First Round SOI 2017

Solution Booklet

Swiss Olympiad in Informatics

January 7, 2017

1 Number Riddle

The Number Riddle is a riddle where you get N numbers and you have to add or subtract them to get a given result R. You have to use all numbers and you are not allowed to subtract the first one. You simply have to output whether it is possible to reach R or not. The subtasks differ from each other solely by the amount of numbers N and the amount of subtractions allowed in the solution.

Subtask 1: Warm up (25 points)

The simplest possible riddle consists only of two numbers, a and b. All we have to do is comparing $a + b$ and $a - b$ to the result R.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int solve(int N, int R, int *n) {
    return (R == n[0] + n[1]) || (R == n[0] - n[1]);
}

int main() {
    int T;
    scanf("%d", &T);

    for (int t = 0; t < T; t++) {
        int N, R;
        scanf("%d %d", &N, &R);
        int n[N];
        for (int i = 0; i < N; i++) {
            scanf("%d", &n[i]);
        }
        printf("Case #%d: %s\n", t + 1, solve(N, R, n) ? "YES" : "NO");
    }
    return 0;
}
```

Subtask 2: One subtraction (25 points)

Things get more interesting when more numbers are given. To reduce the complexity, at most one subtraction is allowed.

You get a linear time solution by calculating the sum and then subtract each number individually. Of course you have to subtract twice its value $a + b + c + ... - 2b = a - b + c + ...$.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

bool isodd(int x) {
  return x%2 != 0;
}
```

```
8
9  int solve(int N, int R, int *n) {
10     int sum = 0;
11     for (int i = 0; i < N; i++)
12         sum += n[i];
13
14     if (sum < R || isodd(sum) != isodd(R))
15         return 0;
16
17     if (sum == R)
18         return 1;
19
20     for (int i = 1; i < N; i++) {
21         if ((sum - 2 * n[i]) == R)
22             return 1;
23     }
24
25     return 0;
26 }
```

Subtask 3: Two subtractions (25 points)

As in the previous task, we now have multiple numbers and a limited set of operations. This time we are allowed to subtract twice. We can solve this by using an additional loop. It doesn't matter in which order we subtract the numbers $a+b+c+\ldots-2b-2c = a+b+c+\ldots-2c-2b = a-b-c+\ldots$. The inner loop could, therefore, start with the proceeding element of the outer loop. The program is not only marginally faster, it also ensures that we do not subtract the same element twice.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  bool isodd(int x) {
6    return x%2 != 0;
7  }
8
9  int solve(int N, int R, int *n) {
10     int sum = 0;
11     for (int i = 0; i < N; i++)
12         sum += n[i];
13
14     if (sum < R || isodd(sum) != isodd(R))
15         return 0;
16
17     if (sum == R)
18         return 1;
19
20     for (int i = 1; i < N; i++) {
21         int p = sum - 2 * n[i];
22         if (p == R)
23             return 1;
24         for (int j = i + 1; j < N; j++)
25             if ((p - 2 * n[j]) == R)
26                 return 1;
```

```
    }
    return 0;
}
```

Subtask 4: Countless possibilities (25 points)

As the title suggests, it is not possible to solve this task using brute-force. You have up to $N = 100$ numbers you can combine with either + or −, thus resulting in 2^{99} possible equations.

Instead of calculating this individualy, we store all the provisional results and apply the next number to them. To store them we could either use some kind of container which removes duplicates or a bool array with the size of the sum wherein we mark the numbers we were able to reach.

```cpp
#include <bits/stdc++.h>
#define FOR(i, n) for (int i = 0; i < n; i++)
#define FORS(i, n, s) for (int i = s; i < n; i++)

using namespace std;

int main() {
    int tcn;
    cin >> tcn;
    FOR(t, tcn) {
        int n, r;
        cin >> n >> r;
        vector<int> num(n);
        FOR(i, n) { cin >> num[i]; }
        set<int> pos;
        pos.insert(num[0]);
        FORS(i, n, 1) {
            set<int> np;
            for (auto p : pos) {
                np.insert(p + num[i]);
                np.insert(p - num[i]);
            }
            pos = np;
        }
        cout << "Case #" << t + 1 << ": "
             << ((string[]){"NO", "YES"})[pos.find(r) != pos.end()] << "\n";
    }

    return 0;
}
```

2 Super Powers

Subtask 1: Exactly C Superheroes (20 points)

In this subtask you have to choose C superheroes and you can choose the same one several times.

As long as there are still heroes with negative values we can just take the smallest one and invert it (this will give us the most possible addition to the sum). To do this we just sort the list and process all values until they get positive. Afterwards we swap the smallest value until we used up all the swaps.

For all subtasks it is important to use `long long int` instead of just `int` because the sum can get quite large.

```cpp
#include <bits/stdc++.h>
using namespace std;
using lli = long long int;

int main() {
    int tcn;
    cin >> tcn;
    for (int t = 1; t <= tcn; t++) {
        int c, n;
        cin >> c >> n;
        vector<int> k(n);
        for (int i = 0; i < n; i++) {
            cin >> k[i];
        }

        // flip negative values as long as possible
        sort(k.begin(), k.end());
        int lim = min(c, n);
        for (int i = 0; i < lim; i++) {
            if (k[i] < 0) {
                k[i] *= -1;
                c--;
            } else {
                break;
            }
        }

        // if we have some flips left we flip the smallest value
        if (c > 0 && c % 2 == 1) {
            int mi = 0;
            for (int i = 0; i < n; i++) {
                if (k[i] < k[mi]) {
                    mi = i;
                }
            }
            k[mi] *= -1;
        }

        lli sum = 0;
        for (int i : k) {
```

```
41              sum += i;
42          }
43          cout << "Case #" << t << ": " << sum << "\n";
44      }
45  }
```

Subtask 2: C Different Superheroes (20 points)

In this subtask you have to choose C different superheroes.

We do the same thing as in the previous subtask but instead of flipping the same value until we used up our flips we flip the smallest possible value that wasn't flipped already. For this we sort the list and flip the first C values.

