Wednesday, March 16, 2022 5:38 PM

Next Greater Element

- Difficulty Level : Medium
- Last Updated : 02 Dec, 2021

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.

Examples:

1. For an array, the rightmost element always has the next greater element as -1.
2. For an array that is sorted in decreasing order, all elements have the next greater element as -1.
3. For the input array [4, 5, 2, 25], the next greater elements for each element are as follows.

```
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1
```

d) For the input array [13, 7, 6, 12}, the next greater elements for each element are as follows.

```
  Element        NGE
   13      -->    -1
   7       -->     12
   6       -->     12
   12      -->     -1
```

Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by the outer loop. If a greater element is found then that element is printed as next, otherwise, -1 is printed.

Below is the implementation of the above approach:

- C++
- C
- Java
- Python
- C#
- PHP
- Javascript

```java
// Simple Java program to print next
// greater elements in a given array

class Main
{
    /* prints element and NGE pair for
     all elements of arr[] of size n */
    static void printNGE(int arr[], int n)
    {
        int next, i, j;
        for (i=0; i<n; i++)
        {
            next = -1;
            for (j = i+1; j<n; j++)
            {
                if (arr[i] < arr[j])
                {
                    next = arr[j];
                    break;
                }
            }
            System.out.println(arr[i]+" -- "+next);
        }
    }

    public static void main(String args[])
    {
        int arr[]= {11, 13, 21, 3};
        int n = arr.length;
        printNGE(arr, n);
    }
}
```

Output

```
11 -- 13
13 -- 21
21 -- -1
3 -- -1
```

Time Complexity: *O(N_2)*

Auxiliary Space: *O(1)*

Method 2 (Using Stack)

- Push the first element to stack.
- Pick rest of the elements one by one and follow the following steps in loop.

1. Mark the current element as *next*.
2. If stack is not empty, compare top element of stack with *next*.
3. If next is greater than the top element, Pop element from stack. *next* is the next greater element for the popped element.
4. Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements.

- Finally, push the next in the stack.
- After the loop in step 2 is over, pop all the elements from the stack and print -1 as the next element for them.
  Below image is a dry run of the above approach:

  Below is the implementation of the above approach:

- C++
- C
- Java
- Python
- C#
- Javascript

```
// Java program to print next
// greater element using stack

public class NGE {
    static class stack {
        int top;
        int items[] = new int[100];

        // Stack functions to be used by printNGE
        void push(int x)
        {
            if (top == 99)
            {
                System.out.println("Stack full");
            }
            else
            {
                items[++top] = x;
            }
        }

        int pop()
        {
            if (top == -1)
            {
                System.out.println("Underflow error");
                return -1;
            }
            else {
                int element = items[top];
                top--;
                return element;
            }
        }

        boolean isEmpty()
        {
            return (top == -1) ? true : false;
        }
    }
```

```
/* prints element and NGE pair for
   all elements of arr[] of size n */
static void printNGE(int arr[], int n)
{
    int i = 0;
    stack s = new stack();
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (i = 1; i < n; i++)
    {
        next = arr[i];

        if (s.isEmpty() == false)
        {

            // if stack is not empty, then
            // pop an element from stack
            element = s.pop();

            /* If the popped element is smaller than
               next, then a) print the pair b) keep
               popping while elements are smaller and
               stack is not empty */
            while (element < next)
            {
                System.out.println(element + " --> "
                                   + next);
                if (s.isEmpty() == true)
                    break;
                element = s.pop();
            }

            /* If element is greater than next, then
               push the element back */
            if (element > next)
                s.push(element);
        }

        /* push next to stack so that we can find next
           greater for it */
        s.push(next);
    }

    /* After iterating over the loop, the remaining
       elements in stack do not have the next greater
       element, so print -1 for them */
    while (s.isEmpty() == false)
    {
        element = s.pop();
        next = -1;
```

```
                System.out.println(element + " -- " + next);
            }
        }

        // Driver Code
        public static void main(String[] args)
        {
            int arr[] = { 11, 13, 21, 3 };
            int n = arr.length;
            printNGE(arr, n);
        }
    }

    // Thanks to Rishabh Mahrsee for contributing this code
```

Output

```
11 --> 13
13 --> 21
3 --> -1
21 --> -1
```

Time Complexity: *O(N)*

Auxiliary Space: *O(N)*

The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

1. Initially pushed to the stack.
2. Popped from the stack when next element is being processed.
3. Pushed back to the stack because the next element is smaller.
4. Popped from the stack in step 3 of the algorithm.

