# Contest 1 Editorial

COMP4128 23T3

24th September 2023

## A. Forgery

### Algorithm

We use brute force.

Read the signature and artwork. We can then run two nested loops to consider each possible position of the signature in the artwork, and another two nested loops to check whether every pixel matches.

This algorithm runs in $O(r_1 c_1 r_2 c_2)$ time, which is sufficient for $r_1, c_1, r_2, c_2 \leq 100$.

This problem was inspired by Mosaic Browsing from the 45th ICPC World Finals. The naïve algorithm above is too slow for this problem. The intended solution involves interpreting the problem as 2D string matching, which can be solved using a creative application of the fast Fourier transform algorithm. You may be interested in the video solution, and these notes may be a useful supplemental resource.

### Implementation Notes

#### General

- There are several equivalent ways to store the input grids: an array of strings, a 2D array of characters, a 2D array of booleans or integers, and perhaps more.
- Avoid using magic numbers for the dimensions.
- To test a possible signature position, either call a testing function or maintain a flag.
  - Both approaches permit early exit, but the worst case running time is unchanged (e.g. if both the signature and artwork have only one black pixel in the bottom right).
  - Likewise, heuristics such as the number of black pixels in a subrectangle (calculated using 2D prefix sum) will improve the average case but not the worst case.
  - Complexity analysis shows that the algorithm is fast enough without any of these optimisations, so don't write unnecessary code (that could introduce bugs).
  - If you are using a function, be careful not to pass vectors by value, as this creates a copy of the vector. Use pass by reference instead, or explicitly pass only indices to a global array or vector.
- Not every square of the artwork is a potential top-left corner of the signature! Check some small cases with pen and paper to calculate the exact bounds.
- This problem could be done with either 0-based or 1-based indexing. Pick one and stay consistent.

### Reference Solution

```cpp
// Solution by Lewis

#include <iostream>
using namespace std;

const int N = 101;
char sig[N][N];
```

```
char art[N][N];

int main() {
    int r1, c1, r2, c2;
    cin >> r1 >> c1 >> r2 >> c2;

    for (int i = 0; i < r1; ++i)
        for (int j = 0; j < c1; ++j)
            cin >> sig[i][j];

    for (int i = 0; i < r2; ++i)
        for (int j = 0; j < c2; ++j)
            cin >> art[i][j];

    for (int i = 0; i <= r2 - r1; ++i)
        for (int j = 0; j <= c2 - c1; ++j) {
            bool is_same = true;
            for (int k = 0; k < r1; ++k)
                for (int l = 0; l < c1; ++l)
                    if (art[i + k][j + l] != sig[k][l])
                        is_same = false;

            if (is_same) {
                cout << "Genuine\n";
                return 0;
            }
        }

    cout << "Forgery\n";
}
```

# B. Connect

## Algorithm

### Method I: simulation

After every move, update the board and check whether the new token completes a line of $k$ tokens in any direction.

This runs in $O(mk) = O(n^3)$ time. This is fast enough for $n \leq 300$, despite a constant factor of at least 4.

### Method II: binary search

We can use discrete binary search on the move number. Place all the tokens, with an associated move number. At any stage, considering only tokens placed until that point:

- if neither player has a win, recurse high
- if either player has a win, update the answer and recurse low.

The final answer is the first move where a win was obtained, and the winner can be deduced from its parity.

This runs in $O(n^2 \log m)$ with a similar constant factor, so it may be even faster. However, it does not benefit from as many early exits as the other approach.

## Implementation Notes

### General

- The biggest challenge was to iterate over all eight compass directions. You may not have known the trick to do this nicely (as shown in the reference solutions below). However, it is informed by the following principles that you should already be familiar with:
  - Try to reuse methods where possible.
  - *The less code you write, the less bugs you will make.*
  - Be particularly wary of copying and pasting entire code snippets. When you later find a bug in one copy, will you remember to edit the other copy accordingly?
- Since tokens accumulate upwards, $(x, y)$ coordinates (numbering from bottom left) are more suitable than $(i, j)$ coordinates (numbering from top left).
- The grid should not be stored as a boolean array since there are three possible values in each cell: no token, red and blue.
- You could find each token placement either by iterating through blank squares from the top or occupied squares from the bottom, or you could maintain the column heights. While the latter takes constant time per move, we are already spending linear time checking a win on each move so there is no harm in spending linear time again here.
- It is tempting to specify explicit bounds on the loop counters when checking each direction. However it's asymptotically no slower (and much simpler) to just check all bounds at each time.
- When you have ample time, write a function to help you debug!

```
void print_board (void) {
    for (int y = n-1; y >= 0; y--) {
        for (int x = 0; x < n; x++)
            if (grid[x][y] == RED)
                cerr << 'A';
            else if (grid[x][y] == BLUE)
                cerr << 'B';
            else
                cerr << ' ';
        cerr << '\n';
    }
}
```

You can even print to the terminal in colour: https://dev.to/tenry/terminal-colors-in-c-c-3dgc.