```
#include <bits/stdc++.h>
using namespace std;
using lli = long long int;

int main() {
    int tcn;
    cin >> tcn;
    for (int t = 1; t <= tcn; t++) {
        int c, n;
        cin >> c >> n;
        vector<int> k(n);
        for (int i = 0; i < n; i++) {
            cin >> k[i];
        }

        // we always swap the smallest value that wasn't already swapped
        sort(k.begin(), k.end());
        int lim = min(c, n);
        for (int i = 0; i < lim; i++) {
            k[i] *= -1;
        }

        lli sum = 0;
        for (int i : k) {
            sum += i;
        }
        cout << "Case #" << t << ": " << sum << "\n";
    }
}
```

Subtask 3: C Contiguous Superheroes (20 points)

In this subtask the C superheroes have to be contiguous.

To solve this we can use a rolling sum to find the segment with the most negative sum. If we flip this segment we will have an optimal solution. The rolling sum works as follows: We sum up the first C values. This is obviously the sum from 0 to $C-1$. Wo now go through the whole list for i in $[C, N[$ and in each step update our sum by adding the next value on the right and subtract the most left element that was still part of the sum. As we added one element on the right and removed one

on the left the sum still consists of C numbers that are continuous. In each step we compare it to our previous most negative sum and update it if necessary.

```cpp
#include <bits/stdc++.h>
using namespace std;
using lli = long long int;

int main() {
    int tcn;
    cin >> tcn;
    for (int t = 1; t <= tcn; t++) {
        int c, n;
        cin >> c >> n;
        vector<int> k(n);
        for (int i = 0; i < n; i++) {
            cin >> k[i];
        }

        lli sum = 0;
        for (int i : k) {
            sum += i;
        }

        // init rolling sum
        lli rolling = 0;
        for (int i = 0; i < c; i++) {
            rolling += k[i];
        }

        // find maximum
        lli ma = rolling;
        for (int i = c; i < n; i++) {
            rolling += k[i] - k[i - c];
            ma = min(ma, rolling);
        }

        cout << "Case #" << t << ": " << (sum - 2 * ma) << "\n";
    }
}
```

Subtask 4: Contiguous Superheroes Small (20 points)

In this two subtask you can choose any number of continuous superheroes. For subtask 4 you were able to use a slower solution (e.g. use the code from the previous subtask with all possible values for C). We will explain the efficient solution in the next section.

Subtask 5: Contiguous Superheroes Large (20 points)

In this two subtask you can choose any number of continuous superheroes. For subtask 4 you were able to use a slower solution (e.g. use the code from the previous subtask with all possible values for C). We will only explain the efficient solution.

The problem one has to solve in this task is known as the maximum subarray problem. Given an array of numbers, find any number of continuous values such that their sum is maximal. In our case we want to find the subarray with the smallest (most negative) sum instead of the largest sum.

To solve this we will go through the list from left to right. We keep track of the smallest segment we have seen so far and the minimal value of all segments that end at the current location. One thing we notice is that should the segment that ends at the current location get positive with the current value added it's better to take no element at all. In this case we reset this value to zero. Otherwise we add the current value.

In the implementation below `ms` stores the minimal segment seen so far and `asu` is the active sum of the segment ending at the current position.

Note that to get the solution we have to subtract the found segment twice from the original sum because we flip the sign and don't just take it away. For example if you had $a + b + c$ and flip b you have now $a - b + c = a + b - b - b + c = a + b - 2 * b + c$

```cpp
#include <bits/stdc++.h>
using namespace std;
using lli = long long int;

int main() {
    int tcn;
    cin >> tcn;
    for (int t = 1; t <= tcn; t++) {
        int n;
        cin >> n;
        vector<int> k(n);
        for (int i = 0; i < n; i++) {
            cin >> k[i];
        }

        lli sum = 0;
        for (int i : k) {
            sum += i;
        }

        // maximum sub array
        lli ms = 0;
        lli asu = 0;
        for (int i = 0; i < n; i++) {
            asu += k[i];
            if (asu > 0) {
                asu = 0;
            }
            ms = min(ms, asu);
        }

        lli sol = sum - 2 * ms;
        cout << "Case #" << t << ": " << sol << "\n";
    }
}
```

3 Cup Sort

This task is about ordering cups of different colors by swapping adjacent cups, or more precisely about calculating the number of such swaps needed to sort the cups according to some criterium. The cups' colors are represented by a list of integers. Mathematically speaking, you were asked to count the number of inversions in such a list.

Subtask 1: At most one move (10 points)

Here, you are given a list of N integers which are all either 0 or 1, and your task is to find out if the list can be sorted ascendingly with only one swap of adjacent numbers.

One possible solution was to look for the first 1. If there is no 1, the cups are already sorted. Otherwise, look for the first 0 after it: If there is none, the cups are sorted. If there is one at the position directly behind the first 1, we need one swap to order them; look for another 0 after the 1, if there is another 0 we need at least two swaps, otherwise we can sort the cups with one move. If there is a 0 more than one position after the first 1, we would have to swap the 0 with all the 1's in between, which would be more than one swap.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int t;
    cin >> t;
    for (int tc = 1; tc <= t; ++tc) {
        int n;
        cin >> n;
        vector<int> colors(n);
        for (int &c : colors)
            cin >> c;
        int oi = -1; // index of first 1
        int zi = -1; // index of first 0 after the first 1
        bool possible = true;
        for (int i = 0; i < n; ++i) {
            if (colors[i] == 1 && oi == -1)
                oi = i;
            else if (colors[i] == 0 && oi >= 0) {
                if (zi >= 0 || i - oi > 1) { // need more than one swap if more
                                             // than one 0 after the first 1 or
                                             // if multiple 1's before the 0
                    possible = false;
                    break;
                }
                zi = i;
            }
        }
        if (possible)
            cout << "Case #" << tc << ": YES\n";
        else
            cout << "Case #" << tc << ": NO\n";
    }
```

```
}
```

Subtask 2: Two colors (20 points)

Now we want to know how many swaps exactly we need for a list as in the first subtask.

