

```

1 int FactorTree (int n){
2   int height = 0;
3   while (n > 1){
4     bool flag = false;
5     for (int i = 2; i <= sqrt(n); i++){
6       if (n % i == 0){
7         n = n / i;
8         flag = true;
9         break;
10      }
11      height++;
12      if (!flag){ break; }
13    }
14    return height;
15  }

```

(a) Type 1: C++

```

1 vector<vector<int>>> ConstructTree (
2   int n, vector<vector<int>>> edges){
3   vector<vector<int>>> adjl;
4   for (int i = 0; i < n; i++){
5     adjl.push_back (vector<int> ());
6   }
7   for (auto e : edges){
8     int u = e[0];
9     int v = e[1];
10    adjl[u].push_back (v);
11    adjl[v].push_back (u);
12  }
13  return adjl;

```

(b) Type 2: C++

```

1 int MostFrequent (vector<int> arr, int n){
2   unordered_map<int, int> hp;
3   for (int i = 0; i < n; i++){
4     if (hp.find (arr[i]) != hp.end()){
5       hp[arr[i]]++;
6     } else {
7       hp[arr[i]] = 1;
8     }
9   }
10  int max_count = 0, res = -1;
11  for (auto &entry : hp){
12    if (max_count < entry.second){
13      res = entry.first;
14      max_count = entry.second;
15    }
16  }
17  return res;

```

(c) Type 3: C++

```

1 int WithdrawBalance (
2   int start, vector<int> withdrawals){
3   int end = accumulate (withdrawals.begin(),
4     withdrawals.end(),
5     start,
6     [](int balance, int next_withdrawal){
7       return next_withdrawal <= balance ?
8         balance - next_withdrawal :
9         balance;
10    });
11  return end;

```

(d) Type 4: C++

```

1 int factorTree (int n){
2   int height = 0;
3   while (n > 1){
4     bool flag = false;
5     for (int i = 2; i <= Math.sqrt(n); i++){
6       if (n % i == 0){
7         n = n / i;
8         flag = true;
9         break;
10      }
11      height++;
12      if (!flag){ break; }
13    }
14    return height;
15  }

```

(e) Type 1: Java

```

1 List<List<Integer>>> constructTree (
2   int n, List<List<Integer>>> edges){
3   List<List<Integer>>> adjl = new ArrayList<> ();
4   for (int i = 0; i < n; i++){
5     adjl.add (new ArrayList<> ());
6   }
7   for (var e : edges){
8     int u = e.get (0);
9     int v = e.get (1);
10    adjl.get (u).add (v);
11    adjl.get (v).add (u);
12  }
13  return adjl;

```

(f) Type 2: Java

```

1 int mostFrequent (List<Integer> arr, int n){
2   Map<Integer, Integer> hp = new HashMap<> ();
3   for (int i = 0; i < n; i++){
4     if (hp.containsKey (arr.get(i))){
5       hp.put (arr.get(i), hp.get (arr.get(i)) + 1);
6     } else {
7       hp.put (arr.get(i), 1);
8     }
9   }
10  int maxCount = 0, res = -1;
11  for (var entry : hp.entrySet()){
12    if (maxCount < entry.getValue()){
13      res = entry.getKey();
14      maxCount = entry.getValue();
15    }
16  }
17  return res;

```

(g) Type 3: Java

```

1 int withdrawBalance (
2   int start, List<Integer> withdrawals){
3   int end = withdrawals.stream().
4     .reduce (start,
5       (balance, nextWithdrawal) ->
6         nextWithdrawal <= balance ?
7           balance - nextWithdrawal :
8           balance
9     );
10  return end;

```

(h) Type 4: Java

```

1 int FactorTree (int n){
2   int height = 0;
3   while (n > 1){
4     bool flag = false;
5     for (int i = 2; i <= Math.Sqrt(n); i++){
6       if (n % i == 0){
7         n = n / i;
8         flag = true;
9         break;
10      }
11      height++;
12      if (!flag){ break; }
13    }
14    return height;
15  }

```

(i) Type 1: C#

```

1 vector<vector<int>>> ConstructTree (
2   int n, vector<vector<int>>> edges){
3   vector<vector<int>>> adjl;
4   for (int i = 0; i < n; i++){
5     adjl.push_back (vector<int> ());
6   }
7   for (auto e : edges){
8     int u = e[0];
9     int v = e[1];
10    adjl[u].push_back (v);
11    adjl[v].push_back (u);
12  }
13  return adjl;