## Reference Solution

```cpp
// Solution by Evan, using simulation

#include <iostream>
using namespace std;

// dir = 0: offset ( 1, 0) -> east-west
// dir = 1: offset ( 1, 1) -> northeast-southwest
// dir = 2: offset ( 0, 1) -> north-south
// dir = 3: offset (-1, 1) -> northwest-southeast
int dx[4] = {1, 1, 0, -1};
int dy[4] = {0, 1, 1, 1};

const int N = 303;
int grid[N][N];
const int NONE = 0;
const int RED = 1;
const int BLUE = 2;

int n, m, k;

// inline functions get substituted in place without creating a stack frame
// more here: https://en.cppreference.com/w/cpp/language/inline
inline bool onboard (int x, int y) {
    return x >= 0 && x < n && y >= 0 && y < n;
}

inline int colour (int move) {
    if (move > m)
        return NONE;
    else if (move % 2)
        return RED;
    else
        return BLUE;
}

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

    int move;
    for (move = 1; move <= m; move++) {
        int clr = colour(move);

        int col;
        cin >> col;
        col--; // change to 0-based indexing

        // scan up current column until unoccupied square found
        // can be optimised by maintaining column heights
        int top;
        for (top = 0; top < n; top++)
            if (grid[col][top] == 0)
                break;

        grid[col][top] = clr;

        // check for a line in each of the four directions
        bool succ = false;
        for (int dir = 0; dir < 4; dir++) {
            int cnt = 1;

            // loop in +ve offset direction
            int x = col, y = top;
            while (true) {
                x += dx[dir];
                y += dy[dir];
                // stop when off board or different colour found
                if (!onboard(x,y) || grid[x][y] != clr)
                    break;
                cnt++;
```

```
            }

            // loop in -ve offset direction
            x = col, y = top;
            while (true) {
                x -= dx[dir];
                y -= dy[dir];
                // stop when off board or different colour found
                if (!onboard(x,y) || grid[x][y] != clr)
                    break;
                cnt++;
            }

            // early exit if line of k found
            if (cnt >= k) {
                succ = true;
                break;
            }
        }

        // early exit if win
        if (succ)
            break;
    }

    if (colour(move) == RED)
        cout << "Ayumi " << move << '\n';
    else if (colour(move) == BLUE)
        cout << "Bunji " << move << '\n';
    else
        cout << "No winner\n";
}
```

# C. Reading Numbers

## Algorithm

Store the starting number as a string, or equivalently as a vector of digits, and apply the procedure described in the problem to generate each subsequent number.

The number of digits is initially at most ten, and can at most double at every step but in practice grows at a slower (but still exponential) rate as runs of more than one digit occur. The time complexity is difficult to precisely estimate, but the actual running time is easily confirmed to be well within the limit.

The sequence in question is a generalisation of the look-and-say sequence, although knowing this wouldn't particularly help solve the problem.

## Implementation Notes

### General

- After the starting number, you cannot have a run of length greater than three identical digits (prove it). It follows that all runs are of single-digit length, so there is no need to convert the run length into a sequence of digits.
- Numbers towards the end of the sequence are very large.
  - They certainly won't fit in a 32-bit or even 64-bit integer.
  - The most challenging cases are those where the starting number has nine different digits.
  - Be careful not to copy the numbers around more than necessary.

## Reference Solution

```
// Solution by Nam

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

int main (void) {
    string s;
    int k;
    cin >> s >> k;

    vector<int> v;
    for (int i = 0; i < s.length(); ++i)
        v.push_back(s[i] - '0');

    for (int i = 2; i <= k; ++i) {
        vector<int> w;
        int len = 1;
        for (int j = 0; j < v.size(); ++j)
            if (j == v.size() - 1 || v[j] != v[j + 1]) { // end of a run of identical
                digits
                w.push_back(len);
                w.push_back(v[j]);
                len = 1;
            } else // extend run
                len++;
        v = w;
    }

    // print answer, digit-by-digit
    for (int i = 0; i < v.size(); ++i)
        cout << v[i];
    cout << '\n';
}
```

# D. Clock Gallery

## Algorithm

After converting the clock times to seconds, it remains to check whether the set of clock times matches the set of time differences, up to some offset. We can't try all $n!$ assignments, so we rely on the fact that any offset preserves (cyclic) ordering.

