WOBURN CHALLENGE

2017-18 Online Round 1

Solutions

Automated grading is available for these problems at:
wcipeg.com

For problems to this contest and past contests, visit:
woburnchallenge.com

Problem J1: Canadian Accent

Simply output the string followed by "`, eh`" as instructed.

Official Solution (C++)

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

int main() {
  string s;
  cin >> s;
  cout << s + ", eh" << endl;
  return 0;
}
```

Problem J2: How's the Weather?

Rewriting the given formula to isolate F, we arrive at
$F = C \times 9 / 5 + 32$. We can then input the integer C, plug it into this formula, and output the resulting value of F. We don't need to worry about integer versus floating point division due to the guarantee that F will come out to an integer.

Official Solution (C++)

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

int main() {
  int c;
  cin >> c;
  cout << (c*9 / 5 + 32) << endl;
  return 0;
}
```

Problem J3/I1: Stanley

Let G_a be the number of games won by Team A, and G_b be the number of games won by Team B. We can initialize both of these values to 0 and then process the games in the series one by one. For each game i, we can input its A_i and B_i values, and then either increment G_a if $A_i > B_i$, or increment G_b otherwise (if $B_i > A_i$). Once either $G_a = 4$ or $G_b = 4$, we should stop processing games and output the current values of G_a and G_b.

Official Solution (C++)

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

int main() {
  int gA = 0, gB = 0;
  while (gA < 4 && gB < 4) {
    int pA, pB;
    cin >> pA >> pB;
    if (pA > pB) {
      gA++;
    } else {
      gB++;
    }
  }
  cout << gA << " " << gB << endl;
  return 0;
}
```

Problem J4/I2: Canuck Detection

The most direct solution is to consider all possible ordered triples of indices (i, j, k) which contain the required characters for the subsequence "`our`" – in other words, such that $i < j < k$, $S[i]$ = "`o`", $S[j]$ = "`u`", and $S[k]$ = "`r`". This can be accomplished with 3 nested loops, iterating over all possible combinations of these indices. If we find any triple of indices meeting these requirements, then they correspond to the subsequence "our" and we can output "`Y`".

Unfortunately, the above solution is too slow to get full marks. It requires considering almost N^3 combinations of indices in the worst case, which in formal computer science terms means that it has a "time complexity" of $O(N^3)$. It's possible to instead solve the problem by only considering each index at most once, with a time complexity of $O(N)$. If we find the first "`o`" in S, then it can't help us to use a later "`o`" instead. So, we should instead then find the first "`u`" after that first "`o`", and then the first "`r`" after that "`u`". If we find all 3 required letters in order like this, then we've found the subsequence "`our`". On the other hand, if we're unable to find the next required letter at any point, then the subsequence must not exist in S.

Official Solution (C++)

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

int main() {
  string s;
  cin >> s;
  // Find the first "o", if any.
  int i = 0;
  while (i < s.size() && s[i] != 'o') {
    i++;
  }
  if (i == s.size()) {
    cout << "N" << endl;
    return 0;
  }
  // Find the first "u" after that, if any.
  while (i < s.size() && s[i] != 'u') {
    i++;
  }
  if (i == s.size()) {
    cout << "N" << endl;
    return 0;
  }
  // Find the first "r" after that, if any.
  while (i < s.size() && s[i] != 'r') {
    i++;
  }
  if (i == s.size()) {
    cout << "N" << endl;
    return 0;
  }
  cout << "Y" << endl;
  return 0;
}
```

Problem I3/S1: On the Rocks

There are several approaches to this problem, such as first determining which team will score points and then counting how many points they should score, or sorting all $N + M$ stones in increasing order of distance from the button and then counting the number of closest stones which all belong to the same team.

One approach with a simple implementation involves the insight that every stone simply contributes 1 point for its team if there are no stones from the opposing team which are closer to the button than it. For each team, we can iterate over each of its stones, and then determine whether or not it satisfies this condition by iterating over all of the other team's stones and checking if at least one of them has a smaller distance value.

Official Solution (C++)

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

int N, M, A[8], B[8];