There is a simple solution in $O(N)$ which we get by slightly generalizing the idea from subtask one. We iterate over the list keeping track of the number of 1's encountered so far, adding that number at every 0 we encounter. This is correct because clearly, we have to swap every 0 with every 1 before it at some point while sorting. But we don't have to do any other swaps than these because swapping two 0's is pointless and swapping a 0 with a 1 after it is also not useful.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int t;
    cin >> t;
    for (int tc = 1; tc <= t; ++tc) {
        int n;
        cin >> n;
        int ones = 0;
        int swaps = 0;
        for (int i = 0; i < n; ++i) {
            int ci;
            cin >> ci;
            if (ci == 1)
                ++ones;
            else
                swaps += ones;
        }
        cout << "Case #" << tc << ": " << swaps << "\n";
    }
}
```

Subtask 3: Different colors (20 points)

Here, all numbers are pairwise distinct, but they can be different from 0 and 1.

Now we can't use an ad-hoc approach as before anymore. One way to solve it is by modifying a classical sorting algorithm.

The conceptually easiest one is Bubble Sort, where we iterate over the list again and again until the list is sorted, each time comparing each number with the one after it, swapping them if the first one is bigger, and counting the number of swaps. As above we see that we never do an unnecessary swap and that the algorithm will have to terminate after looping over the list at most N times, leaving us with a sorted list, therefore the number we get is correct. But this takes $O(N^2)$ time in the worst case. We can do better.

Thus we use Merge Sort. It only takes $O(N \cdot \log(N))$ time, which will even be fast enough for the last subtask. Merge Sort is a divide and conquer algorithm, splitting the list into two halves, sorting each half by recursively applying itself, and merging the two halves again. The modification we have to make in order to count the inversions is counting them in each merge step and summing all counts up. Merging two sorted arrays works as follows: We iterate over both arrays at the same

time, only comparing our current element in the left array with our current element in the right array, putting the smaller one into the resulting list. We can do this because the two arrays are already sorted, therefore we know that all elements before the current element are smaller and all elements after the current element are larger. The number of inversions we undo at each step is:

- 0, if we take an element from the left half (we already counted the swaps with earlier numbers from the right half, do not count them twice),
- the number of elements in the left half which are larger than the element we take, if we take an element from the right half.

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> colors;
vector<int> tmp;

int merge(int l, int m, int r) {
    int inversions = 0;
    int lefti = l;
    int righti = m;
    int outi = l;
    while (lefti < m && righti < r) {
        if (colors[lefti] <= colors[righti]) {
            tmp[outi] = colors[lefti];
            ++outi;
            ++lefti;
        } else {
            tmp[outi] = colors[righti];
            inversions += m - lefti; // all colors with indices in [lefti,m)
                                     // need to be swapped with colors[righti]
            ++outi;
            ++righti;
        }
    }
    while (lefti < m) {
        tmp[outi] = colors[lefti];
        ++outi;
        ++lefti;
    }
    while (righti < r) {
        tmp[outi] = colors[righti];
        ++outi;
        ++righti;
    }
    copy(tmp.begin() + l, tmp.begin() + r,
         colors.begin() + l); // copy the sorted range back
    return inversions;
}

int mergesort(int l, int r) {
    if (r - l <= 1) // there are no inversions in an empty list or a list with
                    // one element
        return 0;
    int m = l + (r - l) / 2;
```

```
45      int inversions = mergesort(l, m) + mergesort(m, r);
46      inversions += merge(l, m, r);
47      return inversions;
48  }
49
50  int main() {
51      int t;
52      cin >> t;
53      for (int tc = 1; tc <= t; ++tc) {
54          int n;
55          cin >> n;
56          colors.resize(n);
57          tmp.resize(n);
58          for (int &c : colors)
59              cin >> c;
60          cout << "Case #" << tc << ": " << mergesort(0, n) << "\n";
61      }
62  }
```

Subtask 4: General case (20 points)

This subtask is almost the same as the one before, but not all numbers have to be distinct. We can use the same algorithms, but need to take care to only swap numbers if the one which comes first is strictly lager than the one later on, not swapping if they are equal. For the Merge Sort variant this means preferring the first list when merging if both current elements are equal. This is already true in the implementation above.

Subtask 5: Many cups (30 points)

This subtask adds two complications: First, $O(N^2)$ algorithms are too slow. And second, each integer is replaced by a group of individually specified size of equal integers, i.e. the numbers get assigned multiplicities. Note that there can still be different groups of the same integer.

When swapping two groups, the amount of swaps is actually the product of the sizes of the groups, because each element of the first group has to be swapped with each element of the second group. In the merge step, instead of keeping track of the number of elements left in the first list, we need to keep track of the sum of their multiplicities.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  vector<pair<long long, long long>> colors;
5  vector<pair<long long, long long>> tmp;
6  vector<long long> prefsum;
7
8  long long merge(int l, int m, int r) {
9      long long inversions = 0;
10     int lefti = l;
11     int righti = m;
12     int outi = l;
13     while (lefti < m && righti < r) {
14         if (colors[lefti].second <= colors[righti].second) {
```

```
            tmp[outi] = colors[lefti];
            ++outi;
            ++lefti;
        } else {
            tmp[outi] = colors[righti];
            inversions += colors[righti].first *
                          (prefsum[m] - prefsum[lefti]); // all colors with
                                                         // indices in [lefti,m)
                                                         // need to be swapped
                                                         // with colors[righti]
            ++outi;
            ++righti;
        }
    }
    while (lefti < m) {
        tmp[outi] = colors[lefti];
        ++outi;
        ++lefti;
    }
    while (righti < r) {
        tmp[outi] = colors[righti];
        ++outi;
        ++righti;
    }
    copy(tmp.begin() + l, tmp.begin() + r,
         colors.begin() + l);     // copy the sorted range back
    for (int i = l; i < r; ++i) { // update prefsum
        prefsum[i + 1] = prefsum[i] + colors[i].first;
    }
    return inversions;
}

long long mergesort(int l, int r) {
    if (r - l <= 1) // there are no inversions in an empty list or a list with
                    // one element
        return 0;
    int m = l + (r - l) / 2;
    long long inversions = mergesort(l, m) + mergesort(m, r);
    inversions += merge(l, m, r);
    return inversions;
}

int main() {
    int t;
    cin >> t;
    for (int tc = 1; tc <= t; ++tc) {
        int k;
        cin >> k;
        colors.resize(k);
        tmp.resize(k);
        prefsum.resize(k + 1);
        for (auto &p : colors)
            cin >> p.first >> p.second;
        prefsum[0] = 0;
        for (int i = 0; i < k; ++i)
            prefsum[i + 1] = prefsum[i] + colors[i].first;
```

