A - Filling Shapes

If you want to have no empty spaces on $3 \times n$ tiles, you should fill leftmost bottom tile. Then you have only 2 choices;

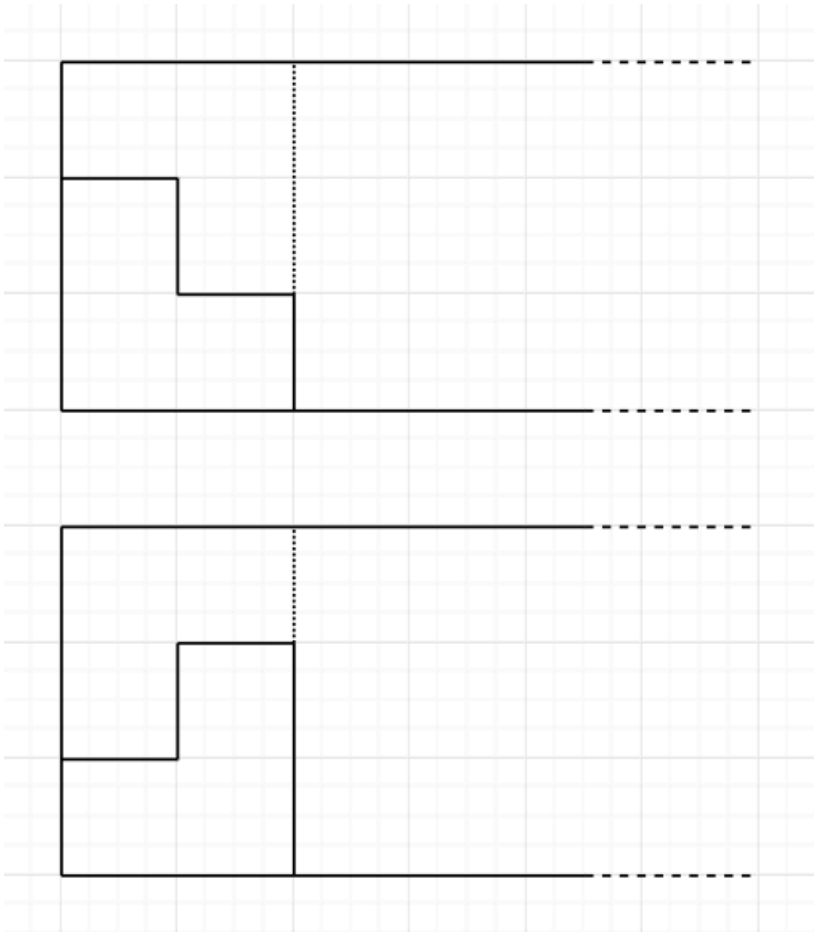

Figure 1: Two ways to fill.

Both cases force you to group leftmost 3×2 tiles and fill. By this fact, we should group each 3×2 tiles and fill independently. So the answer is — if n is odd, then the answer is 0 (impossible), otherwise, the answer is 2^n.

Time complexity is $O(1)$ with bit operation or $O(n)$ with iteration.

> **Tips**
>
> Calculating 2^n using `pow(2, n)` is awkward, and runs slow.
>
> We suggest using `1 << n` instead.

Solution Code

```c
#include <stdio.h>
int main(void){
  int n; scanf("%d", &n);
  if(n%2==0) printf("%d", 1<<(n/2));
  else printf("0");
  return 0;
}
```

B - Finding the Car

For each query, we binary search to find the last sign we passed (since the array a is sorted). Say it is a_r. Then, b_r minutes have passed.

To find out how much time has passed since we left that sign, we know that the speed between sign r and $r+1$ is $\frac{a_{r+1}-a_r}{b_{r+1}-b_r}\frac{\text{distance}}{\text{minute}}$ (by the usual formula for speed).

We have passed a distance $c - a_r$ since the last sign, so the total number of minutes since the last sign is

$$\frac{b_{r+1}-b_r}{a_{r+1}-a_r}\frac{\text{minute}}{\text{distance}} \times c - a_r \text{ distance} = (c-a_r) \times \frac{b_{r+1}-b_r}{a_{r+1}-a_r} \text{ minutes.}$$

Adding this to our b_r from before, we get the answer.

Be careful about using floating-point numbers, as they can behave strangely. Our solution below doesn't use them at all.

> **Tips** b C++ STL `lower_bound` and `upper_bound` saves you from writing binary search. You can learn about it here for `lower_bound` and here for `upper_bound`.

Solution Code

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

void solve() {
    int n, k, q;
    cin >> n >> k >> q;
    vector<long long> a(k+1), b(k+1);
    a[0] = 0;
    b[0] = 0;
    for(int i = 1; i <= k; i++)
        cin >> a[i];

    for(int i = 1; i <= k; i++)
        cin >> b[i];

    for(int i = 0; i < q; i++) {
        long long c;
        cin >> c;
        int l = 0, r = k;

        r = lower_bound(a.begin(), a.end(), c) - a.begin();

        if(a[r] == c) {
            cout << b[r] << " ";
            continue;
        }

        long long ans = b[r - 1] + (c - a[r - 1]) * (b[r] - b[r - 1]) / (a[r] - a[r -
1]);
```