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> A[i];
  }
  for (int i = 0; i < M; i++) {
    cin >> B[i];
  }
  int sA = 0, sB = 0;
  for (int i = 0; i < N; i++) {
    // This stone counts for Team A if there's no closer Team B stone.
    bool closerOpp = false;
    for (int j = 0; j < M; j++) {
      if (B[j] < A[i]) {
        closerOpp = true;
      }
    }
    if (!closerOpp) {
      sA++;
    }
  }
  for (int i = 0; i < M; i++) {
    // This stone counts for Team B if there's no closer Team A stone.
    bool closerOpp = false;
    for (int j = 0; j < N; j++) {
      if (A[j] < B[i]) {
        closerOpp = true;
      }
    }
    if (!closerOpp) {
      sB++;
    }
  }
  cout << sA << ' ' << sB << endl;
  return 0;
}
```

Problem I4/S2: Ride the Rocket

The first main insight to make is that each student should either walk all the way to their destination, or take the bus all the way. For example, the only reason a student might want to take the bus partially and then walk is to make space for another student getting on at that stop, but at that point those 2 students might as well have switched places from the start, so it can't be any more beneficial.

The second main insight to make is that students with further destinations are more likely to want to take the bus than students with closer ones. In other words, if only 1 of 2 students is going to take the bus, then it should be the one with the further destination, as the time saved by taking the bus will be multiplied over more stops.

The above pair of insights suggest the following greedy algorithm: Sort the students' destinations $D_{1..M}$, and process them in non-increasing order (the furthest ones first). For each student in that order, greedily have them either walk all the way or take the bus all the way, whichever is faster. Students assigned to taking buses may increase the bus travel times for later students, which is why it's important to process the students with the furthest destinations first, so that they get priority for filling the buses.

As for the details of computing the students' travel times, we can maintain 3 values during the greedy process: The sum s of travel times so far (initially 0), the time t at which the current bus will arrive at stop 1 (initially 0), and the number of remaining spots r on the current bus (initially C). The travel time if the student i walks is therefore $(D_i - 1)\times W$, while the travel time if they take the bus is $t + (D_i - 1)\times B$. s should be increased by the smaller of these 2 values. Then, if we've decided that this student should take the bus, we need to record that by decrementing r. If r becomes 0, then future students will need to take the next bus, meaning that we should increase t by P and reset r to C. After processing all M students in this fashion, s will contain the final answer.

The time complexity of this algorithm is $O(M \log M)$, which is the time required to optimally sort the values $D_{1..M}$.

Official Solution (C++)

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

int N, P, B, C, M, W, D[100000];
long long ans = 0;

int main() {
  cin >> N >> P >> B >> C >> M >> W;
  for (int i = 0; i < M; i++) {
    cin >> D[i];
  }
  sort(D, D + M);
  int busStart = 0, busSpace = C;
  // Process students with furthest destinations first.
  for (int i = M - 1; i >= 0; i--) {
    // Greedily either walk or take the bus, whichever is faster.
    long long walkTime = (long long)(D[i] - 1) * W;
    long long busTime = busStart + (long long)(D[i] - 1) * B;
    ans += min(walkTime, busTime);
    // If taking the bus, fill it up.
    if (busTime < walkTime) {
      busSpace--;
      // If the bus is full, later students must wait for the next bus.
      if (busSpace == 0) {
        busSpace = C;
        busStart += P;
      }
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem S3: Crosscountry Canada

This problem can be reinterpreted as asking for the shortest path on an implicit directed, weighted graph.

Each node in the graph should represent a particular state which you may be in during your trip. This can be described by a pair of integers $\{i, t\}$, where i is the city you're currently in, and t is the number of minutes which have gone by since your last Tim Horton's visit (or since the start of your trip). In total, there are $O(NL)$ nodes.

The edges in this graph should then correspond to valid transitions between such states, of which there are two types. The first type of transition involves stopping at a Tim Horton's. For each node $\{i, t\}$ such that $R_i = 1$, there exists an edge to node $\{i, 0\}$ with weight T. The second type of transition involves driving along a road. For each road i and value t, there exists an edge from node $\{A_i, t\}$ to node $\{B_i, t + C_i\}$ with weight C_i, as long as $t + C_i \le L$. There's also a similar edge for taking the road in the opposite direction. In total, there are $O((N + M)\times L)$ edges.

With this interpretation, the answer is the length of the shortest path from node $\{1, 0\}$ to any node $\{N, t\}$. This can be computed using the well-known "Dijkstra's algorithm" in $O(E \log(V))$ time, where $E = (N + M)\times L$ and $V = NL$.

Official Solution (C++)

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

struct edge {
  int i, t;
  edge(int i, int t): i(i), t(t) {}
};

struct state {
  int i, c, t;
  state(int i, int c, int t): i(i), c(c), t(t) {}
};

bool operator<(const state &a, const state &b) {
  return a.t > b.t;
}

int N, M, L, T, R[1001];
vector<edge> adj[1001];
priority_queue<state> q;

// MT[i][c] = min. time to reach city i with c mins since the last Tim Horton's.
int MT[1001][101] = {0};

int main() {
  cin >> N >> M >> L >> T;
  for (int i = 1; i <= N; i++) {
    cin >> R[i];
  }
  for (int i = 0; i < M; i++) {
    int a, b, c;
    cin >> a >> b >> c;
    adj[a].push_back(edge(b, c));
    adj[b].push_back(edge(a, c));
  }


  // Dijkstra's algorithm.
  memset(MT, 0x3f, sizeof MT);  // Initialize to large values.
  MT[1][0] = 0;  // Push the initial state.
  q.push(state(1, 0, 0));
  while (!q.empty()) {
    state s = q.top();
    q.pop();
    if (s.t != MT[s.i][s.c]) {
      continue;  // Already explored this state.
    }
    // Reached city N?
    if (s.i == N) {
      cout << s.t << endl;
      return 0;
    }
    // Consider stopping at Tim Horton's.
    if (R[s.i]) {
      state s2 = state(s.i, 0, s.t + T);  // Compute destination state.
      if (s2.t < MT[s2.i][s2.c]) {
        q.push(s2), MT[s2.i][s2.c] = s2.t;  // Push destination state.
      }
    }
    // Consider each adjacent road
    for (int i = 0; i < (int)adj[s.i].size(); i++) {
      int j = adj[s.i][i].i, t = adj[s.i][i].t;
      if (s.c + t > L) {
        continue;  // Taking the road would exceed the Tim Horton's limit?
      }
      state s2 = state(j, s.c + t, s.t + t);  // Compute destination state.
      if (s2.t < MT[s2.i][s2.c]) {
        q.push(s2);  // Push destination state.
        MT[s2.i][s2.c] = s2.t;
      }
    }
  }
  cout << -1 << endl;
  return 0;
}
```

