Contest 1 Editorial

COMP4128 22T3

19th September 2022

A. Compression

Algorithm

We use brute force.

Read the input into an array of strings. We can then run two nested loops to iterate through the 3×3 squares, counting the white and/or black pixels in each and setting the corresponding pixel in the final image.

This algorithm runs in $O(mn)$ time, which is sufficient for $m, n \leq 1,000$.

Implementation Notes

General

- Where possible, use multiplication instead of division, so that you don't have to think about rounding.
- Test your code locally on the sample inputs before you submit.
- Make sure you reserve enough space to store the original board; it has dimensions $3m \times 3n$, not $m \times n$.
- You could either print out each character of the image as you go, or store the answer in an array for printing at the end.

Reference Solutions

```
// Solution by Raveen, printing immediately
#include <iostream>
using namespace std;

const int N = 1010;
char a[3*N][3*N];

int main (void) {
  int m, n;
  cin >> m >> n;
  for (int i = 0; i < 3*m; i++)
    cin >> a[i];

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      // square consisting of:
      // rows 3i, 3i+1 and 3i+2
      // cols 3j, 3j+1 and 3j+2

      int cnt = 0;
      for (int k = 0; k < 3; k++)
        for (int l = 0; l < 3; l++)
          if (a[3*i+k][3*j+l] == '*')
            cnt++;
      cout << (cnt >= 5 ? '*' : '.');
    }
    cout << '\n';
  }
}
```

```
// Solution by Kevin, printing at the end
#include <cstdio>
#include <iostream>
using namespace std;

char a[3005][3005]; // uncompressed
char b[1005][1005]; // compressed

int main () {
  int M, N;
  cin >> M >> N;

  for (int i = 0; i < 3*M; i++)
    for (int j = 0; j < 3*N; j++)
      cin >> a[i][j];

  for (int i = 0; i < 3*M; i += 3)
    for (int j = 0; j < 3*N; j += 3) {
      // square consisting of:
      // rows i, i+1 and i+2
      // cols j, j+1 and j+2

      int nstars = 0;
      for (int y = 0; y < 3; y++)
        for (int x = 0; x < 3; x++)
          if (a[i+y][j+x] == '*')
            nstars++;

      if (nstars >= 5)
        b[i/3][j/3] = '*';
      else
        b[i/3][j/3] = '.';
    }

  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++)
      printf("%c", b[i][j]);
    printf("\n");
  }
}
```

B. Rocks

This was the only problem to be solved by all students.

Algorithm

An algorithm which considers the energy spent on each unit of distance will run in time proportional to m, which is too slow. On the other hand, since n is up to $1,000,000$, an $O(n \log n)$ algorithm will run in time.

Method I: simulation

The rocks must be collected in increasing order of position, so we begin by sorting the x_i. Then, for each pair of consecutive rocks, calculate the amount of energy expended to travel between them. Finally, we also have to include the distance from the last rock to the ending point m.

This runs in $O(n \log n)$ time, dominated by sorting.

Method II: regrouping

We can regroup the cost; instead of counting each unit of distance separately, we can count each rock separately. Each rock contributes 1 energy cost per unit of distance that it is carried. Each rock must be carried from its starting point x_i to the ending point m, so it contributes $m - x_i$ to the total.

This runs in $O(n)$ time, as each rock can be processed in constant time without sorting.

Implementation Notes

General

- Students had to account for the presence of two or more rocks at a single position. Sample case 1 was designed to confirm that students were aware of this possibility.
- The answer does not fit in an `int` variable, so students had to use `long long` to avoid overflow. Sample case 2 was designed to help students catch this bug, but in future contests students should not rely on sample cases for debugging.

Method I

- It is useful to add a dummy rock at position m, to avoid implementing a special case for the last segment of the walk. This is one reason to declare arrays larger than necessary.
- The C++ Standard Template Library provides a `sort` function in the `<algorithm>` header, which can be used with the default comparator `<` or a custom comparator.
  - No particular sorting algorithm is required by the language standard, but the function is guaranteed to have worst case time complexity of $O(n \log n)$.
  - `sort` is not stable; if stable sorting is required, use the `stable_sort` function (also in `<algorithm>`).

