תכונות סגירות של שפות

	שפות רגולריות	שפות חסרות הקשר
חלקיות \ הכלה	X	X
משלים	V	X
חיתוך	V	X
איחוד	V	V
היפוך	V	V
שרשור	V	V

פעולות על string

Properties	Description
Clone()	Make clone of string.
CompareTo()	Compares two specified String objects and returns an integer that indicates their relative position in the sort order.
Contains()	Returns a value indicating whether a specified substring occurs within this string.
EndsWith()	Determines whether the end of this string instance matches the specified string.
Equals()	Determines whether this instance and another specified String object have the same value
IndexOf(String)	Reports the zero-based index of the first occurrence of the specified string in this instance.
ToLower()	Returns a copy of this string converted to lowercase.
ToUpper()	Returns a copy of this string converted to uppercase.
Insert()	Returns a new string in which a specified string is inserted at a specified index position in this instance.
LastIndexOf(String)	Reports the zero-based index position of the last occurrence of a specified string within this instance.
Remove()	This method deletes all the characters from beginning to specified index position.
Replace()	This method helps to replace the character.
Split()	This method splits the string based on specified value.
StartsWith(String)	Determines whether the beginning of this string instance matches the specified string.
Substring()	Retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string.
Trim()	It removes extra whitespaces from beginning and ending of string.

Random

```
Random rnd = new Random();
int num = rnd.Next(); // int as param -> 0 < random < x
```

Node Utils Section

```
using System;
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Collections.Generic;
using System.Runtime.ExceptionServices;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Security.Policy;

namespace Unit4
{
    public static class NodeUtils
    {
        public static Node<T> CreateListFromArray<T>(T[] arr)
        {
            Node<T> head = null;

            for (int i = arr.Length-1; i >= 0; i--)
            {
                head = new Node<T>(arr[i], head);
            }

            return head;
        }

        public static Node<T> CreateListFromArrayR<T>(T[] arr, int i)
        {
            Node<T> head = new Node<T>(arr[i]);

            if (i <= 0)
                return head;

            head = new Node<T>(arr[i - 1], head);

            return CreateListFromArrayR(arr, i - 1);
        }

        public static void PrintList<T>(Node<T> l)
        {
            while(l != null)
            {
                Console.Write(l.GetValue() + "-->");
                l = l.GetNext();
            }
            Console.WriteLine();
        }

        public static void PrintListR<T>(Node<T> l)
        {
            if (l == null)
```

```
                return;

            Console.Write(l + "-->");

            PrintList(l.GetNext());
        }

        public static Node<T> CloneList<T>(Node<T> l)
        {
            if (l == null)
                return null;

            Node<T> new_head = new Node<T>(l.GetValue());
            Node<T> pos = new_head;

            while(l.GetNext() != null)
            {
                l = l.GetNext();
                pos.SetNext(new Node<T>(l.GetValue()));
                pos = pos.GetNext();
            }

            return new_head;
        }

        public static bool CompareList(Node<int> left, Node<int> right)
        {
            bool equal = true;

            while(left != null && right != null && equal)
            {
                if ((left.GetNext() == null && right.GetNext() != null) || (left.GetNext() != null && right.GetNext() == null))
                    equal = false;

                equal = left.GetValue() == right.GetValue() && equal;

                left = left.GetNext();
                right = right.GetNext();
            }

            return equal;
        }

        public static bool CompareListR(Node<int> left, Node<int> right)
        {
            if (left == null && right == null)
                return true;