```
71          cout << "Case #" << tc << ": " << mergesort(0, k) << "\n";
72      }
73  }
```

Subtask 6: Alternative Solution

There is also another way than Merge Sort to solve all subtasks in $O(N \cdot \log(N))$, namely using the sort function provided by C++ and a data structure to calculate prefix sums quickly. Its advantage is that it is shorter to code (and it is more frequently useful for the SOI than Merge Sort). The probably best way to calculate prefix sums here is using a Fenwick Tree (sometimes also called Binary Indexed Tree), as it is short to code. This is because the tree structure is implicitly given by the binary representation of the indices, and the tree is stored as a simple vector/array. It represents an array of numbers and allows us to query the prefix sum of any index in $O(\log n)$ and change an element of the array in $O(\log n)$. The idea is then to store for each group of cups a tuple of three numbers, namely the index in the input, the color and the multiplicity. Then we sort the list of these tuples by their color. Note that if two cup groups have the same color we need to order them by input index, so we do not count swaps of cups of the same color. Now the list is in the order we want to sort it to and we can calculate the number of inversions from the indices in the input that we stored:

Initialize the Fenwick Tree. Iterate over the list, adding the multiplicity of the current element at the input index of the current element. All cups which were inserted earlier have smaller color. All cups which were inserted earlier but have a higher input index (=index in Fenwick Tree) would have to be swapped with the current element when sorting. Therefore for each element we count the sum of elements later in the array, which is just the sum of the whole array minus the prefix sum of the current element (again we need to take care to not count swaps with cups of same color). During the iteration over the list we sum up all of these suffix sums. This gives us the total number of inversions.

Note that there was a variant of this which is a bit worse: One could do the same approach "the other way round", directly indexing the Fenwick Tree by the colors, but this requires an array of size $O(\max_{i=1,\cdot,N} c_i)$ where c_i are the colors, i.e. an array of the size of the maximum number in the input, instead of the size of the input.

```
#include <bits/stdc++.h>
using namespace std;

vector<long long> ft;

void init(int n) {
    ++n; // use one-based indexing
    int l = 1;
    while (l < n)
        l *= 2;
    ft = vector<long long>(l, 0);
}

void update(int i, long long k) {
    ++i;
    while (i < ft.size()) {
        ft[i] += k;
```

```
18          i += i & -i;
19      }
20  }
21
22  long long query(int i) {
23      ++i;
24      long long prefsum = 0;
25      while (i > 0) {
26          prefsum += ft[i];
27          i -= i & -i;
28      }
29      return prefsum;
30  }
31
32  int main() {
33      int t;
34      cin >> t;
35      for (int tc = 1; tc <= t; ++tc) {
36          int k;
37          cin >> k;
38          vector<tuple<long long, long long, long long>> cg(k);
39          for (int i = 0; i < k; ++i) {
40              long long c, n;
41              cin >> n >> c;
42              cg[i] = make_tuple(c, i, n);
43          }
44          // sort by increasing colors, and if those are equal by increasing input
45          // index, so we do not count swaps of elements of the same color
46          sort(cg.begin(), cg.end());
47          init(k);
48          long long inv = 0;
49          for (int i = 0; i < k; ++i) {
50              inv += get<2>(cg[i]) * (query(k - 1) - query(get<1>(cg[i])));
51              update(get<1>(cg[i]), get<2>(cg[i]));
52          }
53          cout << "Case #" << tc << ": " << inv << "\n";
54      }
55  }
```

4 Strange Function

In this task, you were given the implementation of a procedure `funny` and you were asked to derive (ideally simpler) algorithms characterizing the behaviour of `funny`. There are inputs for which `funny` crashes or does not terminate.

Subtask 1: Small Values with Termination Guarantee (10 points)

For this subtask, it was sufficient to execute the provided implementation of `funny` in order to gather the required function outputs.

```cpp
#include <iostream>
using namespace std;

long long funny(long long a, long long b) {
    long long x = 1;
    while (x < b) {
        x *= 2;
    }
    long long s = x % b;
    while (a >= b) {
        a = (a & (x - 1)) + s * (a / x);
    }
    return a;
}

int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        long long a, b;
        cin >> a >> b;
        cout << "Case #" << t << ": ";
        cout << funny(a, b) << '\n';
    }
}
```

Subtask 2: Small Values (10 points)

Here, `funny` does not terminate for some of the inputs. Experimenting with such inputs a bit, one quickly arrives at the hypothesis that whenever the `while` loop does not terminate, it reaches a state from which the value of the variable `a` no longer changes. This turns out to be true. Hence it was enough to check for two subsequent equal values of `a` to detect non-termination. (The code below also exploits the fact that `funny` only returns non-negative values.)

```cpp
#include <iostream>
using namespace std;

long long funny(long long a, long long b) {
    long long x = 1;
```

```
    while (x < b) {
        x *= 2;
    }
    long long s = x % b;
    while (a >= b) {
        long long c = (a & (x - 1)) + s * (a / x);
        if (a == c) {
            return -1;
        }
        a = c;
    }
    return a;
}

int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        long long a, b;
        cin >> a >> b;
        cout << "Case #" << t << ": ";
        long long r = funny(a, b);
        if (r == -1)
            cout << "NOTHING\n";
        else
            cout << r << '\n';
    }
}
```

Subtask 3: Larger Values with Termination Guarantee (10 points)

Provided that the input/output code was able to deal with numbers that don't fit inside a 32-bit integer, the solution to subtask 1 was sufficient to score all points for this subtask. However, in order to obtain all points for the theoretical justification, it was important to find an alternative implementation that executes in $O(1)$ time.

It turns out that if `funny` terminates with a result, it simply computes $a\%b$, i.e. the remainder when performing integer division of a by b.