Method I, alternative implementation

- Some students attempted to use STL's `multiset` or `map` data structures to store the rock positions, instead of sorting.
- This also achieves an $O(n \log n)$ time complexity. However, the overhead from these data structures is significant, meaning that the constant factor in these submissions was greater than in those based on sorting.
- In this problem, n was up to $1,000,000$, so $n \log n \approx 2 \times 10^7$. As such, only a fairly 'tight' $O(n \log n)$ solution (i.e. one with a small constant factor) would pass. Consequently, most of these submissions barely exceeded the time limit.
- If n were only up to $100,000$, this approach would have passed.

Reference Solutions

```
// Solution by Angus, by simulation
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

typedef long long ll;
ll ans, sum;

const int MAXN = 1001001;
int a[MAXN];

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

  a[n] = m;
  sort(a, a+n+2);

  for (int i = 1; i <= n+1; i++) {
    ans += sum*(a[i]-a[i-1]);
    sum++;
  }
  cout << ans << "\n";
}
```

```
// Solution by Isaiah, by regrouping
#include <cstdio>
using namespace std;

typedef long long ll;
int main() {
  ll n, m;
  ll ans = 0;
  scanf("%lld%lld", &n, &m);

  ll x;
  for (ll i = 0; i < n; i++) {
    scanf("%lld", &x);
    ans += m - x;
  }
  printf("%lld\n", ans);
}
```

C. Sister Cities

Algorithm

There are up to $100,000$ cities, so we cannot examine the temperature difference between each pair of cities; this would run in $O(n^2)$ time and hence exceed the time limit.

There were at least three viable approaches.

Method I: sorting

Sort the cities by temperature, then consider only pairs that are adjacent in the sorted list.

The time complexity is $O(n \log n)$, as the second step takes only linear time. The overall complexity is $O(n \log n)$.

Method II: explicit separate chaining

For each temperature, record all cities with that temperature. If there are two or more cities with equal temperature, output them. Otherwise, iterate through the temperatures in increasing order (skipping any which do not represent any city) and consider only those pairs of cities whose temperatures are adjacent in this list.

The time complexity is $O(k)$, where $k = 10^6 + 1$ is the number of possible temperatures.

Implementation Notes

General

- When iterating through a list to find the maximum or minimum, make sure to update the 'best so far' variable every time its value is beaten.
- Read the output format carefully. You were not required to output the temperatures; just the city names.
- To sort while maintaining the association between temperatures and city names, make a `pair<double,string>` or similar for each city.
- When sorting pairs, the default comparison is by first entry, only comparing the second entry if it is necessary to break a tie.
- Remember that comparing strings takes time linear in the length of the string in the worst case, which contributes a constant factor of 10 to the running time of the algorithm.

Method I, alternative implementation

- Instead of sorting, one could Insert the cities into a self-balancing binary search tree or hash table, using the temperature as the key, then consider only pairs that are adjacent in the inorder traversal.
- The time complexity is $O(n \log n)$ with an SBBST or expected $O(n)$ with a hash table, as the second step takes only linear time.

Data types for temperatures

- Each temperature could be stored either as an `int` (exact temperature $\times 10^4$) or as a `double` (double-precision floating point number, precise enough).
- However, storing it as a `pair<int,int>` (where the first part denotes the value before the decimal point, and the second part the value after the decimal point) is difficult due to the presence of negative values; in particular, one must distinguish between 0.1234 and -0.1234.
- Finally, some students used a `float` (single-precision floating point number) for each temperature. This is not precise enough, and resulted in wrong answers due to inaccuracies in the subtraction of temperatures.

