Soluciones OIFem 2020
Entrenamiento 4

POI
Teoría

- Ordenamiento con comparador propio

Solución

Primero, calcularemos a partir de la tabla de entrada cuántos concursantes han resuelto cada problema. Sabiendo esto, sumaremos la puntuación de cada jugador y contaremos el número de problemas que ha resuelto.

Usando un comparador que simula el ordenamiento descrito en el enunciado, ordenamos a los jugadores. Finalmente, buscamos a Phillip en el ránking e imprimimos su puntuación y clasificación final.

Código

C++

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

vector<int> puntos, solucionesPorProblema, problemasResueltos;

bool comp(int i, int j) {
        // comparador para ordenar los concursantes
        if (puntos[i] != puntos[j]) return puntos[i] > puntos[j];
        else if (problemasResueltos[i] != problemasResueltos[j]) return
        ↪  problemasResueltos[i] > problemasResueltos[j];
        return i < j;
}
```

```cpp
void ordenarConcursantes(vector<vector<int>> & tabla, vector<int> & concursantes) {
	int T = (int) tabla[0].size(), N = (int) concursantes.size();
	// guardamos en un vector cuántos concursantes han resuelto cada problema
	for (int i = 0; i < T; i++) {
		for (int j = 0; j < N; j++) solucionesPorProblema[i] += tabla[j][i];
	}
	// calculamos jugador a jugador cuántos puntos ha obtenido y cuántos problemas ha resuelto
	for (int i = 0; i < N; i++) {
		for (int j = 0; j < T; j++) {
			if (tabla[i][j]) {
				puntos[i] += (N-solucionesPorProblema[j]);
				problemasResueltos[i]++;
			}
		}
	}
	// ordenamos a los concursantes con el algoritmo descrito en las instrucciones
	sort(concursantes.begin(), concursantes.end(), comp);
}

int main() {
	int N, T, P;
	cin >> N >> T >> P;
	vector<vector<int>> tabla(N, vector<int>(T));
	vector<int> concursantes(N);
	puntos.assign(N, 0);
	problemasResueltos.assign(N, 0);
	solucionesPorProblema.assign(T, 0);
	for (int i = 0; i < N; i++) {
		concursantes[i] = i;
		for (int j = 0; j < T; j++) {
			cin >> tabla[i][j];
		}
	}

	ordenarConcursantes(tabla, concursantes);

	// Buscamos a Phillip en la clasificación e imprimimos sus puntos y su puesto final
	for (int i = 0; i < N; i++) {
		if (concursantes[i] == P-1) {
			cout << puntos[P-1] << " " << i+1 << "\n";
			break;
		}
	}

	return 0;
}
```

Las galletas de Jan

Teoría

- Ordenamiento
- Algoritmos voraces

Solución

La solución óptima consiste en comer la última galleta de todos los tipos posibles, consiguiendo así maximizar el número de tipos de galleta probadas, ya que la última es la que tardará más en desaparecer. De cada tipo de galleta guardamos el momento en el que desaparece la última de ese tipo. Ordenamos estos últimos momentos de forma ascendente.

Vamos tipo a tipo viendo si su última galleta desaparecerá antes de que termine el descanso o si Jan llegará a tiempo. Si llega a tiempo, la comerá, sumándola al total y actualizando el momento en el que terminará su descanso. Sabemos que esto es óptimo, ya que las galletas que vengan después en la lista desaparecerán más tarde y, por tanto, no perdemos nada por comer la actual.

Código

C++

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

int galletas (vector<int> & tipoATiempo) {
    int totalGalletas = 0, momentoActual = 0;
    sort(tipoATiempo.begin(), tipoATiempo.end()); // ordenamos los tipos de
    ↪ galleta por orden de cuándo sale la última galleta de ese tipo
    for (int i = 0; i < (int) tipoATiempo.size(); i++) {
        if (tipoATiempo[i] >= momentoActual) {
                // podemos comer una galleta de ese tipo
            totalGalletas++;
            momentoActual += totalGalletas;
        }
    }
    return totalGalletas;
}
```

```cpp
int main() {
    int t, m, n, tipo, momento;
    vector<int> tipoATiempo;
    cin >> t;
    while(t--) {
        cin >> m >> n;
        tipoATiempo.assign(m, 0);
        for (int i = 0; i < n; i++) {
                // guardamos de cada tipo de galleta la última galleta en salir de
                ↪  ese tipo
            cin >> tipo >> momento;
            tipoATiempo[tipo-1] = max(tipoATiempo[tipo-1], momento);
        }
        cout << galletas(tipoATiempo) << '\n';
    }
    return 0;
}
```

Monedas

Teoría

- Ordenamiento

Solución

Primero, ordenamos las monedas de menor a mayor. Si la moneda en i-ésima posición tiene un valor m, todos los valores de 1 a $m-1$ tienen que poder obtenerse con las primeras $i-1$ monedas. Si no, la primera suma que no podemos obtener tendrá un valor inferior a m, que será 1 más que la suma de las primeras $i-1$ monedas.