```
#include <iostream>
using namespace std;

int main() {
    int T;
    cin >> T;
    for (int t = 1; t <= T; t++) {
        long long a, b;
        cin >> a >> b;
        cout << "Case #" << t << ": ";
        cout << a % b << '\n';
    }
}
```

Subtask 4: Theoretical Justification for Subtask 3 (25 points)

In this subtask, you were asked to argue why your solution for subtask 3 produces the correct result, and to analyze its running time. For the presented solution, it is easy to see that the new implementation of `funny` runs in constant time ($O(1)$), as it involves only a single modulo operation. It is slightly more difficult to see that it is actually correct.[1]

We first consider the values of the variables x and s. It is easy to see that x is always a power of 2. (As it is initialized to 1, and subsequent modifying operations multiply its value by 2.) The final value x is the smallest power of two that is not smaller than b, as the first `while` loop uses this as the termination criterion. In particular, $x < 2 \cdot b$, because otherwise $x/2 \geq b$, and last iteration of the `while` loop would not have been executed. From $b \leq x < 2 \cdot b$ it follows that $s = x\%b = x - b$.

Also note that the value of the variable a always remains non-negative: it is non-negative in the beginning and the expression $a' = (a\&(x-1)) + s \cdot \left\lfloor \frac{a}{x} \right\rfloor$ is non-negative for non-negative a.

If $x = 2^i$, then the binary representation of x is $\cdots 01\underbrace{0\cdots 0}_{i \text{ times}}$ and the binary representation of $x - 1$ is $\cdots 00\underbrace{1\cdots 1}_{i \text{ times}}$. Therefore, the expression $a\&(x-1)$ extracts the value represented by the last i digits of the binary representation of a, which is precisely the same as $a\%2^i = a\%x$ (this is a well-known bit trick).

It follows that we can express a' as follows:

$$a' = (a\&(x-1)) + s \cdot \left\lfloor \frac{a}{x} \right\rfloor = a\%x + (x-b) \cdot \left\lfloor \frac{a}{x} \right\rfloor = a\%x + x \cdot \left\lfloor \frac{a}{x} \right\rfloor - b \cdot \left\lfloor \frac{a}{x} \right\rfloor.$$

For non-negative a and x, $a = a\%x + x \cdot \left\lfloor \frac{a}{x} \right\rfloor$, therefore

$$a' = a - b \cdot \left\lfloor \frac{a}{x} \right\rfloor.$$

In particular, $a'\%b = a\%b$. In other words, the value of $a\%b$ remains invariant throughout the execution of the procedure. If the program terminates, the final value of a satisfies $0 \leq a < b$ (otherwise the second `while` loop would not have terminated). It follows that the final value of a is $a\%b$ (evaluated with the initial values of a and b).

Subtask 5: Larger Values (10 points)

Again, the solution for subtask 2 was sufficient to score all points for this subtask. However, we present a solution that runs in constant time.

```cpp
#include <iostream>
using namespace std;

int main() {
    int T;
    cin >> T;
```

[1] For the solution to subtask 1, correctness is trivial, but running time is not. It turns out that it is $O(\log a + \log b)$, but this is harder to analyze.

```
    for (int t = 1; t <= T; t++) {
        long long a, b;
        cin >> a >> b;
        cout << "Case #" << t << ": ";
        if (b == 0 || a >= b && (b & (b - 1)) != 0 && ((a % b + b) & ~b) < b)
            cout << "NOTHING\n";
        else
            cout << a % b << '\n';
    }
}
```

Subtask 6: Theoretical Justification for Subtask 5 (35 points)

It suffices to argue that the solution prints `NOTHING` if and only if `funny(a, b)` does not produce a result. If $b = 0$, then the computation of $x\%b$ will crash the program and it will not produce a result, hence we should print `NOTHING` in this case.[2] On the other hand, if $a < b$ then the second `while` loop in `funny` is not executed a single time, and `funny` trivially terminates.

Finally we consider the case where $a \geq b$, and therefore the second `while` loop is executed at least once.

We again consider the expression $a' = a - b \cdot \lfloor a/x \rfloor$. From this, we can immediately see that $a' < a$ if $a \geq x$, and otherwse $a' = a$. Therefore, `funny` does not terminate if and only if the variable a is at one point in time assigned a value such that $b \leq a < x$. After this happens, a will be stuck at this value forever. As $x < 2 \cdot b$, and $a\%b$ remains invariant, there is at most one such value: $a\%b + b$ (using the initial values of a and b). It follows that `funny(a, b)` terminates if $a\%b + b \geq x$.

We will now argue that the converse is also true: `funny(a, b)` does not terminate if $a\%b + b < x$. There exists a k such that $a = a\%b + k \cdot b$.[3] As $a \geq b$, k is at least 1. Also recall that $x < b$. Using $a\%b + b < x$, we derive the following lower bound for a'.

$$
\begin{aligned}
a' &= a\%b + k \cdot b - b \cdot \left\lfloor \frac{a\%b + k \cdot b}{x} \right\rfloor \\
&= a\%b + b \cdot \left(k - \left\lfloor \underbrace{\frac{\overbrace{a\%b + b}^{<x} + \overbrace{(k-1)b}^{\leq (k-1)x}}{x}}_{<k} \right\rfloor \right) \\
&\geq a\%b + b \cdot (k - (k-1)) \\
&= a\%b + b \\
&\geq b.
\end{aligned}
$$

It follows that in this case, a can never be assigned a value below b, which means that `funny(a, b)` will never terminate.

Therefore, `funny(a, b)` produces a result unless $b = 0$ or $a \geq b$ and $a\%b + b < x$. In order to evaluate this expression, it is necessary to compute x. If we use the method given in the procedure

[2] From a mathematical standpoint, $x\%0 = x$, but hardware and programming languages do not support this. Note that technically, $x\%0$ exhibits *undefined behaviour* in C++ which means that you are not guaranteed to not get a result, but this is what happens in practice. The Python version is required to raise an exception, however.

[3] This k is given by $k = \left\lfloor \frac{a}{b} \right\rfloor$.

`funny(a, b)` to compute x, this computation will take time $\Theta(\log b)$. If we avoid computing x in case $a < b$, we can bring the computation time down to $O(\log(\min(a, b)))$.

However, there was a hint in the task description that there are solutions that execute in time $o(\log(\min(a, b)))$. The simplest way to achieve this sub-logarithmic running time is to observe that using bit shifts, powers of two can be computed in constant time: $2^i = 1 << i$. Since x is the smallest power of two that is not smaller than b, we can compute x in logarithmic time by using binary search on the exponent for a total running time of $O(\log(\log(\min(a, b))))$.

However, there are bit tricks that bring execution time down to $O(1)$ by avoiding computation of x. First consider the special case where b is a power of two. We can check for this efficiently: b is a power of two if and only if it is not zero, but $(b\&(b-1)) = 0$ (another well-known bit trick). In this case, it is easy to verify that `funny(a, b)` terminates very quickly ($x = b$, $s = 0$, $a' = a\%b$).

If b is not a power of two, then $b < x$. Since x is a power of two, this means that $(b\&x) = 0$: In binary, x has exactly one digit 1, and this is the digit of smallest value that has a larger value than any bit that is set in b. Because $x/2 < a\%b + b < 2 \cdot x$, if we want to check whether $a\%b + b$ is smaller than x, it suffices to compute $(a\%b + b)\&x$: We have $a\%b + b \geq x$ if and only if $a\%b + b$ has the bit with value x set. This is also the highest bit that can be set in $a\%b + b$. Furthermore we know that b has the bit with value $x/2$ set and that this is the highest bit set in b.

Now consider the expression $(a\%b + b)\&\sim b$. If $a\%b + b \geq x$, then also $((a\%b + b)\&\sim b) \geq x > b$, as the bit with value x is set in $a\%b + b$ and $\sim b$ also has the bit with value x set because $(b\&x) = 0$. On the other hand, if $a\%b + b < x$, then we will have that $((a\%b + b)\&\sim b) < b$, because $a\%b + b$ does not have any bits with value at least x set, and the bitwise and with $\sim b$ ensures that the highest bit set in b is not set in the result. As this bit is set in b, the inequality follows. But this means that $a\%b + b < x$ if and only if $((a\%b + b)\&\sim b) < b$.

Therefore, we can conclude that `funny(a, b)` returns a result unless $b = 0$ or $a \geq b$ and $(b\&(b-1)) \neq 0$ and $((a\%b + b)\&\sim b) < b$. This condition can be evaluated in constant time.

5 Stofl's Dissertation

This task is about creating a valid subthesis by tearing out some of the pages of a given thesis. A thesis was considered valid if its page numbers were in increasing order. You were asked to compute the length of the longest possible valid thesis, calculate the number of such theses or to construct a thesis given a specific number of different longest valid subtheses it has to contain. Mathematically speaking you had to compute the length of the longest increasing subsequence of a permutation, count all longest increasing subsequences of a given permutation and construct a permutation having exactly a certain number of longest increasing subsequences.

Subtask 1: Two page thesis (10 points)

Here you were given a list of N page numbers and had to compute whether there is a valid subthesis of length ≥ 2. One possibility was to check whether the sequence was decreasing, in which case the answer is NO and otherwise the answer is always YES. You could also verify this by comparing all pairs of page numbers. Or you could also run the code for the second subtask an check whether its solution was ≥ 2.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T, N;
    cin >> T;
    for (int t = 1; t <= T; ++t) {
        cout << "Case #" << t << ": ";
        cin >> N;
        vector<int> data(N);
        copy_n(istream_iterator<int>(cin), N, data.begin()); // read input
        if (is_sorted(data.rbegin(), data.rend()))
            cout << "NO\n"; // check if sequence is decreasing.
        else
            cout << "YES\n";
    }
}
```

