New Zealand Informatics Competition 2019
Round 3 Solutions

July 17, 2019

Overview

Questions

1. **Quackers for Crackers**: Calculation and output
2. **Keyboard**: Looping and strings
3. **Boranges**: Arrays, optimisation
4. **You, Robot**: Breadth-First Search
5. **Tournament**: Ad-hoc, brute force

The solutions to these questions are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution. e.g. $O(N)$. There is an explanation of Big O at the end of this document.

Resources

Ever wondered what the error messages mean?

`https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf`

Read about how the server marking works:

`https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf`

Other resources: `https://www.nzoi.org.nz/nzic/resources.html`

Tips for next time

Remember, this is a contest. The only thing we care about is that your code runs. It doesn't need to be pretty or have comments. There is also no need to worry about invalid input. Input will always be as described in the problem statement. For example, the code below is not necessary.

```
# Not needed
def error_handling(prompt):
   while True:
       try:
           tea = int(input(prompt))
           if tea < 0 or tea > 100:
               print('That was not a valid integer!')
           else:
               return tea
       except ValueError:
           print('Not a valid integer')
...
```

There are a few other things students can do to improve their performance in contests.

Move on to the next question

If you are spending too much time on a question, move on. There could be easy subtasks waiting in the next question. Often, you will think of a solution to your problem while working on another question. It also helps to come back to a question with a fresh pair of eyes.

Take time to read the questions

Don't underestimate the value of taking time to read and understand the question. You will waste exponentially more time powering off thinking you have the solution only to discover you missed something obvious.

In the real world, it is very rare to be provided with a problem in a simplistic form. Part of the challenge with these contests is reading and understanding the question, then figuring out what algorithm will be needed. Before submitting, check to make sure you have done everything the question asks you to do.

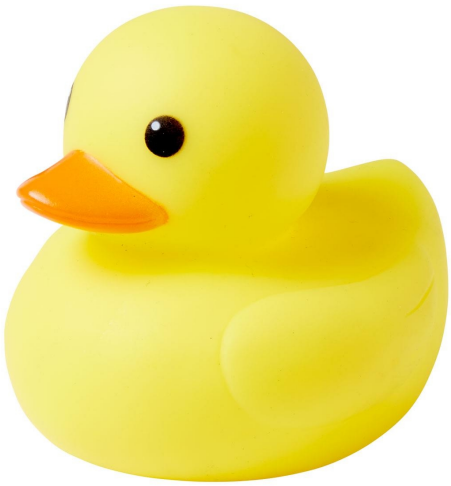
Test your code first!

It is much more time efficient to test your solutions on your own computer first. It can take a while to wait for the server to run your code. It is far more efficient to test it yourself first, submit it, then start reading the next question while you wait for your previous solution to be marked by the server.

While testing, remember to add some edge case tests. For example, if a question specifies "$1 \leq N \leq 10$" then try out an input where $N = 1$. Think of other tricky inputs that might break your code. Look at the feedback the server is giving you.

Use a rubber duck

`https://en.wikipedia.org/wiki/Rubber_duck_debugging`



Quackers for Crackers

https://train.nzoi.org.nz/problems/1032

Python 3 Examples

```python
# example 1
S, D, C, P = [int(input()) for _ in range(4)]

total = S
days = 0
while total <= P:
    total += D - C
    days += 1

print(days - 1)  # -1 to exclude day of the attack
```

```python
# example 2
S, D, C, P = [int(input()) for _ in range(4)]
print((P-S) // (D-C))  # use integer division
```

C++ Example

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

int main() {
    int S, D, C, P;
    cin >> S >> D >> C >> P;
    int days = (P-S) / (D-C);
    cout << days << '\n';
}
```

Keyboard

https://train.nzoi.org.nz/problems/1033

Subtask 1

For the first subtask, we require one keypress for each character, and an additional keypress for each character that has changed from the previous character.