```

```
            if ((left.GetNext() == null && right.GetNext() != null) ||
              (left.GetNext() != null && right.GetNext() == null))
                return false;

            return left.GetValue() == right.GetValue() && CompareListR
              (left.GetNext(), right.GetNext());
        }

        public static int CountList<T>(Node<T> lst)
        {
            int cnt = 0;

            while(lst != null)
            {
                cnt++;
                lst = lst.GetNext();
            }

            return cnt;
        }

        public static int CountListR<T>(Node<T> lst)
        {
            if (lst == null)
                return 0;

            return 1 + CountListR(lst.GetNext());
        }

        public static int SumList(Node<int> lst)
        {
            int sum = 0;

            while (lst != null)
            {
                sum += lst.GetValue();
                lst = lst.GetNext();
            }

            return sum;
        }

        public static int SumListR(Node<int> lst)
        {
            if (lst == null)
                return 0;

            return lst.GetValue() + SumListR(lst.GetNext());
        }
```

```
        public static bool IsExist(Node<int> lst, int val)
        {
            bool found = false;

            while (!found && lst != null)
            {
                found = lst.GetValue() == val;
                lst = lst.GetNext();
            }

            return found;
        }

        public static bool IsExistR(Node<int> lst, int val)
        {
            if (lst == null)
                return false;

            return lst.GetValue() == val || IsExistR(lst.GetNext(), val);
        }

        public static int FindMax(Node<int> lst)
        {
            int max = int.MinValue;

            while(lst != null)
            {
                if(lst.GetValue() > max)
                {
                    max = lst.GetValue();
                }
                lst = lst.GetNext();
            }

            return max;
        }

        public static int FindMaxR(Node<int> lst)
        {
            if (lst == null)
                return int.MinValue;

            return Math.Max(lst.GetValue(), FindMaxR(lst.GetNext()));
        }

        public static void AbsValue(Node<int> lst)
        {
            while(lst != null)
```

```
            {
                lst.SetValue(Math.Abs(lst.GetValue()));
                lst = lst.GetNext();
            }
        }

        public static void AbsValueR(Node<int> lst)
        {
            if (lst == null)
                return;

            lst.SetValue(Math.Abs(lst.GetValue()));

            AbsValue(lst.GetNext());
        }

        public static Node<T> GetNodeRef<T>(Node<T> lst, int loc)
        {
            Node<T> r = null; // refernce

            int save_i = 1;

            for (int i = 1; i <= loc && lst != null; i++)
            {
                r = lst;
                save_i = i;
                lst = lst.GetNext();
            }

            if (save_i < loc)
                return null;
            return r;
        }

        public static Node<T> GetNodeRefR<T>(Node<T> lst, int loc)
        {
            if (loc == 0)
                return lst;

            if (lst == null)
                return null;

            return GetNodeRef<T>(lst.GetNext(), loc - 1);
        }

        public static bool IsSorted(Node<int> lst)
        {
            bool is_sorted = true;