Reference Solutions

```
// Solution by Raveen, by sorting, storing temperatures as doubles
#include<algorithm>
#include<iostream>
#include<utility>
using namespace std;

const int N = 100100;
pair<double,string> cities[N];

int main (void) {
  int n;
  cin >> n;

  string name;
  double temp;
  for (int i = 0; i < n; i++) {
    cin >> name >> temp;
    cities[i] = make_pair(temp,name);
  }

  // sort by increasing temperature, breaking ties by city name lexicographically
  sort(cities,cities+n);

  // compare consecutive cities in this sorted order
  double bestdiff = 200;
  int ansindex = -1;
  for (int i = 0; i < n-1; i++)
    if (cities[i+1].first - cities[i].first < bestdiff){
      bestdiff = cities[i+1].first - cities[i].first;
      ansindex = i;
    }

  cout << cities[ansindex].second << ' ' << cities[ansindex+1].second << '\n';
}
```

```
// Solution by Raveen, by sorting, converting temperatures to integers
#include<algorithm>
#include<iostream>
#include<string>
#include<utility>
using namespace std;

const int N = 100100;
pair<int,string> cities[N];

int main (void) {
  int n;
  cin >> n;

  string name;
  string temp;
  for (int i = 0; i < n; i++) {
    cin >> name >> temp;
    // store temperature as a string, delete the decimal point and convert to int
    int dcpl = temp.find('.');
    temp.erase(temp.begin()+dcpl);
    cities[i] = make_pair(stoi(temp),name);
  }

  sort(cities,cities+n);

  // compare consecutive cities in this sorted order
  int bestdiff = 2000000;
  int ansindex = -1;
  for (int i = 0; i < n-1; i++)
    if (cities[i+1].first - cities[i].first < bestdiff) {
      bestdiff = cities[i+1].first - cities[i].first;
      ansindex = i;
    }

  cout << cities[ansindex].second << ' ' << cities[ansindex+1].second << '\n';
}
```

```cpp
// Solution by Zhizhou, using explicit separate chaining
#include<iostream>
#include<utility>
#include<vector>
using namespace std;

const int M=1000100;
const double EPS=1e-6;
vector<string> v[M];
// any list could have all 1,000,000 cities, so we can't reserve enough space for all of
     them
// instead use vectors (i.e. scalable arrays)

int main(void) {
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    double x;
    string s;
    cin >> s >> x;

    // convert to array index: -50.0000 -> 0, -49.9999 -> 1, etc
    // add small epsilon, because implicit cast to int involves rounding down
    int idx = (x+50) * 10000 + EPS;
    v[idx].push_back(s);
  }

  // previous array index where the vector was nonempty
  int pre = -1;
  // array indices of answer cities
  int l=0, r=M-1;
  for (int i = 0; i < M; i++) {
    // nothing to do if there are no cities of the temperature corresponding to this
        index
    if (v[i].empty())
      continue;

    // if there are two cities with the same temperature, we're done
    if (v[i].size() >= 2) {
      cout << v[i][0] << ' ' << v[i][1] << '\n';
      return 0;
    }

    // otherwise, exactly one city of this temperature, so update answer (if necessary)
        and previous
    if (pre != -1 && i-pre < r-l){
      l=pre;
      r=i;
    }
    pre=i;
  }

  // answer output here, when the temperature difference is nonzero
  cout << v[l][0] << ' ' << v[r][0] << '\n';
}
```

D. Poker

A good example of a task which is very easy to do by eye, but difficult to program correctly! This was arguably the most difficult problem, with only about one in six submissions accepted.

Algorithm

We use brute force.

Iterate through all selections of five cards from the given seven. For each five-card hand, build a table of suit frequencies and a table of rank frequencies, and check each hand type from best to worst.

The algorithm runs in $O(\binom{n}{k}(k+m))$ time, where $n = 7$ is the number of available cards, $k = 5$ is the number of selected cards and $m = 13$ is the number of ranks. $\binom{7}{5}$ is only 21, so this is easily sufficient.