### Method I: comparison sort

Sort all the clock times and all the time differences. Try assigning each of the time differences to the first clock, and check whether the remaining clocks and time differences are consistent with this.

The sorting takes $O(n \log n)$, followed by $n$ rounds each requiring an $O(n)$ loop. Therefore the time complexity is $O(n^2)$, which is fast enough for $n \leq 1000$.

### Method II: red-black tree

As in Method I, but using a `set` to store the clock times and another for the time differences.

The insertions take $O(n \log n)$, followed by $n$ rounds each requiring $O(n)$ `set` operations. Therefore the time complexity is $O(n^2 \log n)$, which is fast enough for $n \leq 1000$.

### Method III: counting sort (ish)

Make boolean arrays to denote whether each possible clock time and each possible time difference appears. Try assigning each of the time differences to the first clock, work out the necessary offset and check whether the arrays match up to this offset.

There are $n$ rounds each requiring a loop through 86400 time steps. This is fast enough for $n \leq 1000$.

### Method IV: who needs ordering?

For every pair of a clock and a time difference, record the offset between them. If there are $n$ pairs with the same offset, this must be the solution as there are no duplicates.

This runs in $O(n^2)$, which is fast enough for $n \leq 1000$.

## Implementation Notes

### General

- `scanf()` may be a convenient way to read formatted input.
- C++ does not observe the mathematical definition of the modulo operator for negative numbers, which makes comparison modulo $m$ complicated. For a number $-m \leq x < 0$, we can retrieve it by computing `(x+m)%m`. See the tips page for how to handle even more negative numbers.
- To deduce the labelling, you have to either keep track of the original indices when sorting or do one final pass with the correct offset known.
- Some students misunderstood the output format, so an additional sample case was added for clarification.

### Method I

- Consistency is transitive; if we have three cities assigned to three clocks, and two pairs are consistent, then the third pair is also consistent. This implies that we only need to check consistency between consecutive clocks.
- You can write a wrap-around loop either by splitting into two loops or using the modulo operator.

### Method II

- If there were duplicate clock times and time differences, we would have to check that the frequencies match.

### Method III

- If there were duplicate clock times and time differences, we would have to use a `multiset`.

## Reference Solutions

```cpp
// Solution by Raveen, by comparison sorting

#include <algorithm>
#include <cstdio>
#include <utility>
using namespace std;

const int N = 1010;
pair<int,int> clk[N]; // (clock time, original index)
pair<int,int> dif[N]; // (time difference, original index)
int lbl[N]; // labels, for output

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

    for (int i = 0; i < n; i++) { // convert clock times to seconds
        int h, m, s;
        scanf("%d:%d:%d",&h,&m,&s);
        clk[i] = make_pair(h*3600+m*60+s,i);
    }
    sort(clk,clk+n);

    for (int j = 0; j < n; j++) { // read time differences
        int t;
        scanf("%d",&t);
        dif[j] = make_pair(t,j);
    }
    sort(dif,dif+n);

    for (int i = 0; i < n; i++) {
        // try assigning clock i (in sorted order) to time difference 0

        bool ok = true;
        // (sorted) clock j will then get (sorted) time difference j-i (mod n)
        // check consistency for each j
        for (int j = i+1; j < n; j++) // loop forward
            if (clk[j].first-clk[i].first       != dif[j  -i].first-dif[0].first)
                ok = false;
        for (int j = 0; j < i; j++) // wrap around
            // add 86400 (# seconds in a day) to fix earlier clocks
            if (clk[j].first-clk[i].first+86400 != dif[j+n-i].first-dif[0].first)
                ok = false;

        if (ok) { // all consistent
            for (int j = 0; j < n; j++) // recover original indices (i.e. unsort)
                lbl[clk[j].second] = dif[(j-i+n)%n].second;
            for (int j = 0; j < n; j++)
                printf("%d%c", lbl[j]+1, (j == n-1 ? '\n' : ' '));
            return 0;
        }
    }

    printf("Impossible\n");
}
```

```cpp
// Solution by Ashley Clayton (student), using offset count
// small edits for brevity

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

```
const int DAY = 24 * 60 * 60;
const int N = 1010;
int clocks[N];
int diffs[N];
int paris[DAY]; // count of clock-diff pairs with this offset

int n;

void print_sol(int time) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            int index = (clocks[i] - diffs[j] + DAY) % DAY;
            if (index == time)
                cout << j + 1 << (i == n-1 ? '\n' : ' ');
        }
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        int h, m, s;
        char c; // colons
        cin >> h >> c >> m >> c >> s;
        int time = h*60*60 + m*60 + s;
        clocks[i] = time;
    }