Subtask 2: Longest thesis (20 points)

In this subtask you had to calculate the length of the longest thesis that can be formed by tearing out some of its pages. The longest such valid subthesis corresponds to the longest increasing subsequence. The longest increasing subsequence can be computed by a dynamic programming in $O(N^2)$ using the state $DP[position]->$ length of the longest increasing subsequence ending at $position$. It is possible to optimize this solution to $O(N \cdot \log N)$ using a binary-search or segment trees or even $O(N \cdot \log \log N)$, but this was not necessary to implement to earn the full score.

```cpp
// O(N^2) implementation of longest increasing subsequence.
#include <bits/stdc++.h>
using namespace std;
```

```cpp

int main() {
    int T, N;
    cin >> T;
    for (int t = 1; t <= T; ++t) {
        cout << "Case #" << t << ": ";
        cin >> N;
        vector<int> data(N);
        copy_n(istream_iterator<int>(cin), N, data.begin()); // read input
        vector<int> dp(N, 1); // dp[pos] -> length of LIS ending at pos
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < i; ++j)
                if (data[j] < data[i])
                    dp[i] = max(dp[i], 1 + dp[j]);
        cout << *max_element(dp.begin(), dp.end()) << "\n";
    }
}
```

Subtask 3: All the possibilities (20 points)

Here you had to count the number of longest valid subtheses. The solutions from the previous subtask can easily be adapted to not only compute the length of the longest increasing subsequence but also the number of such sequences. A solution running in $O(N^2)$ was sufficiently fast to fully solve this subtask.

```cpp
// O(N^2) implementation for counting the number of longest increasing
// subsequence.
#include <bits/stdc++.h>
using namespace std;
const long long MOD = 1e9 + 7;
int main() {
    int T, N;
    cin >> T;
    for (int t = 1; t <= T; ++t) {
        cout << "Case #" << t << ": ";
        cin >> N;
        vector<int> data(N);
        copy_n(istream_iterator<int>(cin), N, data.begin()); // read input
        vector<int> dp(N, 1); // dp[pos] -> length of LIS ending at pos
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < i; ++j)
                if (data[j] < data[i])
                    dp[i] = max(dp[i], 1 + dp[j]);
        vector<long long> cnt;
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < i; ++j)
                if (data[j] < data[i] && dp[j] == dp[i] + 1)
                    cnt[i] = (cnt[i] + cnt[j]) % MOD;
        cout << *max_element(dp.begin(), dp.end()) << "\n";
    }
}
```

```cpp
// O(N log N) implementation for counting the number of longest increasing
// subsequence using binary-search.
#include <bits/stdc++.h>
using namespace std;

const long long MOD = 1e9 + 7;
int main() {
    int testcases;
    cin >> testcases;
    for (int tt = 1; tt <= testcases; ++tt) {
        int N;
        cin >> N;
        vector<vector<int>>
            LIS; // decreasingly sorted vector of page numbers in
                 // which a LIS of a certain length ends
        vector<vector<long long>>
            LIScount; // prefix-sum over number of increasing
                      // subsequences of maximal length ending
                      // in that page number.
        for (int i = 0; i < N; ++i) {
            int curPage;
            cin >> curPage;

            // binary-search for the length of the LIS ending with the current
            // page.
            int pos = lower_bound(LIS.begin(), LIS.end(), curPage,
                                  [](vector<int> const &vec, int a) {
                                      return a > vec.back();
                                  }) -
                      LIS.begin();
            long long
                curCount; // number of increasing subsequences of maximal length
                          // ending with the current page number.
            if (pos == 0) {
                curCount = 1; // special case: current page has the smallest
                              // number so far.
            } else {
                int prePos = pos - 1;
                // binary-search on a decreasing vector we can use
                // reverse-iterators.
                int stackPos = LIS[prePos].rend() -
                               lower_bound(LIS[prePos].rbegin(),
                                           LIS[prePos].rend(), curPage);
                // calculate the number of LIS ending with the current page
                // using the
                // prefix-sums.
                curCount = LIScount[prePos].back();
                if (stackPos != 0)
                    curCount -= LIScount[prePos][stackPos - 1];
            }
            if (pos == LIS.size()) {
                // special case: current page has a higher LIS than all previous
                // pages.
                LIS.push_back(vector<int>(1, curPage));
                LIScount.push_back(vector<long long>(1, curCount));
            } else {
```

```
                // Add current page.
                LIS[pos].push_back(curPage);
                // Update prefix sum.
                LIScount[pos].push_back(LIScount[pos].back() + curCount);
            }
            // calculate modulo 1e9+7 to avoid overflows.
            LIScount[pos].back() = (LIScount[pos].back() % MOD + MOD) % MOD;
        }
        cout << "Case #" << tt << ": " << LIScount.back().back() << "\n";
    }
}
```