This complexity analysis should inform the approach you take in implementation. We do not have to worry about efficiency at all, so we should make choices that:

- directly and explicitly encode the rules from the statement
- can be easily verified by reading the code
- can be easily debugged

Indeed, just about any algorithm would run in time here; there is even enough time to iterate through all possible hands from a 52-card deck.

Implementation Notes

General

- As this problem is a test of implementation and debugging skills, much of the guidance from earlier courses such as COMP1511 is in fact relevant here.
- Good style makes it easier to read and debug your code.
- Use `#define` directives rather than magic numbers.
- Before you start coding, make an implementation plan on pen and paper, in comments or in a separate document.
- *The less code you write, the less bugs you will make.*
- Try to reuse methods where possible.
- 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?

Iterating over hands

- Some students chose to classify directly from the seven available cards, rather than considering all five-card hands.
  - This is viable, but additional checks are required.
  - With seven cards, there can be six or seven cards of a single suit, three pairs, two triples, and so on.
  - This often led to incorrect classification of "full house", "three of a kind", "two pair" and "one pair" cases, as well as some "straight flush" cases as detailed below.
- Since there is ample time, we recommend testing each five-card hand separately instead. This could be done by:
  - running five nested loops (corresponding to the selected cards)
  - running two nested loops (corresponding to the unselected cards) or

  – using `next_permutation()` to iterate from $[0,0,1,1,1,1,1]$ to $[1,1,1,1,1,0,0]$.

Straight Flush

- A particularly tricky case was `5H 8H 3H 6S 7C 9H AH`, which has both a straight and a flush, but not in the same five-card hand, so there is no straight flush. Many students who tested the entire collection of seven cards for a straight flush by checking `flush && straight` failed this case.
- Another common breaking case was `TH QH JH 9C 8C KH AH`, which has straights `89TJQ` and `9TJQK` but also a straight flush `TJQKA`.

Straight

- Care had to be taken to find straights of both the forms `A2345` and `TJQKA`.
- Some submissions incorrectly classified `2C 3H 5S 7D 9D 4C AH` and/or `TH QH JD 9C 8C KS AC` as "high card".
- One approach was to maintain 14 ranks, marking both rank 1 and rank 14 present when an ace was seen.

Full House

- Some students had difficulty testing for "full house" cases, and distinguishing them from "two pair" and "three of a kind".
- Once a five-card hand has tested negative for "four of a kind", we can simply check whether exactly two ranks are present in the hand: if so, they must be a triple and a pair.