```
        cout << ans << " ";
    }
    cout << endl;
}

int main() {
    int t = 1;
    cin >> t;
    while (t--) {
        solve();
    }
}
```

C - Download Speed Monitor

Using two pointers, one pointing to the first element, one pointing to the k-th, and maintain the sum between the two pointers. When moving rightward, update the sum.

Solution Code

```
#include <iostream>
#include <numeric>
#include <vector>

using namespace::std;

int main() {
    int n, k;
    cin >> n >> k;

    vector<long long> a(n + 1);

    for(int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    long long ans = accumulate(a.begin(), a.begin() + k, 0ll);

    for(int i = 0, j = k; j <= n; ++i, ++j) {
        if(ans < 1024 * k)
            cout << double(ans) / k << " KiBps" << '\n';
        else
            cout << double(ans) / k / 1024 << " MiBps" << '\n';
        ans += a[j];
        ans -= a[i];
    }

    return 0;
}
```

D - XOUR

Note that if ai **XOR** $a_j < 4$, then a_i and a_j must share all bits in their binary representation, except for the last 2 bits. This is because if they have a mismatch in any greater bit, their **XOR** will include this bit, making its value $\geq 2^2 = 4$.

This means that we can group the numbers by removing the last two bits and putting equal numbers into the same group. In each group, we can order the numbers freely (since we can swap any two of them), so it's optimal to sort the numbers in each group.

Thus we can just divide the numbers into groups and sort each, solving the problem in O(nlogn) . There are several ways to implement this: for instead, you can use a map storing all the groups, and then sort the values in each group. The implementation we used maps each integer to a priority queue, which automatically will sort the numbers in each group.

Solution Code

```
#include <map>
#include <vector>
#include <iostream>
#include <algorithm>
#include <deque>

using namespace std;

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        map<int, deque<int>> m;
        vector<int> a(n);
        for (int i = 0; i < n; i++) {
            cin >> a[i];
            m[a[i] >> 2].push_back(a[i]);
        }

        for (auto &a : m) {
            sort(a.second.begin(), a.second.end());
        }

        for (auto &a : a) {
            cout << m[a >> 2].front() << " ";
            m[a >> 2].pop_front();
        }
        cout << endl;
    }
    return 0;
}
```

E - Number Deletion Game

We note that when the number largest numbers is odd, Alice wins. Otherwise, Bob wins.

Proof:

We use mathematical induction to prove that. The statement when the max number is k is denoted by $P(k)$.

Base case. When $k = 1$, it is clear that $P(1)$ holds, since each player can take only one number and add nothing.

Inductive Steps. We suppose that $P(k-1)$ holds for $k > 1$, we will show that $P(k)$ holds.

When there are an even number of ks in the sequence, Alice will take the last k, since they will take k's in turns and any number greater than or equal to k will never be added. Alice can decide whether the number of $k-1$s will be even or odd. If there are an even number of $k-1$s, Alice will do nothing. If tere are an odd number of $k-1$s, Alice will choose $k-1$ as y. Thus the number of $k-1$s will be even. Since $P(k-1)$ holds, we have that Alice will win.

When there are an odd number of ks, it can be proved similarly.

Solution Code

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace::std;

const char * ans[] = {"Bob", "Alice"};

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for(int i = 0; i < n; ++i)
        cin >> a[i];
    int k = *max_element(a.begin() ,a.end());
    cout << ans[count_if(a.begin(), a.end(), [k](int x){return x == k;}) & 1]
         << '\n';
    return 0;
}
```

F - DIDA's Secret Room

We use an integer x to represent keys, where the i-th digit in x represents holding key i.

Then we can use breadth-first search to find the shortest path.

Solution Code

```
#include <iostream>
#include <vector>
#include <queue>

using namespace::std;

struct Edge {
    int u, v;
    int key;

    Edge(int u, int v, int key) : u(u), v(v), key(key){}
};

struct State {
    int pos;
    int holding;
    int ans;

    State(int pos, int holding, int ans) : pos(pos), holding(holding), ans(ans){}
};
```

```cpp
int main() {
    int n, m, k;
    cin >> n >> m >> k;

    vector<int> keys(n + 1, 0);
    vector<vector<Edge>> graph(n + 1, vector<Edge>());

    for(int i = 1; i <= n; ++i) {
        for(int j = 0; j < k; ++j) {
            int t;
            cin >> t;
            keys[i] |= t << j;
        }
    }

    for(int i = 0; i < m; ++i) {
        int u, v, key = 0;
        cin >> u >> v;
        for(int j = 0; j < k; ++j) {
            int t;
            cin >> t;
            key |= t << j;
        }
        graph[u].emplace_back(u, v, key);
    }

    queue<State> q;
    vector<bool> visited((n + 2) << k, false);

    q.emplace(1, keys[1], 0);
    visited[(1 << k) | keys[1]] = true;

    while(!q.empty()) {
        auto state = q.front();
        q.pop();

        for(auto edge : graph[state.pos]) {
            int next_keys = state.holding | keys[edge.v];
            if(!(edge.key & ~state.holding) && !visited[edge.v << k | next_keys]) {
                q.emplace(edge.v, next_keys, state.ans + 1);
                visited[(edge.v << k) | next_keys] = 1;
                if (edge.v == n) {
                    cout << state.ans + 1 << '\n';
                    return 0;
                }
            }
        }
    }

    cout << "No Solution\n";

    return 0;
}
```