Subtask 4: Reconstruction (20 points)

Now you had to construct a thesis having exactly K longest valid subtheses. Mathematically speaking you had to find a permutation of the numbers from 1 to N which has exactly K longest increasing subsequences. One possibility is to first build a permutation having 2^l longest increasing subsequences. This can be done with an increasing sequence of l decreasing twin-pairs $(2 \cdot k, 2 \cdot k - 1)$: The sequence $2, 1, 4, 3, 6, 5, \ldots, 2 \cdot k, 2 \cdot k - 1, \ldots, 2 \cdot l, 2 \cdot l - 1$ has exactly 2^l increasing subsequences of length l. (From each twin-pair you could pick exactly one element.) We now construct a sequence with K longest increasing subsequences by writing K in base 2 by choosing l such that $2^l \geq K$ and adding an increasing "collector rail" of the numbers $x + 1, \ldots, x + l$ where $x = 2 \cdot l$ to our sequence.

For the i-th twin pair, if the i-th bit is set, we put the collector number after the first element: $\ldots, 2 \cdot k, x + k, 2 \cdot k - 1, \ldots$ and if the i-th bit is not set, we put the collector number in front of the whole pair: $\ldots, x + k, 2 \cdot k, 2 \cdot k - 1, \ldots$. The length of the longest increasing subsequence is now $l + 1$. This means that any longest increasing subsequence has to contain some number of the collector rail and some number of a twin pair. As all the numbers in the collector rail are larger than the numbers in the twin-pair, each longest increasing subsequence has to start with some twin-pair numbers and switch to collector rail numbers at some point. The number of subsequences entering the collector rail at the number $x + i$ is exactly 2^i if the i-th bit was set in K and 0 otherwise. The total number of longest increasing subsequences is thus K. Any leftover numbers due to $n > 3 \cdot l$ can simply be appended as an increasing sequence of numbers bigger that all the previous numbers. These numbers will be part in all the longest increasing subsequences at thus won't chance the number of them.

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

using namespace std;

int main(int, char **) {
    int T;
    cin >> T;

    for (int t = 1; t <= T; ++t) {
        cout << "Case #" << t << ": ";

        int N;
        long long K;
        cin >> N >> K;

```

```
17          // If K == 1, build the smallest permutation in order
18          // to have one increasing subsequence of length N.
19          if (K == 1) {
20              for (int i = 1; i <= N; i++)
21                  cout << i << ' ';
22              cout << endl;
23              continue;
24          }
25
26          // Find the most significant bit set to 1 in K
27          // B := 1+floor(log2(K))
28          int B = 0;
29          while (K >= (1LL << B))
30              ++B;
31
32          // L,R are two fragments of the permutation to be concatenated
33          // afterwards.
34          // R will contain B-1 'twin' in the form (1+R_i ,R_i), with R_i
35          // increasing
36          // between
37          // pairs. This allows exactly 2^(B-1) increasing subsequences of length
38          // B-1.
39          // L will increasing subsequence of length at most B, and will be used
40          // to
41          // select
42          // bits from number K, to build appropriate amount of subsequences of
43          // length
44          // B.
45          //
46          // Specifically, if we add increasing sequence of M numbers to L, while
47          // the
48          // last one
49          // is smaller than R_M but larger than R_{M+1}, we will get exactly
50          // 2^(B-M-1) subs.
51          // of length B.
52          vector<int> L, R;
53          L.reserve(B);
54          R.reserve(B * 2);
55          int skipped_numbers_in_L = 0, next_number = 1;
56
57          for (int b = B - 1; b >= 0; --b) {
58              if (K & (1LL << b)) {
59                  // This bit needs to be selected. There are already some numbers
60                  // in L, we need to add C more, larger than everything in R so
61                  // far
62                  // to have exactly B-b numbers there.
63                  for (int i = next_number;
64                       i <= next_number + skipped_numbers_in_L; i++) {
65                      L.push_back(i);
66                  }
67                  next_number += skipped_numbers_in_L + 1;
68                  skipped_numbers_in_L = 0;
69              } else {
70                  // This bit is not selected, remember that it was skipped
71                  ++skipped_numbers_in_L;
72              }
```