How to get elements in the same order as input?

The above approach may not produce output elements in the same order as the input. To achieve the same order, we can traverse the same in reverse order

Below is the implementation of the above approach:

- C++
- Java
- C#
- Javascript

```
// A Stack based Java program to find next
// greater element for all array elements
// in same order as input.
import java.util.Stack;

class NextGreaterElement {

    static int arr[] = { 11, 13, 21, 3 };
```

```
    /* prints element and NGE pair for all
    elements of arr[] of size n */
    public static void printNGE()
    {
        Stack<Integer> s = new Stack<>();
        int nge[] = new int[arr.length];

        // iterate for rest of the elements
        for (int i = arr.length - 1; i >= 0; i--)
        {
            /* if stack is not empty, then
            pop an element from stack.
            If the popped element is smaller
            than next, then
            a) print the pair
            b) keep popping while elements are
            smaller and stack is not empty */
            if (!s.empty())
            {
                while (!s.empty()
                       && s.peek() <= arr[i])
                {
                    s.pop();
                }
            }
            nge[i] = s.empty() ? -1 : s.peek();
            s.push(arr[i]);
        }
        for (int i = 0; i < arr.length; i++)
            System.out.println(arr[i] +
                               " --> " + nge[i]);
    }

    /* Driver Code */
    public static void main(String[] args)
    {
        // NextGreaterElement nge = new
        // NextGreaterElement();
        printNGE();
    }
}
```

Output

```
11 ---> 13
13 ---> 21
21 ---> -1
3 ---> -1
```

Time Complexity: *O(N)*

Auxiliary Space: *O(N)*

Method 3:

1. This is same as above method but the elements are pushed and popped only once into

the stack. The array is changed in place. The array elements are pushed into the stack until it finds a greatest element in the right of array. In other words the elements are popped from stack when top of the stack value is smaller in the current array element.

2. Once all the elements are processed in the array but stack is not empty. The left out elements in the stack doesn't encounter any greatest element . So pop the element from stack and change it's index value as -1 in the array.

- Python3

```
# Python3 code
class Solution:
    def nextLargerElement(self,arr,n):
        #code here
        s=[]
        for i in range(len(arr)):
            while s and s[-1].get("value") < arr[i]:
                d = s.pop()
                arr[d.get("ind")] = arr[i]
            s.append({"value": arr[i], "ind": i})
        while s:
            d = s.pop()
            arr[d.get("ind")] = -1
        return arr

if __name__ == "__main__":
    print(Solution().nextLargerElement([6,8,0,1,3],5))
```

Output

```
[8, -1, 1, 3, -1]
```

Time Complexity: *O(N)*

Auxiliary Space: *O(N)*

Please see for an optimized solution for printing in same order.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same proble

From <https://www.geeksforgeeks.org/next-greater-element/>

……/……/……

Replace every element with the least greater element on its right

- Difficulty Level : Hard
- Last Updated : 24 Nov, 2021

Given an array of integers, replace every element with the least greater element on its right side in the array. If there are no greater elements on the right side, replace it with -1.

Examples:

```
Input: [8, 58, 71, 18, 31, 32, 63, 92,
         43, 3, 91, 93, 25, 80, 28]
Output: [18, 63, 80, 25, 32, 43, 80, 93,
         80, 25, 93, -1, 28, -1, -1]
```

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

A naive method is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the smallest element greater than the picked element on its right side. Finally, the outer loop will replace the picked element with the element found by inner loop. The time complexity of this method will be $O(n^2)$.