Reference Solutions

```
// Solution by Raveen, testing each five-card hand separately
#include<algorithm>
#include<iostream>
using namespace std;

#define STRAIGHT_FLUSH 0
#define FOUR_OF_A_KIND 1
#define FULL_HOUSE 2
#define FLUSH 3
#define STRAIGHT 4
#define THREE_OF_A_KIND 5
#define TWO_PAIR 6
#define ONE_PAIR 7
#define HIGH_CARD 8

int rnk[7];
char suit[7];

int classify (int x[7]) {
  // local variables are garbage-initialised, not zero-initialised
  int freq[13];
  fill(freq,freq+13,0);

  // count ranks among selected cards
  for (int i = 0; i < 7; i++)
    if (x[i])
      freq[rnk[i]]++;

  bool straight = false;
  for (int j = -1; j <= 8; j++) {
    // (j+13)%13 = 12   if j is -1
    //           = j    otherwise,
    // so this catches A2345
    if (freq[(j+13)%13] && freq[j+1] && freq[j+2] && freq[j+3] && freq[j+4]) {
      straight = true;
      break;
    }
  }
  bool flush = true;
  // find first selected card
  for (int i = 0; i < 7; i++)
    if (x[i]) {
      // compare suit against all other selected cards
      for (int k = i+1; k < 7; k++)
        if (x[k] && suit[k] != suit[i])
          flush = false;
      break;
    }

  // check straight flush
  if (straight && flush)
    return STRAIGHT_FLUSH;

  // check quads
  if (count(freq,freq+13,4))
    return FOUR_OF_A_KIND;

  // check full house
  if(count(freq,freq+13,0) == 11)
    return FULL_HOUSE;

  // check flush
  if (flush)
    return FLUSH;

  // check straight
  if (straight)
    return STRAIGHT;

  // check trips
  if (count(freq,freq+13,3))
    return THREE_OF_A_KIND;

  // count pairs
  int pairs = count(freq,freq+13,2);
  if (pairs == 2)
    return TWO_PAIR;
  else if (pairs == 1)
    return ONE_PAIR;
  else
    return HIGH_CARD;
}

int main (void) {
  string s;
  for (int i = 0; i < 7; i++) {
    cin >> s;
    // 2 - 9 as is
    if (isdigit(s[0]))
      rnk[i] = s[0]-'0';
    else if (s[0] == 'T')
      rnk[i] = 10;
    else if (s[0] == 'J')
      rnk[i] = 11;
    else if (s[0] == 'Q')
      rnk[i] = 12;
    else if (s[0] == 'K')
      rnk[i] = 13;
    else if (s[0] == 'A')
      rnk[i] = 14;
    // record 2 as rank 0, 3 as rank 1, ..., ace as rank 12
    rnk[i] -= 2;
    suit[i] = s[1];
  }

  // iterate through five-card hands
  int x[7] = {0,0,1,1,1,1,1};
  int ans = HIGH_CARD;
  do
    ans = min(ans,classify(x));
  while (next_permutation(x,x+7));

  string output[HIGH_CARD+1] = {"straight flush", "four of a kind", "full house", "flush
      ", "straight", "three of a kind", "two pair", "one pair", "high card"};
  cout << output[ans] << '\n';
}
```

```
// Solution by Laeeque, testing all seven cards together
#include<iostream>
#include<vector>
using namespace std;

// variable name 'rank' causes a naming conflict
int suit[7], _rank[7];
vector<int> ofsuit[4];
vector<int> ofrank[13];
// m[][] is a table of all possible cards
bool m[4][14];

// variable name 'flush' causes a naming conflict
bool straight_flush = false;
bool four_of_a_kind = false;
bool full_house = false;
bool _flush = false;
bool straight = false;
bool three_of_a_kind = false;
bool two_pair = false;
bool one_pair = false;
bool high_card = true;

int main() {
  for (int i = 0; i < 7; i++) {
    string s;
    cin >> s;
    switch (s[0]) {
      case 'A': _rank[i] = 1-1; break;
      case 'T': _rank[i] = 10-1; break;
      case 'J': _rank[i] = 11-1; break;
      case 'Q': _rank[i] = 12-1; break;
      case 'K': _rank[i] = 13-1; break;
      default: _rank[i] = s[0] - '0'-1; break;
    }
    switch (s[1]) {
      case 'D': suit[i] = 0; break;
      case 'C': suit[i] = 1; break;
      case 'H': suit[i] = 2; break;
      case 'S': suit[i] = 3; break;
    }
    ofsuit[suit[i]].push_back(i);
    ofrank[_rank[i]].push_back(i);
    m[suit[i]][_rank[i]] = true;
  }

  for (int i = 0; i < 7; i++) {
    int s = suit[i], r = _rank[i];

    // test for straight flush starting here
    // check next four ranks of this suit, wrapping around if necessary to catch TJQKA
    straight_flush = straight_flush
      || (r <= 10
          && m[s][(r+1) % 13]
          && m[s][(r+2) % 13]
          && m[s][(r+3) % 13]
          && m[s][(r+4) % 13]);

    // test for straight starting here
    // check next four ranks of this suit, wrapping around if necessary to catch TJQKA
    straight = straight
      || (r <= 10
          && ofrank[(r+1) % 13].size() >= 1
          && ofrank[(r+2) % 13].size() >= 1
          && ofrank[(r+3) % 13].size() >= 1
          && ofrank[(r+4) % 13].size() >= 1);
    // test for flush including this card
    _flush = _flush || ofsuit[s].size() >= 5;
  }

  // for each rank, check quads, trips and pairs
  for (int r = 0; r < 13; r++) {
    four_of_a_kind = four_of_a_kind || ofrank[r].size() >= 4;
    three_of_a_kind = three_of_a_kind || ofrank[r].size() >= 3;
    // if at least one pair was seen earlier, we now have two_pair or better
    two_pair = two_pair || (one_pair && ofrank[r].size() >= 2);
    one_pair = one_pair || ofrank[r].size() >= 2;
  }
  // two_pair includes the triple
  full_house = three_of_a_kind && two_pair;

  if (straight_flush) cout << "straight flush";
  else if (four_of_a_kind) cout << "four of a kind";
  else if (full_house) cout << "full house";
  else if (_flush) cout << "flush";
  else if (straight) cout << "straight";
  else if (three_of_a_kind) cout << "three of a kind";
  else if (two_pair) cout << "two pair";
  else if (one_pair) cout << "one pair";
  else cout << "high card";
}
```

