*push*(i);  }  while (!stk.*empty*()) {  b[stk.*top*()] = a.*size*()-1;  stk.*pop*();  }  return b;  }  int largestHistogramArea(*vector*<int>& a) {  auto NSL = nearestSmallerToTheLeft(a);  //for (auto i : NSL)  // cout << i << " ";  //cout << "\n";  auto NSR = nearestSmallerToTheRight(a);  //for (auto i : NSR)  // cout << i << " ";  //cout << "\n";  int maxArea = (NSR[0] - NSL[0] + 1) \* a[0];  for (auto i = 1; i < a.*size*(); i++) {  auto tempArea = (NSR[i] - NSL[i] + 1)\*a[i];  if (tempArea > maxArea)maxArea = tempArea;  }  return maxArea;  }  int main(void) {  *vector*<int> a = { 2, 1, 5, 6, 2, 3};  *vector*<int> b = { 3, 5, 1, 4 };  *vector*<int> c = { 5, 2, 9, 1, 5, 2, 3, 6 };  *cout* << largestHistogramArea(a) << *endl*;  *cout* << largestHistogramArea(b) << *endl*;  *cout* << largestHistogramArea(c) << *endl*;  return 0;  } |

## Observations and Summary about Applications of Stack/Queue

1. If the inner loop has to examine right elements or left elements for every index.
2. Reversal Problems can be easily done with stack.
3. Undo-Redo
4. Queue - In implicit or explicit TREE/GRAPH for LEVEL ORDER or BREADTH FIRST SEARCH
5. Message Queue / Service Bus [Real applications of queues]