```
s = input()
count = len(s)
prev = 'A'  # always starts with A
for c in s:
    if c != prev:
        count += 1
    prev = c
print(count)
```

Subtask 2

For the full solution, we need to calculate the minimum number of keypresses needed to change between any two letters.

Python Example

In Python, we can get the number representation for a character using `ord()`. Using `map(ord, s)` converts each character in the string `s` to its corresponding number. By subtracting two characters from each other, we can calculate the distance between them. However, there are two ways to move between characters.

For example, we can get from A to Z with 25 keypresses right, or just one keypress left. Hence, we can use the mod operator `%` to wrap around the 26 characters of the alphabet and choose the option with the minimum number of keypresses.

```python
s = input()
count = len(s)
prev = ord('A')  # always starts with A
for c in map(ord, s):
    left = (prev - c) % 26
    right = (c - prev) % 26
    count += min(left, right)
    prev = c
print(count)
```

C++ Example

In C++, characters are also numbers. Hence, we can subtract two characters from each other to calculate the number of keypresses needed to move left or right between them. The remainder operator is used to wrap around the 26 characters in the alphabet. However, to ensure we don't get a negative remainder, we add 26 beforehand. Since we want the optimal number of keypresses, we always take the minimum of the two directions.

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

int main() {
    string s;
    cin >> s;
    char prev = 'A';
    int count = s.length();
    for(char c : s) {
        int left = (prev - c + 26) % 26;
        int right = (c - prev + 26) % 26;
        count += min(left, right);
        prev = c;
    }
    cout << count << endl;
}
```

Boranges

https://train.nzoi.org.nz/problems/1031

This problem might seem easy at first, but only a few students managed to work out the full solution.

Subtask 1

For subtask 1, we can loop through each height between 1 and the maximum tree height and calculate which heights are valid.

Python Example

```python
N, c = map(int, input().split())
hs = [int(input()) for _ in range(N)]

h_max = max(hs)
n_deployed = 0
for h in range(1, h_max + 1):
    profit = 0
    for k in range(N):
        if hs[k] >= h:
            profit += h * 5

    if profit > cost:
        n_deployed += 1

print(n_deployed)
```

C++ Example

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

int main() {
    int N; long long c;
    cin >> N >> c;
    vector<long long> hs(N);
    for (int j = 0; j < N; j++) cin >> hs[j];

    long long n_deployed = 0;
    long long h_max = *std::max_element(hs.begin(), hs.end());
    for (long long h = 1; h <= h_max; h++) {
        long long profit = 0;
        for (int k = 0; k < N; k++) {
            if (hs[k] >= h) profit += h * 5;
        }
        if (profit > c) n_deployed++;
    }
    cout << n_deployed << endl;
}
```

However, this solution to far too slow for subtask 2. Notice that in subtask 2 the trees can be up to 20 billion high. Then for each tree you have to loop through 100,000 trees. That's a total of 2 quadrillion iterations. Try creating a program with a loop which goes from 1 up to 2 quadrillion and see how long it takes. We need a better solution.

Subtask 2

For the full solution, it helps to notice two things.

- It doesn't matter which order the trees are in.
- If the trees are in sorted order, it is easy to work out how many boranges we can collect at each height.

If the trees are in sorted order from highest to lowest (left to right), then the number of boranges we can collect at height h can be calculated by finding the

greatest index i such that the i-th tree is at least h high (where trees numbered starting from 1). With a sorted list, all the trees to the right of tree i will be shorter than h, thus providing no boranges. Since the remaining i trees are all at least h high, the number of boranges collected at height h is simply the product ih. The key advantage here is that we no longer need to calculate the number of boranges collected at every possible height. Instead, we can do a calculation for each tree height. Since there are only up to 100,000 trees, our program will be much much faster.

Suppose we have five trees – three of height 2, and two of height 10 – and the cost c is 40c. Our sorted list would be $[10, 10, 2, 2, 2]$. We consider the two different heights, starting at 10. Moving to the second 10, at position $i = 2$ (remember we are indexing from 1), we find the greatest height h at which picking boranges from i trees is no longer profitable. Since each borange is worth 5c, it must hold that $5ih \leq c$ for the deployment to be non-profitable. Rearranging for h we get $h \leq c/5i$, and as we are interested in the greatest integer height, we end up with $h = \lfloor c/5i \rfloor$.[1] In this case, $h = \lfloor 40/(5 \times 2) \rfloor = 4$. Sure enough, at $h = 4$ we can calculate the total value collected as $2 \times 4 \times 5 = 40c$, which is not profitable. This means that all heights from 5 to 10 are profitable and heights 3 and 4 are not profitable. Note that we cannot yet determine the profitability at height 2 and below since the number of trees has changed at this height. Therefore, we move to $i = 5$ and recalculate $h = \lfloor 40/(5 \times 5) \rfloor = \lfloor 1.6 \rfloor = 1$. In this case, the height $h = 1$ is not profitable but $h = 2$ is. Overall, the heights 10, 9, 8, 7, 6, 5, and 2 are profitable, giving a final answer of 7 deployments. However, notice that we only needed to do two calculations at two different heights instead of calculating the profit at each of the 10 heights!

Python Example