```

(j) Type 2: C#

```

1 int MostFrequent (List<int> arr, int n){
2   Dictionary<int, int> hp = new Dictionary<int, int> ();
3   for (int i = 0; i < n; i++){
4     if (hp.ContainsKey (arr[i])){
5       hp[arr[i]]++;
6     } else {
7       hp[arr[i]] = 1;
8     }
9   }
10  int maxCount = 0, res = -1;
11  foreach (var entry in hp){
12    if (maxCount < entry.Value){
13      res = entry.Key;
14      maxCount = entry.Value;
15    }
16  }
17  return res;

```

(k) Type 3: C#

```

1 int WithdrawBalance (
2   int start, List<int> withdrawals){
3   int end = withdrawals.Aggregate (start,
4     (balance, nextWithdrawal) =>
5       nextWithdrawal <= balance ?
6         balance - nextWithdrawal :
7         balance
8   );
9   return end;

```

(l) Type 4: C#

```

1 def factor_tree (n):
2   height = 0
3   while n > 1:
4     flag = False
5     for i in range (2, int (math.sqrt (n)) + 1):
6       if n % i == 0:
7         n = n // i
8         flag = True
9         break
10    height += 1
11    if not flag:
12      break
13  return height

```

(m) Type 1: Python

```

1 def construct_tree (n, edges):
2   adjl = []
3   for i in range (n):
4     adjl.append ([])
5   for e in edges:
6     u = e[0]
7     v = e[1]
8     adjl[u].append (v)
9     adjl[v].append (u)
10  return adjl

```

(n) Type 2: Python

```

1 def most_frequent (arr, n):
2   hp = {}
3   for i in range (n):
4     if arr[i] in hp:
5       hp[arr[i]] += 1
6     else:
7       hp[arr[i]] = 1
8   max_count, res = 0, -1
9   for key, value in hp.items():
10    if max_count < value:
11      res = key
12      max_count = value
13  return res

```

(o) Type 3: Python

```

1 def withdraw_balance (
2   start, withdrawals):
3   end = functools.reduce (
4     (balance, next_withdrawal) =>
5       balance - next_withdrawal
6     if next_withdrawal <= balance
7     else balance
8     , withdrawals
9     , start)
10  return end

```

(p) Type 4: Python

```

1 function factorTree (n){
2   let height = 0;
3   while (n > 1){
4     let flag = false;
5     for (let i = 2; i <= Math.sqrt(n); i++){
6       if (n % i === 0){
7         n = Math.floor (n / i);
8         flag = true;
9         break;
10      }
11      height++;
12      if (!flag){ break; }
13    }
14    return height;
15  }

```

(q) Type 1: JavaScript

```

1 function constructTree (n, edges){
2   let adjl = [];
3   for (let i = 0; i < n; i++){
4     adjl.push ([]);
5   }
6   for (let e of edges){
7     let u = e[0];
8     let v = e[1];
9     adjl[u].push (v);
10    adjl[v].push (u);
11  }
12  return adjl;

```

(r) Type 2: JavaScript

```

1 function mostFrequent (arr, n){
2   let hp = new Map();
3   for (let i = 0; i < n; i++){
4     if (hp.has (arr[i])){
5       hp.set (arr[i], hp.get (arr[i]) + 1);
6     } else {
7       hp.set (arr[i], 1);
8     }
9   }
10  let maxCount = 0, res = -1;
11  for (let entry of hp){
12    if (maxCount < entry[1]){
13      res = entry[0];
14      maxCount = entry[1];
15    }
16  }
17  return res;

```

(s) Type 3: JavaScript

```

1 function withdrawBalance (
2   start, withdrawals){
3   let end = withdrawals.reduce (
4     (balance, nextWithdrawal) =>
5       balance - nextWithdrawal <= balance ?
6         balance - nextWithdrawal :
7         balance;
8     , start);
9   return end;

```

(t) Type 4: JavaScript

Fig. 1: Examples for the four translation types in our constructed benchmark. The translation for each type is highlighted with yellow blocks.