A tricky solution would be to use Binary Search Trees. We start scanning the array from right to left and insert each element into the BST. For each inserted element, we replace it in the array by its inorder successor in BST. If the element inserted is the maximum so far (i.e. its inorder successor doesn't exist), we replace it by -1.

Below is the implementation of the above idea –

- C++
- Java
- Python3
- C#
- Javascript

```java
// Java program to replace every element with
// the least greater element on its right
import java.io.*;

class BinarySearchTree{

// A binary Tree node
class Node
{
    int data;
    Node left, right;

    Node(int d)
    {
        data = d;
        left = right = null;
    }
}

// Root of BST
static Node root;
```

```
static Node succ;

// Constructor
BinarySearchTree()
{
    root = null;
    succ = null;
}

// A utility function to insert a new node with
// given data in BST and find its successor
Node insert(Node node, int data)
{

    // If the tree is empty, return a new node
    if (node == null)
    {
        node = new Node(data);
    }

    // If key is smaller than root's key,
    // go to left subtree and set successor
    // as current node
    if (data < node.data)
    {
        succ = node;
        node.left = insert(node.left, data);
    }

    // Go to right subtree
    else if (data > node.data)
        node.right = insert(node.right, data);

    return node;
}

// Function to replace every element with the
// least greater element on its right
static void replace(int arr[], int n)
{
    BinarySearchTree tree = new BinarySearchTree();

    // start from right to left
    for(int i = n - 1; i >= 0; i--)
    {
        succ = null;

        // Insert current element into BST and
        // find its inorder successor
        root = tree.insert(root, arr[i]);

        // Replace element by its inorder
        // successor in BST
        if (succ != null)
            arr[i] = succ.data;
```

```
        // No inorder successor
        else
            arr[i] = -1;
    }
}

// Driver code
public static void main(String[] args)
{
    int arr[] = new int[] { 8, 58, 71, 18, 31,
                            32, 63, 92, 43, 3,
                            91, 93, 25, 80, 28 };
    int n = arr.length;

    replace(arr, n);

    for(int i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
}
}

// The code is contributed by Tushar Bansal
```

Output

```
18 63 80 25 32 43 80 93 80 25 93 -1 28 -1 -1
```

The **worst-case time complexity** of the above solution is also O(n^2) as it uses BST. The worst-case will happen when array is sorted in ascending or descending order. The complexity can easily be reduced to O(nlogn) by using balanced trees like red-black trees.

Another Approach:

We can use the **Next Greater Element using stack** algorithm to solve this problem in **O(Nlog(N))** time and **O(N)** space.

Algorithm:

5. *First, we take an array of pairs namely temp, and store each element and its index in this array,i.e.* ***temp[i] will be storing {arr[i],i}****.*
6. ***Sort the array*** *according to the array elements.*
7. *Now get the next greater index for each and every index of the temp array in an array namely index by using* ***Next Greater Element*** *using stack.*
8. *Now index[i] stores the index of the next least greater element of the element temp[i].first and if index[i] is -1, then it means that there is no least greater element of the element temp[i].second at its right side.*
9. *Now take a result array where result[i] will be equal to* ***a[indexes[temp[i].second]]*** *if index[i] is not -1 otherwise result[i] will be equal to -1.*