```python
N, c = map(int, input().split(" "))
hs = [int(input()) for _ in range(N)]

hs.sort(reverse=True)  # largest to smallest
hs.append(0)  # so that k + 1 doesn't exceed the list bounds

n_deployed = 0
for k in range(N):
    # one based indexing
    i = k + 1
```

[1] $\lfloor x \rfloor$ denotes the floor of x (rounded down to the nearest integer).

```python

    # calculate first non-profitable height or the next
    # tree height if all heights in between are profitable
    h = max(c // (i * 5), hs[k + 1])

    # add profitable heights between current height and h
    if h < hs[k]:
        n_deployed += hs[k] - h

print(n_deployed)
```

C++ Example

When using C++, make sure to use `long long` integers since 20 billion cannot fit in a 32-bit integer. The answer might also be larger than a 32-bit integer.

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

int main() {
    int N; long long c;
    cin >> N >> c;
    vector<long long> hs(N + 1);  // + 1 so we don't overrun k + 1
    for (int j = 0; j < N; j++) cin >> hs[j];
    sort(hs.begin(), hs.end(), std::greater<long long>());

    long long n_deployed = 0;
    for (int k = 0; k < N; k++) {
        int i = k + 1;

        // calculate first non-profitable height or the next
        // tree height if all heights in between are profitable
        long long h = std::max(c / (i * 5), hs[k + 1]);

        // add profitable heights between current height and h
        if (h < hs[k]) n_deployed += hs[k] - h;
    }

    cout << n_deployed << endl;
}
```

You, Robot

The first part of this problem is assembling a representation of the graph (where a graph is a series of vertices and edges, or rooms and passages). To do this, we create an adjacency list.

An adjacency list keeps track of all the vertices (rooms) which neighbour each other. For instance, if the vertex/room numbered 0 has two neighbours, 1 and 2, the adjacency list should look like:

```
0: 1, 2
1: 0
2: 0
```

(Note that 0 is also listed as neighbours of 1 and 2, since we can travel both ways along the passages in this problem.)

To assemble the adjacency list, we need to create a 2-dimensional array in Python. Then, the first dimension of the array refers to each vertex, and the second dimension stores all of that vertex's neighbours. The code snippet below is one way to do this.

```
N, P, S, E = map(int, input().split())
passages = [[] for i in range(N)]
for i in range(P):
    A, B = map(int, input().split())
    passages[A].append(B)
    passages[B].append(A)
```

(You may have stored your neighbours in a different way - if you are interested, look up adjacency matrices, and compare this strategy to an adjacency list.)

Now that we've stored the passages, we can focus on completing the problem.

Subtask 1

The first subtask states that the rooms will be in one line of passages, with no three-way or bigger intersections. This makes our job easier, as we only have to find the number of rooms in between the start and the exit rooms. We loop through the vertices, moving along the chain of rooms in one direction. As we move, we increment a counter, which will then tell us the distance once we have reached the exit.

There are two possible directions to choose, so to avoid any problems, you can choose the initial direction arbitrarily and try it. If there are problems or the exit is never reached, simply switch to the other direction. One of the two directions will always work for this subtask.