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

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // compute offset, making sure not to get a negative answer
            int offset = (clocks[i] - diffs[j] + DAY) % DAY;
            // increment count of this offset
            paris[offset]++;
            // if count is n, we are done
            if (paris[offset] == n) {
                print_sol(offset);
                return 0;
            }
        }
    }

    cout << "Impossible" << '\n';
}
```

# E. Coin Puzzle

## Algorithm

This problem is exactly 3SAT! Famously, 3SAT is NP-complete, and we cannot do much better than brute force.

There are $n \leq 22$ variables, and for each of the $2^n$ assignments of values to those variables, we could need to check each literal of $m \leq 100$ clauses. Therefore the time complexity is $O(m2^n) \approx 4 \times 10^8$, with a constant factor of at least three. This explains the larger time limit for this problem.

Exercise: design a worst case input.

## Implementation Notes

### General

- The main implementation decision is how to iterate over all values of $n$ boolean variables. There are three options, which were all acceptable.
  - Similar to the Queens problem from Lecture 2, we can proceed by recursion. Assign the first variable as true, then recurse, then assign it as false, then recurse again.
  - We can use a bitset. Iterate an integer from 0 to $2^n - 1$, and interpret its binary representation as denoting which of the variables are true and which are false.
  - Finally we can just run many nested loops. It's not the most elegant, but it works! Consider writing a program to print out the source code for you.

## Reference Solutions

```
// Solution by Raveen, using recursion

#include <iostream>
#include <cstdlib> // for exit()
using namespace std;

const int N = 24;
bool a[N]; // which way each coin faces in the current combination: H is true, T is
    false

const int M = 110;
bool f[M][3]; // which face
int c[M][3]; // which coin

int n, m;

void check (void) {
  for (int i = 0; i < m; i++) { // for each clause
    bool ok = false;
    for (int j = 0; j < 3; j++) // for each literal of that clause
      if (f[i][j] ? a[c[i][j]] : !a[c[i][j]]) // look for matching face of respective
          coin
        ok = true;
    if (!ok) // nothing in this clause satisfied
      return;
  }

  for (int i = 1; i <= n; i++)
    cout << (a[i] ? 'H' : 'T') << ((i==n) ? '\n' : ' ');
  exit(0);
}

void go (int i = 1) {
  if (i == n+1) // assigned all
    check();
  else { // try recursing with each face of current coin
    a[i] = true;
    go(i+1);
    a[i] = false;
```

```
    go(i+1);
  }
}

int main (void) {
  cin >> n >> m;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < 3; j++) {
      int r;
      cin >> r;
      f[i][j] = r>0;
      c[i][j] = abs(r);
    }

  go();

  cout << "Doomed\n"; // only get here if no solution found
}
```

```
// Solution by Raveen, using a bitset

// replaces go() in recursive solution
void go (void) {
  for (int x = 0; x < (1<<n); x++) { // try all configurations
    for (int i = 0; i < n; i++) // least significant bit represents coin 1, etc
      a[i+1] = x & (1<<i); // true if bit i is 1, false if 0
      check();
  }
}
```

```
// Solution by Raveen, using loops

// a[] must now be an array of int to support ++ operation

// replaces go() in recursive solution
void go (void) {
  // may as well go over all 22 possible variables, even if there are fewer
  for (a[0] = 0;  a[0] < 2;  a[0]++)
    for (a[1] = 0;  a[1] < 2;  a[1]++)
      for (a[2] = 0;  a[2] < 2;  a[2]++)
        for (a[3] = 0;  a[3] < 2;  a[3]++)
          for (a[4] = 0;  a[4] < 2;  a[4]++)
            for (a[5] = 0;  a[5] < 2;  a[5]++)
              for (a[6] = 0;  a[6] < 2;  a[6]++)
                for (a[7] = 0;  a[7] < 2;  a[7]++)
                  for (a[8] = 0;  a[8] < 2;  a[8]++)
                    for (a[9] = 0;  a[9] < 2;  a[9]++)
                      for (a[10] = 0; a[10] < 2; a[10]++)
                        for (a[11] = 0; a[11] < 2; a[11]++)
                          for (a[12] = 0; a[12] < 2; a[12]++)
                            for (a[13] = 0; a[13] < 2; a[13]++)
                              for (a[14] = 0; a[14] < 2; a[14]++)
                                for (a[15] = 0; a[15] < 2; a[15]++)
                                  for (a[16] = 0; a[16] < 2; a[16]++)
                                    for (a[17] = 0; a[17] < 2; a[17]++)
                                      for (a[18] = 0; a[18] < 2; a[18]++)
                                        for (a[19] = 0; a[19] < 2; a[19]++)
                                          for (a[20] = 0; a[20] < 2; a[20]++)
                                            for (a[21] = 0; a[21] < 2; a[21]++)
                                              check();
}
```