```

```
            while (lst.GetNext() != null && is_sorted)
            {
                is_sorted = lst.GetValue() <= lst.GetNext().GetValue();
                lst = lst.GetNext();
            }

            return is_sorted;

        }

        public static bool IsSortedR(Node<int> lst)
        {
            if (lst.GetNext() == null)
                return true;

            return lst.GetValue() <= lst.GetNext().GetValue() && IsSortedR (lst.GetNext());
        }

        public static int CountSeqs(Node<int> lst, int n)
        {
            int seq_cnt = 0;
            int curr_seq_len = 1;

            while(lst != null)
            {
                if (lst.GetValue() ==  n)
                {
                    if (curr_seq_len == 1)
                        seq_cnt++;

                    curr_seq_len++;
                }

                else
                {
                    curr_seq_len = 1;
                }

                lst = lst.GetNext();
            }

            return seq_cnt;
        }

        public static void PrintAtoB(Node<int> lst, int a, int b)
        {
            for (int i = 0; i < a-1; i++)
            {
```

```
                lst = lst.GetNext();
            }

            for (int i = 0; i < b-a+1; i++)
            {
                Console.Write(lst + "-->");
                lst = lst.GetNext();
            }
        }

        public static Node<int> RemoveDuplicates(Node<int> lst)
        {
            Node<int> new_lst = new Node<int>(lst.GetValue());
            lst = lst.GetNext();
            Node<int> head = new_lst;

            while(lst != null)
            {
                if (!IsExist(head, lst.GetValue()))
                {
                    new_lst.SetNext(new Node<int>(lst.GetValue()));
                    new_lst = new_lst.GetNext();
                }

                lst = lst.GetNext();
            }

            return head;
        }

        public static bool BalancedList(Node<int> lst)
        {
            int sum = 0;
            int cnt = 0;

            Node<int> save_lst = lst;

            while(lst != null)
            {
                sum += lst.GetValue();
                cnt++;
                lst = lst.GetNext();
            }

            double avg = (double)(sum) / cnt;

            lst = save_lst;

            int aboveAvg = 0;
```

```
            int belowAvg = 0;

            while(lst != null)
            {
                if (lst.GetValue() > avg)
                    aboveAvg++;

                else if (lst.GetValue() < avg)
                    belowAvg++;

                lst = lst.GetNext();
            }

            return aboveAvg == belowAvg;

        }

        public static (Node<int>, Node<int>) RemoveMax(Node<int> lst)
        {
            Node<int> save_lst = lst;

            Node<int> maxNode = lst;
            Node<int> previousMaxNode = null;
            int max = lst.GetValue();
            previousMaxNode = null;


            while(lst.GetNext() != null)
            {

                if (lst.GetNext().GetValue() > max)
                {
                    maxNode = lst.GetNext();
                    max = lst.GetNext().GetValue();
                    previousMaxNode = lst;
                }

                lst = lst.GetNext();
            }

            if (previousMaxNode == null)
                lst = maxNode.GetNext();

            else
            {
                previousMaxNode.SetNext(maxNode.GetNext());
                maxNode.SetNext(null);
                lst = save_lst;
            }
```

```
            return (maxNode, lst);
        }

        public static Node<int> InsertToSortedList(Node<int> lst, Node<int> node)
        {
            Node<int> save_lst = lst;
            Node<int> previous = null;

            while (lst != null && lst.GetValue() < node.GetValue())
            {
                previous = lst;
                lst = lst.GetNext();
            }

            if (previous == null)
                lst = new Node<int>(node.GetValue(), lst);

            else
            {
                node.SetNext(lst);
                previous.SetNext(node);
                lst = save_lst;
            }

            return lst;

        }

        public static Node<int> Sort(Node<int> lst)
        {
            Node<int> sorted = new Node<int>(lst.GetValue());

            while(lst.GetNext() != null)
            {
                lst = lst.GetNext();
                Node<int> new_node = new Node<int>(lst.GetValue());
                sorted = InsertToSortedList(sorted, new_node);
            }

            return sorted;
        }

        public static Node<T> ReverseList<T>(Node<T> lst)
        {
            Node<T> curr = lst;
            Node<T> next = lst.GetNext();
            Node<T> next_next = lst.GetNext().GetNext();