Subtask 2

This subtask is slightly easier than the full problem simply because the number of vertices and edges we have to deal with is less. The solution to this subtask is similar to the full problem, but the lower constraints allows for some inefficiencies in the implementation - for instance, re-visiting rooms unnecessarily.

Subtask 3

The basic strategy we use to solve this problem is moving through the graph systematically, starting at the start vertex and finishing when we reach the exit vertex.

To solve the problem, we need to keep track of various things.

1. We need to know which vertices we have already visited, so we need a list `visited` which stores `True` if the vertex has been visited, `False` if not.
2. We need a list of vertices to explore next. We can delete these as we visit them, and add when we discover new neighbours. This going to be a first-in-first-out structure called a queue (in Python we can use a `deque`).
3. As we explore the graph, we need to know how far away from the start room we are. Since we are only interested in the distance to the exit, our solution below uses a queue maintaining `(room, distance)` tuples / pairs,

but note that other implementations might store the distances in a separate list instead.

We can now traverse (explore) the graph using an algorithm called Breadth-First Search (BFS). There are some good resources available online if you are interested in learning more about BFS. The algorithm description is as follows:

1. Add starting vertex to the queue.
2. While the queue has vertices remaining in it:
   (a) Take the vertex and its current distance value from the top of the queue, and store it (make sure it is removed from the queue!)
   (b) If the vertex has already been visited, then continue back to 2.
   (c) Set the visited value of the vertex to True.
   (d) For each of the vertex's neighbours, add the neighbour to the queue with the current distance increased by one.

The code for the full solution is shown below. Note that there are other algorithms you can use to solve this problem!

Python Example

```
from collections import deque

N, P, S, E = map(int, input().split())
passages = [[] for i in range(N)]
for i in range(P):
    A, B = map(int, input().split())
    passages[A].append(B)
    passages[B].append(A)

q = deque()
q.append((S, 0))
visited = [False] * N

while len(q) > 0:
    room, dist = q.popleft()
    if visited[room]: continue
    visited[room] = True
    # at this point, it is guaranteed that `dist`
    # is the shortest distance from `S` to `room`
```

```
    if room == E:
        print(dist)
        break
    for next_room in passages[room]:
        q.append((next_room, dist+1))
```

C++ Example

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

int main() {
    int n,s,e,p,a,b;
    cin >> n >> p >> s >> e;

    vector<vector<int>> adj(n);
    while(p--) {
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    queue<pair<int,int>> q;
    q.push({s, 0});
    vector<bool> visited(n);

    while(!q.empty()) {
        int room = q.front().first;
        int dist = q.front().second;
        q.pop();
        if(visited[room]) continue;
        visited[room] = 1;
        if(room == e) {
            cout << dist << endl;
            break;
        }
        for(int next : adj[room]) {
            q.push({next, dist+1});
        }
    }
}
```

Tournament

https://train.nzoi.org.nz/problems/883

Subtask 1

For this subtask, N is at most 64. As the bounds are low, we can try all possible swaps of players, which is $O(N^2)$, and for each swap, we will simulate the tournament in $O(N)$ time, keeping track of the number of times Toddy wins. We also need to remember that if swapping players does not help Toddy win any more matches than he already could, we must output -1 instead. Thus, we simulate the tournament once more with the initial bracket ordering. Overall, this solution runs in $O(N^3)$ time.

Simulating the tournament can be done by simply adding the winner of every match (the maximum of every 2 players) into a new list, and repeating this for each of the K rounds. As there are only $N - 1$ matches in total, each simulation runs in $O(N)$ time.