```
73
74              // Append 1+R_i and R_i. Those have to be the smallest unused
75              // numbers
76              // so that we cannot add any more increasing subsequences using
77              // these two
78              // numbers
79              if (b > 0) {
80                  R.push_back(next_number + 1);
81                  R.push_back(next_number);
82                  next_number += 2;
83              }
84          }
85
86          // Now we connect all bits together. First, we may have some unused
87          // numbers
88          // from 1..N,
89          // and those will be larger than anything in L or R. Put them in
90          // decreasing
91          // order in the
92          // front of the solution, so that they can only be part of a increasing
93          // subsequence of
94          // length 1.
95          cerr << "used: " << next_number - 1 << "\n";
96          for (int i = N; i >= next_number; i--)
97              cout << i << ' ';
98
99          // Now, append L - up to B numbers used to select bits from K.
100         for (size_t i = 0; i < L.size(); ++i)
101             cout << L[i] << ' ';
102
103         // At last, add everything from B
104         for (size_t i = 0; i < R.size(); ++i)
105             cout << R[i] << ' ';
106
107         // Each subsequence of length B now looks as follows -
108         // select M lowest numbers from L, and then B-M increasing numbers from
109         // R,
110         // each
111         // number coming from a 'twin' pair. This gives 2^(B-M) sequences of
112         // length
113         // B.
114         //
115         // Note that if we don't pick the smallest numbers from L, or if we pick
116         // more than
117         // M, the largest of them will be larger than both the numbers in the
118         // first
119         // pair
120         // we used from R. Similar observation holds for selecting numbers from
121         // R.
122         cout << endl;
123     }
124 }
```

Subtask 5: Reconstruction with Few Pages (30 points)

The solutions from the previous subtask achieves $C = 3$. This however is not optimal. By using a base b instead of base 2 one can get $C = \frac{(b+1)}{\log_2(b)}$ with for $b = 4$ results in $C = 2.5$. This is the best solution that we are aware of, but we haven't got a proof that this is optimal.

What we can prove is that $C \geq \frac{3}{\log_2 3}$. For this we want to bound the number K of Longest increasing subsequences of a permutation of length N. Let L be the length of a longest increasing subsequence and a_i be the number of pages p, for which the longest increasing subsequence ending in p has length i. The number of longest increasing subsequences is bounded by $\prod_{i=1}^{L} a_i$, as every longest increasing subsequence has length L and the length of the j-th element of the longest increasing subsequence has a longest increasing subsequence ending with it of length j. All longest increasing subsequences consist of picking one page from each set of pages having the same length of longest increasing subsequence ending with them. As there are a_i numbers in the i-th such set there are exactly $\prod_{i=1}^{L} a_i$ such possibilities. Now suppose there exist two indices i, j such that $a_j < a_i - 1$. If we replace a_j with $a_j + 1$ and replace a_i with $a_i - 1$, we increase the product $\prod_{i=1}^{L}$, since

$$(a_i - 1) \cdot (a_j + 1) = a_i \cdot a_j + a_i - a_j - 1 > a_i \cdot a_j$$

we can thus assume that $\max a_i - \min a_i \leq 1$, since we only increase $\prod a_i$ this way. Let $a = \min a_i$. We can rewrite the product to

$$\prod a_i = a^x \cdot (a+1)^y$$

for x and y such that $a \cdot x + (a+1) \cdot y = N$. Thus

$$\prod a_i = a^{\frac{x}{a}} \cdot (a+1)^{\frac{N-x}{a+1}}$$

where $0 \leq x \leq N$. This function is convex in x and thus is maximal when $x = 0$ or $x = N$. Let $b = a$ if this is maximal at $x = N$ and $b = a + 1$ otherwise. We can rewrite to

$$\prod a_i \leq b^{(N/b)}$$

This function has a minimum at $b = e$. As b is an integer we get a maximum at $b = 3$. What we have proven is that $K = \prod a_i \leq b^{(N/b)} \leq 3^{(N/3)}$. By taking the logarithm we get

$$\frac{3}{\log_2(3)} \leq \frac{N}{\log_2(K)}$$

which is the lower bound $C \geq \frac{3}{\log_2(3)}$.

If anyone can prove a tighter lower bound or find a better construction, let us know at info@soi.ch!

```cpp
/*
 * prints (base+1) * log_base(K) numbers, for any integer base>=2
 * this achieves C=2.5 when base=4
*/
#include <bits/stdc++.h>

using namespace std;

// applies coordinate compression to transform v into a permutation of the
// numbers 1 to n.
void compress(vector<int> &v) {
```

```cpp
    vector<int> v2 = v;
    sort(v2.begin(), v2.end());
    for (auto &e : v)
        e = lower_bound(v2.begin(), v2.end(), e) - v2.begin() + 1;
}

int main() {
    const int base = 4;
    int T;
    cin >> T;
    for (int tt = 1; tt <= T; ++tt) {
        cout << "Case #" << tt << ": ";
        int N;
        long long K;
        cin >> N >> K;
        vector<int> ret;

        int b = 1, t = N * base;

        for (long long basePow = 1; basePow <= K; basePow *= base) {
            for (int i = 0; i <= base; ++i) {
                // base construction for powers of base
                if (i != 0)
                    ret.emplace_back(b + base - i);
                if (i == (K / basePow) % base) {
                    // collector rail
                    ret.emplace_back(t++);
                }
            }
            b += base;
        }
        // fill up with large increasing numbers
        while ((int)ret.size() < N)
            ret.emplace_back(N * N + (int)ret.size());
        // compress to permutation
        compress(ret);
        copy(ret.begin(), ret.end(), ostream_iterator<int>(cout, " "));
        cout << "\n";
    }
}
```

6 Submarine

Subtask 1: Find a path (20 points)

In this subtask you had to guide the ships to the destination and the submarines didn't attack.

To find a path from start points to destinations we use BFS. You can do this either from each start point or from each destination and then store the found paths. There are mainly two things to look out for: choose only start points that have a path to a destination and make sure that there are never two ships on the same position.

Subtask 2: Destroy all ships (25 points)

In this subtask you controlled the submarines and you had to destroy all ships.

As the ships move at random there was no point in defending any particular destination. As it turns out it's enough to move randomly and attack all found ships. Basically you can copy the sample bot.

Subtask 3: Creativity Tournament (50 points)

As this is the creativity task, there are many possible solutions and we don't know the best one either.

We played several rounds on several maps to determine the quality of the submitted algorithms. According to those games we calculated a ranking and awarded points according to it.

Unfortunately most submitted programs contained more or less serious bugs we had to fix before we were able to test the algorithms. Depending on what we had to fix we deducted some points.

For the base score before the deduction we used the following table:

- 50 points for ranks 1, 2 and 3
- 40 points for ranks 4 and 5
- 25 points for ranks 6 and 7
- 20 points for ranks 8 and 9
- 15 points for rank 10
- 10 points for rank 11
- 5 points for rank 12