```

```
            while(next != null)
            {
                next.SetNext(curr);
                curr = next;
                next = next_next;
                if (next_next != null)
                    next_next = next_next.GetNext();
            }

            lst.SetNext(null);

            return curr;
        }

        public static Node<T> ReverseListR<T>(Node<T> lst)
        {
            if (lst.GetNext() == null)
                return lst;

            Node<T> reversed = ReverseListR(lst.GetNext());
            lst.GetNext().SetNext(lst);
            lst.SetNext(null);

            return reversed;
        }

        public static int NegativeSequence(Node<int> lst)
        {
            int max_seq_len = 0;
            int curr_seq_len = 1;

            while(lst.GetNext() != null)
            {
                if (lst.GetValue() < 0 && lst.GetNext().GetValue() < 0)
                {
                    curr_seq_len++;
                }

                else if (curr_seq_len > 1)
                {
                    if (curr_seq_len > max_seq_len)
                        max_seq_len = curr_seq_len;
                    curr_seq_len = 1;
                }

                lst = lst.GetNext();

            }
```

```

            return max_seq_len;
        }

        public static Node<int> MergeSortedLists(Node<int> l1, Node<int> l2)
        {

            Node<int> merged = new Node<int>(0);
            Node<int> head = null;
            Node<int> last = null;

            while (l1 != null && l2 != null)
            {
                if (l1.GetValue() < l2.GetValue())
                {
                    merged.SetValue(l1.GetValue());
                    if (head == null)
                        head = merged;
                    l1 = l1.GetNext();
                }

                else
                {
                    merged.SetValue(l2.GetValue());
                    if (head == null)
                        head = merged;
                    l2 = l2.GetNext();
                }

                merged.SetNext(new Node<int>(0));
                last = merged;
                merged = merged.GetNext();

            }

            if (l1 != null)
                last.SetNext(l1);

            else if (l2 != null)
                last.SetNext(l2);

            return head;
        }

        public static Node<int> MergeSortedLists2(Node<int> l1, Node<int> l2)
        {
            Node<int> next_l1, next_l2 = null;
```

```
            Node<int> head = null;

            while(l1 != null && l2 != null)
            {
                if (l1.GetValue() > l2.GetValue())
                {
                    if (head == null)
                        head = l2;

                    next_l2 = l2.GetNext();
                    l2.SetNext(l1);
                    l2 = next_l2;
                }

                else
                {
                    if (head == null)
                        head = l1;

                    next_l1 = l1.GetNext();
                    l1.SetNext(l2);
                    l1 = next_l1;
                }
            }


            return head;
        }

        public static void AddFollowingNum(Node<int> lst, int val)
        {
            while(lst.GetValue() != val)
            {
                lst = lst.GetNext();
            }

            lst.SetNext(new Node<int>(val + 1, lst.GetNext()));
        }

        public static Node<int> ListsIntersection(Node<int> l1, Node<int> l2)
        {
            Node<int> head = null;
            Node<int> save_head = null;


            while(l1 != null)
            {
                if (IsExist(l2, l1.GetValue()))
```

```
                {
                    if (head == null)
                        head = new Node<int>(l1.GetValue());
                    else
                    {
                        head.SetNext(new Node<int>(l1.GetValue()));
                        head = head.GetNext();
                    }

                    if (save_head == null)
                        save_head = head;
                }

                l1 = l1.GetNext();
            }

            return save_head;
        }

        public static bool Compare(Node<int> left, Node<int> right)
        {
            bool equals = true;

            while(left.GetNext() != null && right.GetNext() != null)
            {
                equals = left.GetValue() == right.GetValue();

                left = left.GetNext();
                right = right.GetNext();
            }

            return equals;
        }

        public static Node<T> GetMiddle<T>(Node<T> lst)
        {
            Node<T> slow = lst;
            Node<T> fast = lst;

            while(fast.GetNext() != null)
            {
                slow = slow.GetNext();
                fast = fast.GetNext().GetNext();
            }

            return slow;
        }

        public static bool IsPalindrome(Node<char> lst)
```

```
        {
            Node<char> rlst = ReverseList(GetMiddle(lst));

            bool pali = true;
            while (pali && lst != rlst && lst != null)
            {
                pali = lst.GetValue() == rlst.GetValue();
                lst = lst.GetNext();
                rlst = rlst.GetNext();
            }

            return pali;

        }
    }
}

```

Round Node Utils Section

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Unit4
8  {
9      class RoundNodeUtils
10     {
11         public static void MakeListRound<T>(Node<T> lst)
12         {
13             Node<T> head = lst;
14
15             while (lst.GetNext() != null)
16             {
17                 lst = lst.GetNext();
18             }
19
20             lst.SetNext(head);
21         }
22
23         public static void PrintRoundList<T>(Node<T> lst)
24         {
25
26             Node<T> pos = lst.GetNext();
27
28             Console.Write(lst + "-->");
29
30             while (pos != lst)
31             {
32                 Console.Write(pos + "-->");
33                 pos = pos.GetNext();
34             }
35         }
36
37         public static void DisconnectRoundList<T>(Node<T> lst)
38         {
39             Node<T> pos = lst.GetNext();
40
41             while (pos.GetNext() != lst)
42             {
43                 pos = pos.GetNext();
44             }
45
46             pos.SetNext(null);
47         }
48
49         public static bool IsRoundList<T>(Node<T> head)
```