```
from collections import defaultdict

def simulate(skills):
    wins = 0
    while len(skills) > 1:  # repeat until 1 player is left
        winners = []
        # determine winners (max of every 2 players)
        for i in range(0, len(skills), 2):
            winners.append(max(skills[i], skills[i+1]))
        wins += winners.count(toddy)  # add 1 if Toddy wins
        skills = winners  # winners advance to the next round
    return wins

```

```
n, k = map(int, input().split())
skills = list(map(int, input().split()))
toddy = skills[0]
ways = defaultdict(int)
initial_wins = simulate(skills)

# to avoid double counting swaps, the inner for loop starts from i+1
for i in range(n):
    for j in range(i+1, n):
        skills[j], skills[i] = skills[i], skills[j]  # swap players
        wins = simulate(skills)
        ways[wins] += 1
        skills[j], skills[i] = skills[i], skills[j]  # swap back

max_wins = max(ways)
if initial_wins == max_wins:
    ways[max_wins] = -1
print(max_wins)
print(ways[max_wins])
```

Subtask 2

With a limit of $N \leq 2048$, an $O(N^3)$ solution is too slow for subtask 2, so we need to speed up our solution. We can do this by *reducing the search space*.

Consider a simulation of the initial bracket, and let p be the first player Toddy loses to. It shouldn't be too hard to see that any optimal swap includes either Toddy or player p – in order to gain more wins, either Toddy has to move to some other position in the bracket, or player p needs to be swapped out with some other player who is weaker than Toddy. Thus, we only need to check $O(N)$ swaps, improving our brute-force solution to run in $O(N^2)$ time.

The main challenge with this is finding which player Toddy first loses to, but since Toddy is always the first player, there is actually a simpler solution: we just iterate across the players in the initial bracket ordering until we find one that is stronger than Toddy. You might notice that this doesn't always find the first player Toddy loses to, since the player we've found might lose to someone else further in the bracket before they can play against Toddy. But in such cases, it isn't possible to gain any more wins without swapping Toddy anyway (since both of them are better than Toddy), so this isn't a problem for our solution.

```python
from collections import defaultdict

def simulate(skills):
    wins = 0
    while len(skills) > 1:  # repeat until 1 player is left
        winners = []
        for i in range(0, len(skills), 2):
            winners.append(max(skills[i], skills[i+1]))
        wins += winners.count(toddy)  # add 1 if Toddy wins
        skills = winners
    return wins

n, k = map(int, input().split())
skills = list(map(int, input().split()))
toddy = skills[0]
ways = defaultdict(int)
initial_wins = simulate(skills)
first_loss = None

for x in range(1, n):
    if skills[x] > skills[0]:
        first_loss = x
        break

for x in range(1, n):
    # try swapping Toddy with player x
    skills[0], skills[x] = skills[x], skills[0]
    wins = simulate(skills)
    ways[wins] += 1
    skills[0], skills[x] = skills[x], skills[0]

    # try swapping the first player Toddy loses to with player x
    if first_loss:
        skills[first_loss], skills[x] = skills[x], skills[first_loss]
        wins = simulate(skills)
        ways[wins] += 1
        skills[first_loss], skills[x] = skills[x], skills[first_loss]

max_wins = max(ways)
if initial_wins == max_wins:
    ways[max_wins] = -1
print(max_wins)
print(ways[max_wins])
```

Subtask 3

To solve the full problem, we need a faster way of simulating the tournament.

A useful observation is that we do not care about the relative skill levels of players other than Toddy – we only care about whether a player is stronger than Toddy or not. For each player, we will store a Boolean value – `True` (1) if that player is stronger than Toddy, and otherwise `False` (0). Then, for each match Toddy plays, we only need to consider all the players within the sub-bracket under that match. If any player in that sub-bracket is stronger than Toddy, then Toddy cannot win that match.

Our solution precomputes the number of players stronger than Toddy under the sub-bracket for each match, allowing us to simulate the initial tournament in $O(\log N)$ time. The only remaining problem is figuring out how to handle the swaps of players.

If we start numbering players and matches from 0, then for some player at position p and round r, we can determine the match that player corresponds to with $\lfloor p/2^r \rfloor$. In most languages we can simplify this to `p >> r`.[2] Then, we know that any two players p_1, p_2 belong to the same sub-bracket at round r if $\lfloor p_1/2^r \rfloor = \lfloor p_2/2^r \rfloor$. This allows us to adjust our precomputed values to handle each swap in $O(1)$ time, by checking whether each swapped player belongs to the same sub-bracket as Toddy.

Since we're trying $O(N)$ swaps, and each swap can be checked in $O(\log N)$ time, the overall run time of this solution is $O(N \log N)$.

Python Example

```
from collections import defaultdict

def simulate(p, i, j):
    """Returns the number of matches the player at position
    `p` would win if players `i` and `j` were swapped."""
    wins = 0
    for r in range(1, k+1):
        stronger_players = stronger[r][p >> r]

```

[2] `x << y` and `x >> y` denote bitwise shift operations, corresponding to $2^y x$ and $\lfloor x/2^y \rfloor$ respectively. See `https://wiki.python.org/moin/BitwiseOperators` for more on bitwise shifts.