Below is the implementation of the above approach

- C++

```
#include <bits/stdc++.h>
using namespace std;
// function to get the next least greater index for each and
// every temp.second of the temp array using stack this
// function is similar to the Next Greater element for each
```

```
// and every element of an array using stack difference is
// we are finding the next greater index not value and the
// indexes are stored in the temp[i].second for all i
vector<int> nextGreaterIndex(vector<pair<int, int> >& temp)
{
    int n = temp.size();
    // initially result[i] for all i is -1
    vector<int> res(n, -1);
    stack<int> stack;
    for (int i = 0; i < n; i++) {
        // if the stack is empty or this index is smaller
        // than the index stored at top of the stack then we
        // push this index to the stack
        if (stack.empty() || temp[i].second < stack.top())
            stack.push(
                temp[i].second); // notice temp[i].second is
                                 // the index
        // else this index (i.e. temp[i].second) is greater
        // than the index stored at top of the stack we pop
        // all the indexes stored at stack's top and for all
        // these indexes we make this index i.e.
        // temp[i].second as their next greater index
        else {
            while (!stack.empty()
                   && temp[i].second > stack.top()) {
                res[stack.top()] = temp[i].second;
                stack.pop();
            }
            // after that push the current index to the stack
            stack.push(temp[i].second);
        }
    }
    // now res will store the next least greater indexes for
    // each and every indexes stored at temp[i].second for
    // all i
    return res;
}
// now we will be using above function for finding the next
// greater index for each and every indexes stored at
// temp[i].second
vector<int> replaceByLeastGreaterUsingStack(int arr[],
                                            int n)
{
    // first of all in temp we store the pairs of {arr[i].i}
    vector<pair<int, int> > temp;
    for (int i = 0; i < n; i++) {
        temp.push_back({ arr[i], i });
    }
    // we sort the temp according to the first of the pair
    // i.e value
    sort(temp.begin(), temp.end());
    // now indexes vector will store the next greater index
    // for each temp[i].second index
    vector<int> indexes = nextGreaterIndex(temp);
    // we initialize a result vector with all -1
```

```
    vector<int> res(n, -1);
    for (int i = 0; i < n; i++) {
        // now if there is no next greater index after the
        // index temp[i].second the result will be -1
        // otherwise the result will be the element of the
        // array arr at index indexes[temp[i].second]
        if (indexes[temp[i].second] != -1)
            res[temp[i].second]
                = arr[indexes[temp[i].second]];
    }
    // return the res which will store the least greater
    // element of each and every element in the array at its
    // right side
    return res;
}
// driver code
int main()
{
    int arr[] = { 8,  58, 71, 18, 31, 32, 63, 92,
                  43, 3,  91, 93, 25, 80, 28 };
    int n = sizeof(arr) / sizeof(int);
    auto res = replaceByLeastGreaterUsingStack(arr, n);
    cout << "Least Greater elements on the right side are "
         << endl;
    for (int i : res)
        cout << i << ' ';
    cout << endl;
    return 0;
} // this code is contributed by Dipti Prakash Sinha
```

Output

```
Least Greater elements on the right side are
18 63 80 25 32 43 80 93 80 25 93 -1 28 -1 -1
```

Another approach with set

A different way to think about the problem is listing our requirements and then thinking over it to find a solution. If we traverse the array from backwards, we need a data structure(ds) to support:

1. Insert an element into our ds in sorted order (so at any point of time the elements in our ds are sorted)

2. Finding the upper bound of the current element (upper bound will give just greater element from our ds if present)

Carefully observing at our requirements, a set is what comes in mind.

Why not multiset? Well we can use a multiset but there is no need to store an element more than once.

Let’s code our approach

Time and space complexity: We insert each element in our set and find upper bound for each element using a loop so its time complexity is O(n*log(n)). We are storing each element in our set so space complexity is O(n)

- C++

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

using namespace std;

void solve(vector<int>& arr) {
    set<int> s;
    vector<int> ans(arr.size());
    for(int i=arr.size()-1;i>=0;i--) { //traversing the array backwards
        s.insert(arr[i]); // inserting the element into set
        auto it = s.upper_bound(arr[i]); // finding upper bound
        if(it == s.end()) arr[i] = -1; // if upper_bound does not exist
then -1
        else arr[i] = *it; // if upper_bound exists, lets take it
    }
}

void printArray(vector<int>& arr) {
    for(int i : arr) cout << i << " ";
    cout << "\n";
}

int main() {
    vector<int> arr = {8, 58, 71, 18, 31, 32, 63, 92, 43, 3, 91, 93, 25, 80,
28};
    printArray(arr);
    solve(arr);
    printArray(arr);
    return 0;
}
```

Output

```
8 58 71 18 31 32 63 92 43 3 91 93 25 80 28
18 63 80 25 32 43 80 93 80 25 93 -1 28 -1 -1
```

From <https://www.geeksforgeeks.org/replace-every-element-with-the-least-greater-element-on-its-right/>