```
        {
            Node<T> pos = head.GetNext();

            while (pos != head && pos != null)
            {
                pos = pos.GetNext();
            }

            return pos == head;
        }

        public static int ListLength<T>(Node<T> head)
        {
            Node<T> pos = head.GetNext();
            int cnt = 1;

            while (pos != head)
            {
                pos = pos.GetNext();
                cnt++;
            }

            return cnt;
        }

        public static int SumList(Node<int> head)
        {
            Node<int> pos = head.GetNext();

            int sum = head.GetValue();

            while(pos != head)
            {
                sum += pos.GetValue();
                pos = pos.GetNext();
            }

            return sum;
        }

        public static Node<T> RemoveHead<T>(Node<T> head)
        {
            Node<T> pos = head;

            while(pos.GetNext() != head)
            {
                pos = pos.GetNext();
            }

```

```
            pos.SetNext(head.GetNext());
            return pos.GetNext();
        }

        public static void RemoveLast<T>(Node<T> head)
        {
            Node<T> pos = head;

            while (pos.GetNext().GetNext() != head)
            {
                pos = pos.GetNext();
            }

            pos.SetNext(head);
        }

        public static bool IsExist(Node<int> head, int value)
        {
            Node<int> pos = head.GetNext();

            bool found = false;

            while (pos != head && !found)
            {
                found = pos.GetValue() == value;
                pos = pos.GetNext();
            }

            return found;
        }

        public static Node<int> RemoveEven(Node<int> head)
        {
            Node<int> last = head;
            Node<int> pos = head.GetNext();

            while (pos != head)
            {
                if (pos.GetValue() % 2 == 0)
                {
                    last.SetNext(pos.GetNext());
                }

                else
                    last = pos;

                pos = pos.GetNext();
            }

```

```
            if (pos.GetValue() % 2 == 0)
            {
                last.SetNext(pos.GetNext());
                return pos.GetNext();
            }

            return head;
        }

        public static void AddToEven(Node<int> head)
        {
            Node<int> last = head;
            Node<int> pos = head.GetNext();

            while(pos != head)
            {
                if (pos.GetValue() % 2 == 0)
                {
                    last.SetNext(new Node<int>(pos.GetValue() - 1));
                    last.GetNext().SetNext(pos);
                    last = pos;
                }

                else
                    last = pos;

                pos = pos.GetNext();
            }

            if (pos.GetValue() % 2 == 0)
            {
                last.SetNext(new Node<int>(pos.GetValue() - 1));
                last.GetNext().SetNext(pos);
            }

        }

        public static Node<T> AddToLoop<T>(Node<T> head, Node<T> new_node)
        {
            Node<T> pos = head.GetNext();

            while (pos.GetNext() != head)
                pos = pos.GetNext();

            pos.SetNext(new_node);
            new_node.SetNext(head);

            return new_node; // new_node => new list head
        }
```

```
        public static void SumNeighbors(Node<int> head)
        {
            Node<int> last = head;
            Node<int> pos = head.GetNext();

            while(pos != head)
            {
                AddToLoop(pos, new Node<int>(last.GetValue() + pos.GetValue()));
                last = pos;
                pos = pos.GetNext();
            }
            AddToLoop(pos, new Node<int>(last.GetValue() + pos.GetValue()));
        }

        public static bool HasLoop<T>(Node<T> head)
        {
            Node<T> curr = head;
            Node<T> next = head.GetNext();
            Node<T> next_next = next.GetNext();

            while (next != null && curr != next_next)

            {
                next.SetNext(curr);

                curr = next;
                next = next_next;
                if (next_next != null)
                    next_next = next_next.GetNext();
            }

            bool foundLoop = curr == next_next;

            return foundLoop;
        }

        public static void CreateLoopList<T>(Node<T> lst, int n)
        {
            Node<T> pos = lst;
            for (int i = 0; i < n; i++)
            {
                pos = pos.GetNext();
            }

            while (lst.GetNext() != null)
                lst = lst.GetNext();
```

```
            lst.SetNext(pos);
        }

        public static Node<T> IntersectionPoint<T>(Node<T> rLst)
        {
            Node<T> turtle = rLst;
            Node<T> rabbit = rLst;

            bool found_intersect = true;

            while (rabbit.GetNext().GetNext() != null && found_intersect)

                for (int i = 0; i < 2; i++)
                    rabbit = rabbit.GetNext();