```
        # adjust to account for swap of players i and j
        if (i >> r) == (p >> r):  # player i within sub-bracket
            stronger_players += stronger[0][j] - stronger[0][i]
        if (j >> r) == (p >> r):  # player j within sub-bracket
            stronger_players += stronger[0][i] - stronger[0][j]

        if stronger_players == 0:
            wins += 1
    return wins

n, k = map(int, input().split())
skills = list(map(int, input().split()))
toddy = skills[0]
ways = defaultdict(int)
first_loss = None
stronger = [[x > skills[0] for x in skills]]
# stronger[0][m] stores whether player m is stronger than Toddy.

for x in range(1, n):
    if stronger[0][x]:
        first_loss = x
        break

# precompute no. of players stronger than Toddy under each match
# stronger[r][m] stores the number of players stronger than Toddy
# under the sub-bracket of match m of round r (with m starting at 0)
for r in range(1, k+1):
    stronger.append([])
    # using 0-based matches, so match m of round r is played against
    # the winners of matches m*2 and m*2+1 from round r-1
    for m in range(n >> r):
        stronger[r].append(stronger[r-1][m*2] + stronger[r-1][m*2+1])

initial_wins = simulate(0, 0, 0)
for x in range(1, n):
    # try swapping Toddy with player x
    wins = simulate(x, 0, x)
    ways[wins] += 1

    # try swapping the first player Toddy loses to with player x
    if first_loss:
        wins = simulate(0, first_loss, x)
        ways[wins] += 1
```

```

max_wins = max(ways)
if initial_wins == max_wins:
    ways[max_wins] = -1
print(max_wins)
print(ways[max_wins])
```

C++ Example

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

int n,k;
int skills[1 << 16];
int stronger[17][1 << 16];

int simulate(int p, int i, int j) {
    int wins = 0;
    for(int r = 1; r <= k; r+) {
        int stronger_players = stronger[r][p >> r];

        // adjust to account for swap of players i and j
        if(p >> r == i >> r) {  // player i within sub-bracket
            stronger_players += stronger[0][j] - stronger[0][i];
        }
        if(p >> r == j >> r) {  // player j within sub-bracket
            stronger_players += stronger[0][i] - stronger[0][j];
        }

        if(stronger_players == 0) wins++;
    }
    return wins;
}

int main() {
    cin >> n >> k;
    for(int i = 0; i < n; i++) cin >> skills[i];

    int firstLoss = 0;
    int maxWins = 0;
    int ways = 0;

```

```
    for(int i = 0; i < n; i++) {
        skills[i] = skills[i] > skills[0];
        if(skills[i] && !firstLoss) firstLoss = i;
    }

    for(int r = 1; r <= k; r++) {
        for(int m = 0; m < (n >> r); m++) {
            stronger[r][m] = stronger[r-1][m*2] + stronger[r-1][m*2+1];
        }
    }

    int initialWins = simulate(0, 0, 0);
    for(int i = 1; i < n; i++) {
        // try swapping Toddy with player i
        int wins = simulate(i, 0, i);
        if(wins > maxWins) maxWins = wins, ways = 1;
        else if(wins == maxWins) ways++;

        // try swapping the first player Toddy loses to with player i
        if(firstLoss) {
            wins = simulate(0, firstLoss, i);
            if(wins > maxWins) maxWins = wins, ways = 1;
            else if(wins == maxWins) ways++;
        }
    }

    if(initialWins == maxWins) ways = -1;
    cout << maxWins << endl << ways << endl;
}
```

There are also other similar solutions that involve using data structures such as prefix sum arrays or segment trees, allowing us to efficiently calculate the number of stronger players in any given range.