Problem S4: Change

Let's start by solving the problem for relatively small values of K (up to a few thousand). This can be accomplished with dynamic programming. Let $DP[k]$ be the answer for $K = k$. That is, the maximum number of denominations no larger K which we can use such that K is unattainable.

For a given k, assuming that we'll be able to use at least 1 denomination, let's consider each possible value d ($2 \le d < k$) such that the largest denomination used will be d. d must not be one of the forbidden denominations, and must not be a divisor of k.

For a given d, the greedy algorithm will use coins of that denomination until it's $r = k \bmod d$ away from its target amount of money, essentially reducing us to a subproblem with $K = r$. Along the way, we can also freely use all non-forbidden denominations between $r + 1$ and d, inclusive.

Let $F(x, y)$ be the number of non-forbidden denominations in the interval $[x, y]$, which can be implemented in $O(\log N)$ time using binary search once the values $D_{1..N}$ are sorted. This brings us to the recurrence $DP[k] = \max\{ DP[k \bmod d] + F((k \bmod d) + 1, d) \}$ over all valid values of d.

The time complexity of the algorithm described above is $O(K^2 \log N)$, which is far too slow for full marks when K is as large as 10^9. We need to next make the insight that larger values of d (ones which are only slightly smaller than k) are generally more optimal than smaller values of d. This is fairly intuitive as choosing a smaller value D_1

over a larger value D_2 means skipping 1 or more valid denominations in the interval $[D_1 + 1, D_2]$ which we could otherwise have used. Of course, $DP[k \bmod D_1]$ could still turn out to be larger than $DP[k \bmod D_2]$, so it's not *always* optimal to use the first valid denomination smaller than k.

However, it's a fairly safe guess that the initial value of d should be no smaller than, let's say, $K – 5000$. This is convenient because, not only do we only need to try 5000 values of d to compute $DP[K]$, but we only need to have computed $DP[1..5000]$ to complement this (as $K \bmod d$ will be at most 5000). We can compute that many DP values efficiently enough.

To be more precise, we can prove that we only need to consider initial values of d no smaller than $K – N – 1$ (and so, only compute $DP[1..(N + 1)]$). If we find a valid value D_2 such that $K \bmod D_2 – 1$ is a non-forbidden denomination, then $DP[K \bmod D_2]$ is as large as possible (as we'll be able to use all smaller non-forbidden denominations), meaning that it can't be more optimal to skip D_2 in favour of any smaller value D_1. A value of D_2 meeting these criteria no smaller than $K – N – 1$ must exist (it may be as small as exactly $K – N – 1$ if all denominations in the interval $[K – N, K – 1]$ are forbidden, for example). In the end, then, we can achieve a time complexity of $O(N^2 \log(N))$.

Official Solution (C++)

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

const int MAXK = 2000, DLIM = 2018;
int K, N, D[MAXK], ans = 0;
int DP[DLIM + 1]; // DP[i] = max. number of denominations no greater than i such that i cannot be made.

int CountAllowed(int a, int b) {  // Returns the number of allowed denominations in the range a..b.
  int t = b - a + 1;
  int d = lower_bound(D, D + N, b + 1) - lower_bound(D, D + N, a);
  return t - d;
}

int main() {
  cin >> K >> N;
  for (int i = 0; i < N; i++) {
    cin >> D[i];
  }
  sort(D, D + N);
  // Precompute DP values up to at most ~2000.
  for (int i = 1; i <= min(K, DLIM); i++) {
    for (int m = 2; m < i; m++) {  // Consider each possible largest used denomination m.
      if (i % m == 0 || !CountAllowed(m, m)) continue;
      int c = DP[i % m] + CountAllowed(i % m + 1, m);
      DP[i] = max(DP[i], c);
    }
  }
  // If K is at most ~2000, then we already have the answer.
  if (K <= DLIM) {
    cout << DP[K] << endl;
    return 0;
  }
  // Otherwise, an optimal largest used denomination must exist within at most ~2000 of K.
  for (int m = K - DLIM; m < K; m++) {
    if (K % m == 0 || !CountAllowed(m, m)) continue;
    int c = DP[K % m] + CountAllowed(K % m + 1, m);
    ans = max(ans, c);
  }
  cout << ans << endl;
  return 0;
}
```