                turtle = turtle.GetNext();

                found_intersect = rabbit != turtle;
            }

            if (rabbit == turtle)
                return rabbit;

            return null;
        }

        public static Node<T> CrossSection<T>(Node<T> lst)
        {
            Node<T> intersect = IntersectionPoint(lst);

            while (intersect != lst)
            {
                intersect = intersect.GetNext();
                lst = lst.GetNext();
            }

            return intersect;
        }

        public static void PrintLoopRoundList<T>(Node<T> lst)
        {
            Node<T> intersect = IntersectionPoint(lst);
            while(lst != intersect)
            {
                Console.Write(lst.GetValue() + "-->");
                lst = lst.GetNext();
            }

```

```
            Console.Write("(");
            Console.Write(lst + "-->");
            lst = lst.GetNext();

            while(lst != intersect)
            {
                Console.Write(lst + "-->");
                lst = lst.GetNext();
            }

            Console.Write(")");
        }
    }
}

```

Fork Utils Section

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Unit4
{
    class ForkNodeUtils
    {
        public static void CreateForkList<T>(Node<T> ls1, Node<T> ls2, int n)
        {
            while(ls2.GetNext() != null)
                ls2 = ls2.GetNext();

            for (int i = 0; i < n-1; i++)
                ls1 = ls1.GetNext();

            ls2.SetNext(ls1);
        }

        public static Node<T> GetForkMeeting<T>(Node<T> ls1, Node<T> ls2)
        {
            int size_ls1 = NodeUtils.CountList(ls1);
            int size_ls2 = NodeUtils.CountList(ls2);

            int diff = Math.Abs(size_ls2 - size_ls1);

            if (size_ls1 > size_ls2)
            {
                for (int i = 0; i < diff; i++)
                    ls1 = ls1.GetNext();
            }

            else
            {
                for (int i = 0; i < diff; i++)
                    ls2 = ls2.GetNext();
            }

            while(ls1 != ls2 && ls1 != null && ls2 != null)
            {
                ls1 = ls1.GetNext();
                ls2 = ls2.GetNext();
            }

            if (ls1 == null || ls2 == null)
                return null;
```

```
            return ls1;
        }

        public static Node<T> MergeFrokList<T>(Node<T> l1, Node<T> l2)
        {
            Node<T> merge = GetForkMeeting(l1, l2);

            int size_l1 = NodeUtils.CountList(l1);
            int size_l2 = NodeUtils.CountList(l2);

            Node<T> head = size_l1 > size_l2 ? l1 : l2;
            Node<T> pos = head;

            while (pos.GetNext() != null)
                pos = pos.GetNext();

            Node<T> conn = size_l1 > size_l2 ? l2 : l1;

            pos.SetNext(conn);

            while (conn.GetNext() != merge)
                conn = conn.GetNext();

            conn.SetNext(null);


            return head;
        }

    }
}

```

Queue Utils Section

```
1  using System;
2  using System.CodeDom.Compiler;
3  using System.Collections.Generic;
4  using System.Data.SqlClient;
5  using System.Runtime.Remoting.Messaging;
6  using System.Security;
7
8  namespace Unit4
9  {
10     class QueueUtils
11     {
12         public static Queue<T> CreateQueueFromArray<T>(T[] arr)
13         {
14             Queue<T> s = new Queue<T>();
15
16             for (int i = 0; i < arr.Length; i++)
17             {
18                 s.Insert(arr[i]);
19             }
20
21             return s;
22         }
23         public static void SpilledOn<T>(Queue<T> dest, Queue<T> src)
24         {
25             while (!src.IsEmpty())
26             {
27                 T temp = src.Remove();
28                 dest.Insert(temp);
29             }
30         }
31
32         public static Queue<T> Clone<T>(Queue<T> s)
33         {
34             Queue<T> t = new Queue<T>();
35             Queue<T> s2 = new Queue<T>();
36
37             SpilledOn(t, s);
38
39             while(!t.IsEmpty())
40             {
41                 T temp = t.Remove();
42                 s.Insert(temp);
43                 s2.Insert(temp);
44             }
45
46             return s2;
47         }
48         public static int GetSize<T>(Queue<T> q)
49         {
```

```
            int count = 0;

            Queue<T> temp = Clone(q);
            while (!temp.IsEmpty())
            {
                temp.Remove();
                count++;
            }

            return count;
        }

        public static int GetSum(Queue<int> q)
        {
            int sum = 0;

            Queue<int> temp = Clone(q);
            while (!temp.IsEmpty())
            {

                sum+=temp.Remove();
            }

            return sum;
        }

        public static bool IsExist<T>(Queue<T> q, T e)
        {
            Queue<T> temp = Clone(q);

            while (!temp.IsEmpty())
            {
                if (EqualityComparer<T>.Default.Equals(temp.Remove(), e))
                    return true;
            }
            return false;
        }

        public static void LastToFirst<T>(Queue<T> q)
        {
            Queue<T> temp = new Queue<T>();

            int l = GetSize(q) - 1;

            for (int i = 0; i < l; i++)
            {
                temp.Insert(q.Remove());
            }