A better solution

If we abandon the brute-force approach, an even faster (and shorter) solution can be found. For each match in the initial tournament bracket at round r, we are only interested in four types of cases:

1. If the match includes Toddy, and no one in the sub-bracket is stronger, Toddy can win r matches without swapping players.

2. If the match includes Toddy, and one player in the sub-bracket is stronger, Toddy can win r matches by swapping that one player with any weaker player outside the sub-bracket.
3. If the match does not include Toddy, and no one in the sub-bracket is stronger, Toddy can win r matches by swapping himself with any of those players in the sub-bracket.
4. If the match does not include Toddy, and one player in the sub-bracket is stronger, Toddy can win r matches by swapping himself with that one stronger player.

Now all we have to do is iterate over all the matches, keeping track of the number of stronger players below each sub-bracket. Since there are $O(N)$ matches, the overall run time of this solution is $O(N)$.

Python Example

```
from collections import defaultdict

n, k = map(int, input().split())
skills = list(map(int, input().split()))
ways = defaultdict(int)
stronger = [x > skills[0] for x in skills]
not_stronger = stronger.count(False)
initial_wins = 0

for r in range(1, k+1):
    for m in range(n >> r):
        stronger[m] = stronger[m*2] + stronger[m*2+1]
        if m == 0:  # match includes Toddy (before swapping)
            if stronger[m] == 0: initial_wins += 1
            if stronger[m] == 1: ways[r] += not_stronger - (1 << r) + 1
        else:
            if stronger[m] == 0: ways[r] += (1 << r)
            if stronger[m] == 1: ways[r] += 1

max_wins = initial_wins
for wins, count in ways.items():
    if count > 0: max_wins = max(max_wins, wins)
if initial_wins == max_wins:
    ways[max_wins] = -1
```

```
print(max_wins)
print(ways[max_wins])
```

C++ Example

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

int n,k;
int skills[1 << 16];
int stronger[1 << 16];
int ways[17];

int main() {
    cin >> n >> k;
    int notStronger = 0;
    int initialWins = 0;
    for(int i = 0; i < n; i++) {
        cin >> skills[i];
        stronger[i] = skills[i] > skills[0];
        notStronger += !stronger[i];
    }

    for(int r = 1; r <= k; r++) {
        for(int m = 0; m < (n >> r); m++) {
            stronger[m] = stronger[m*2] + stronger[m*2+1];
            if(m == 0 && stronger[m] == 0) initialWins++;
            if(m == 0 && stronger[m] == 1) ways[r] += notStronger - (1
             ↪  << r) + 1;
            if(m != 0 && stronger[m] == 0) ways[r] += (1 << r);
            if(m != 0 && stronger[m] == 1) ways[r]++;
        }
    }

    int maxWins = initialWins;
    for(int wins = 1; wins <= k; wins++) {
        if(ways[wins]) maxWins = max(maxWins, wins);
    }
    if(initialWins == maxWins) ways[maxWins] = -1;
    cout << maxWins << endl << ways[maxWins] << endl;
}
```

Big O complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can do about 10 million to 100 million useful operations per second. For example, if the problem specifies a run time of 1 second and an input N as big as 100 thousand, then you automatically know an $O(N^2)$ algorithm will not work since one hundred thousand squared is 10 billion operations.

Time complexity

The time taken by a program is usually estimated on the number of processor operations. For example, an addition $a+b$ or a compare $a < b$ is one operation.

$O(1)$ time means that the number of operations it performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. This will take the same amount of computation regardless of N. You can't get much better efficiency than that.

$O(\log N)$ time suggests the program takes a couple of extra operations every time N doubles in size.[3] For example, finding a number in a sorted list using binary

[3]More formally, it means there exists some constant c for which the program takes at most c extra operations every time N doubles in size.

search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb, in contests, $O(N^2)$ algorithms are only useful for input sizes of up to $N \leq 10\,000$.