E. Contest Strategy

Algorithm

We use the greedy method.

We have a total of km computer-minutes to allocate.

We should allocate as many minutes as possible to our best team member, the one whose p_i value is largest. Since all t_i are at most m, this person's entire stamina can be used.

Note that any allocation which does not use the entire stamina of this person can easily be transformed into an equal or better allocation that does.

We can then continue to the next team member, and so on until either all team members or all computer-minutes are exhausted.

Recall that no team member can use two or more computers at a time. The solution above does not address this constraint directly, so there might be some doubt as to its correctness. However, we can assign the best person to the first t_i minutes on computer 1, then the next best person to the next t_j minutes on computer 1, and so on. Once the total time m on computer 1 is exhausted, we then allocate minutes on computer 2 beginning at the start of the contest. Since all t_i are at most m, no person uses two computers at once.

Implementation Notes

Sorting

- The C++ STL `sort` uses the default comparator `<` unless otherwise specified. When comparing two `pairs` `a = (a.first, a.second)` and `b = (b.first, b.second)`, `a < b` is true if:
  - `a.first < b.first` or
  - `a.first = b.first` and `a.second < b.second`.
- So if we store the team members in an array of pairs, with first entry p_i and second entry t_i, the default sort will be ascending by points per minute while tracking the corresponding t_i values.
- However, we want to sort in decreasing order of p_i, so we can either provide the "greater than" comparison in place of the default, or just sort as normal and reverse the output.

Overflow

- Values up to 10^{18} can be stored in a `long long`.
- Note that the following snippet has an integer overflow bug.

```
int m, k;
cin >> m >> k;
long long computer_minutes = m*k;
```

  The expression 'm*k' is calculated as a 32-bit integer and only then stored in a 64-bit integer.
- There are several ways to fix this:
  - store `m` and `k` as `long long` also,
  - explicitly cast `m` and/or `k` to `long long` or
  - write `1LL*m*k` instead, to force the calculation to be done in 64-bit arithmetic.

Reference Solution

```
// Solution by Raveen
#include<algorithm>
#include<iostream>
#include<utility>
using namespace std;

```

```
typedef long long ll;

const int N = 200200;
pair<ll,ll> member[N];

int main (void) {
  ll n, m, k;
  cin >> n >> m >> k;
  ll mins = m*k;

  for (int i = 0; i < n; i++)
    cin >> member[i].first >> member[i].second;

  sort(member,member+n,greater<pair<ll,ll>>());
  // alternatively:
  // sort(member,member+n);
  // reverse(member,member+n);

  ll ans = 0;
  ll used = 0;
  for (int i = 0; i < n; i++) {
    ll curr = min(mins-used,member[i].second);
    ans += curr * member[i].first;
    used += curr;
  }

  cout << ans << '\n';
}
```