```

```
            while(!temp.IsEmpty())
            {
                q.Insert(temp.Remove());
            }
        }

        public static bool IsSorted(Queue<int> q)
        {
            Queue<int> temp = Clone(q);

            bool sorted = true;
            int last = q.Head();
            while(!temp.IsEmpty() && sorted)
            {
                int curr = temp.Remove();
                sorted = last <= curr;
                last = curr;
            }

            return sorted;
        }

        public static void InsertToSorted(Queue<int> q,  int val)
        {
            Queue<int> temp = new Queue<int>();

            bool found = false;

            while (!q.IsEmpty())
            {
                int curr = q.Remove();
                if (val < curr && !found)
                {
                    temp.Insert(val);
                    temp.Insert(curr);
                    found = true;
                }
                else
                {
                    temp.Insert(curr);

                }

            }

            if (!found)
            {
                temp.Insert(val);
            }
```

```

            SpilledOn(q, temp);
        }

        public static int FindMin(Queue<int> q)
        {
            Queue<int> temp = Clone(q);

            int min = int.MaxValue;

            while(!temp.IsEmpty())
            {
                int val = temp.Remove();
                if (val < min)
                    min = val;
            }

            return min;
        }

        public static int FindMax(Queue<int> q)
        {
            Queue<int> temp = Clone(q);

            int max = int.MinValue;

            while (!temp.IsEmpty())
            {
                int val = temp.Remove();
                if (val > max)
                    max = val;
            }

            return max;
        }

        public static void RemoveMin(Queue<int> q)
        {
            Queue<int> temp = Clone(q);

            int min_index = 0;
            int min = int.MaxValue;
            int cnt = 0;

            while (!temp.IsEmpty())
            {
                int val = temp.Remove();
                if (val < min)
                {
```

```
                    min = val;
                    min_index = cnt;
                }
                cnt++;
            }

            SpilledOn(temp, q);

            int l = GetSize(temp);

            for (int i = 0; i < l; i++)
            {
                int val = temp.Remove();
                if (i != min_index)
                {
                    q.Insert(val);
                }
            }

        }

        public static void RemoveMax(Queue<int> q)
        {
            Queue<int> temp = Clone(q);

            int max_index = 0;
            int max = int.MinValue;
            int cnt = 0;

            while (!temp.IsEmpty())
            {
                int val = temp.Remove();
                if (val > max)
                {
                    max = val;
                    max_index = cnt;
                }
                cnt++;
            }

            SpilledOn(temp, q);

            int l = GetSize(temp);

            for (int i = 0; i < l; i++)
            {
                int val = temp.Remove();
                if (i != max_index)
                {
```

```
                    q.Insert(val);
                }
            }
        }

        public static void SortQueue(Queue<int> q)
        {
            int l = GetSize(q);
            Queue<int> temp = new Queue<int>();

            for (int i = 0; i < l; i++)
            {
                int val = FindMin(q);
                temp.Insert(val);
                RemoveMin(q);
            }

            SpilledOn(q, temp);
        }

        public static void Reverse<T>(Queue<T> q)
        {
            int l = GetSize(q);

            Queue<T> new_q = new Queue<T>();
            Queue<T> save_q = new Queue<T>();
            SpilledOn(save_q, q);

            for (int i = 0; i < l; i++)
            {
                Queue<T> temp = Clone(save_q);
                for (int j = 0; j < l - i - 1; j++)
                    temp.Remove();
                q.Insert(temp.Remove());
            }
        }

        public static void RemoveDuplicates(Queue<int> q)
        {
            Queue<int> temp = new Queue<int>();
            SpilledOn(temp, q);

            while (!temp.IsEmpty())
            {
                int val = temp.Remove();
                if (!IsExist(q, val))
                    q.Insert(val);
            }
        }
```

```
295
296         public static int Count(Queue<int> q, int n)
297         {
298             Queue<int> temp = Clone(q);
299
300             int cnt = 0;
301
302             while (!temp.IsEmpty())
303             {
304                 if (temp.Remove() == n)
305                     cnt++;
306             }
307
308             return cnt;
309         }
310
311         public static void RemoveSpec(Queue<int> q, int val)
312         {
313             Queue<int> temp = new Queue<int>();
314             SpilledOn(temp, q);
315
316             while (!temp.IsEmpty())
317             {
318                 int curr = temp.Remove();
319                 if (curr != val)
320                     q.Insert(curr);
321             }
322         }
323
324         public static void InsertAtPos<T>(Queue<T> q, T e, int n)
325         {
326         }
327
328         // --- Bagrut Exercices ---
329         public static int ToNumber(Queue<int> q)
330         {
331             int num = 0;
332
333             while (!q.IsEmpty())
334             {
335                 int val = q.Remove();
336
337                 num *= 10;
338                 num += val;
339             }
340
341             return num;
342         }
343
```

```
        public static int BigNumber(Queue<Queue<int>> q)
        {
            int max = int.MinValue;

            Queue<Queue<int>> clone = Clone(q);

            while (!clone.IsEmpty())
            {
                int val = ToNumber(clone.Remove());
                if (val > max)
                    max = val;
            }

            return max;
        }

        // ---
        public static void ConnectQueues<T>(Queue<T> q1, Queue<T> q2)
        {
            Queue<T> temp = Clone(q2);

            while (!temp.IsEmpty())
                q1.Insert(temp.Remove());
        }
        public static Queue<int> DoublesToPali(Queue<int> qd)
        {
            if (GetSize(qd) == 2)
                return qd;

            Queue<int> res = new Queue<int>();
            res.Insert(qd.Remove());
            int val = qd.Remove();
            ConnectQueues(res, DoublesToPali(qd));
            res.Insert(val);

            return res;
        }

        // ---
        public static bool IsIdentical(Queue<int> q1, Queue<int> q2)
        {
            Queue<int> copy1 = Clone(q1);
            Queue<int> copy2 = Clone(q2);

            bool identical = true;

            while(identical && !copy1.IsEmpty() && !copy2.IsEmpty())
            {

```

```
                identical = copy1.Remove() == copy2.Remove();

                if ((!copy1.IsEmpty() && copy2.IsEmpty()) ||
                  (copy1.IsEmpty() && !copy2.IsEmpty()))
                    identical = false;
            }

            return identical;
        }

        public static bool IsSimilar(Queue<int> q1, Queue<int> q2)
        {
            int size = GetSize(q1);

            bool similar = false;

            for (int i = 0; i < size && !similar; i++)
            {
                similar = IsIdentical(q1, q2);
                LastToFirst(q1);
            }

            return similar;

        }

        // -- Bagrut 2023
        public static bool TwoSum(Queue<int> q, int x)
        {
            bool found = false;
            Queue<int> copy = Clone(q);

            while (!found && !copy.IsEmpty())
            {
                Queue<int> temp = Clone(copy);
                int head = temp.Remove();

                while(!temp.IsEmpty() && !found)
                {
                    found = head + temp.Remove() == x;
                }

                copy.Remove();
            }

            return found;


        }
```

```
441
442     }
443 }
444
445
446
447
448
449
450
451
452
453
454
455
```