Código

C++

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

long long int minimaCantidad(vector<long long int> & monedas) {
    sort(monedas.begin(), monedas.end()); // ordenamos las monedas de menor a
    ↪ mayor
    long long int respuesta = 0;
    for (int i = 0; i < (int) monedas.size(); i++) {
        if (monedas[i] > respuesta+1) {
            // ya hemos encontrado el máximo
            break;
        }
        respuesta += monedas[i];
    }
    return respuesta+1; // retornamos la suma máxima a la que hemos llegado más 1
}


int main() {
    int t, n;
    vector<long long int> monedas;
    cin >> t;
    while(t--) {
        cin >> n;
        monedas.assign(n, 0);
        for (int i = 0; i < n; i++)
            cin >> monedas[i];
        cout << minimaCantidad(monedas) << '\n';
    }
    return 0;
}
```

Monedas 3

Teoría

- Ordenamiento en orden descendente

Solución

Para resolver este problema, basta con imitar de forma cuidadosa los movimientos de Pedro. Ordenamos las monedas de mayor a menor, como él, y vamos moneda a moneda viendo si es menor o igual al total restante. Si este es el caso, pagamos con esa moneda, restando su valor del total restante. Una vez comprobadas todas las monedas, retornamos el total restante (este puede ser 0 si se ha pagado con exactitud).

Código

C++

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

long long int diferencia(long long int k, vector<long long int> & monedas) {
    sort(monedas.begin(), monedas.end(), greater<int>()); // ordenamos las monedas en orden descendente
    for (int i = 0; i < (int) monedas.size(); i++) {
        if (monedas[i] <= k)
            k -= monedas[i]; // vamos restando del total a pagar las monedas que entrega Pedro al dependiente
    }
    return k; // retornamos lo que falta por pagar
}

int main() {
    int t, n;
    long long int k;
    cin >> t;
    while(t--) {
        cin >> n >> k;
        vector<long long int> monedas(n);
        for (int i = 0; i < n; i++)
            cin >> monedas[i];
        cout << diferencia(k, monedas) << '\n';
    }
    return 0;
}
```

Horarios

Teoría

- Ordenamiento con comparador propio
- Estrategia "sort and greedy" (ordena y luego aplica voraz)

Solución

Ordenamos las asignaturas en función de cuándo terminan y, de una forma voraz, vamos seleccionando todas las que podemos. Llevamos la cuenta en todo momento de cuántas asignaturas hemos escogido y de cuándo termina la última asignatura seleccionada que, por el comparador que hemos seleccionado, será la que termine más tarde. Vamos elemento a elemento de la lista y, si la asignatura en cuestión no empieza antes de que termine la última que hemos seleccionado, la seleccionaremos también, actualizando la cuenta y el final de la última asignatura.

Código

C++

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

bool comparadorSegundo(pair<int, int> A, pair<int, int> B) {
    if (A.second != B.second)
        return A.second < B.second;
    return A.first < B.first;
}

int maxAsignaturas(vector<pair<int, int>> & asignaturas) {
    if (asignaturas.empty()) {
        // no hay asignaturas, por lo que la respuesta es 0
        return 0;
    }

    sort(asignaturas.begin(), asignaturas.end(), comparadorSegundo); // ordenamos las asignaturas en función de cuándo terminan

    int respuesta = 1, momentoActual = asignaturas[0].second;
    for (int i = 1; i < (int) asignaturas.size(); i++) {
        // si la asignatura empieza después del final de nuestra actual última asignatura, la escogemos y actualizamos el final y la cuenta
        if (momentoActual <= asignaturas[i].first) {
            momentoActual = asignaturas[i].second;
            respuesta++;
        }
    }
    return respuesta;
}
```

```cpp
int main() {
    int n, m;
    vector<pair<int, int>> asignaturas;
    while(cin >> n >> m) {
        asignaturas.assign(n, pair<int, int>());
        for (int i = 0; i < n; i++)
            cin >> asignaturas[i].first >> asignaturas[i].second;
        cout << maxAsignaturas(asignaturas) << '\n';
    }
    return 0;
}
```

Sum of Three Values

Teoría

- Método de los dos punteros

Solución

Para no perder el índice (posición) inicial de cada número de la lista, guardaremos la lista como un vector de parejas, donde el primer número de la pareja es el valor del número y el segundo corresponde al índice inicial de dicho número.

Ordenamos la lista por el valor de los números. Entonces, procedemos número a número de la lista en orden ascendente. Fijamos el primer número del trío de esta forma, lo que nos permite usar el método de los dos punteros con el resto de la lista para ver si hay una pareja que no incluya este número inicial y que, junto a él, sume el total deseado. Si este es el caso, imprimimos los índices iniciales y cortamos la búsqueda. Si no, procederemos, cambiando el número fijo del trío hasta concluir que no hay solución posible o encontrar una.

Código

C++

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

bool haySolucion(int X, vector<pair<int, int>> &numeros) {
        sort(numeros.begin(), numeros.end()); // ordenamos la lista de números de menor a mayor
        int i, j, N = (int) numeros.size(), sumaActual, sumaDeseada;

        for (int k = 0; k < N; k++) {
                sumaDeseada = X - numeros[k].first; // guardamos el total necesario menos el número fijo del trío
                i = 0;
                j = N - 1;
                while (i < j) {
                        // usamos el método de los dos punteros para encontrar dos índices diferentes a k que sumen sumaDeseada
                        if (numeros[i].first >= sumaDeseada)
                                break;
                        if (i == k) {
                                i++; // evitamos que i y k coincidan
                                continue;
                        }
                        if (j == k) {
                                j--; // evitamos que j y k coincidan
                                continue;
                        }
```

```
                        sumaActual = numeros[i].first + numeros[j].first;

                        if (sumaActual == sumaDeseada) {
                                // encontramos una solución
                                cout << numeros[i].second << " " <<
                                ↪  numeros[j].second << " "
                                                << numeros[k].second << "\n";
                                return true;
                        } else if (sumaActual < sumaDeseada)
                                i++;
                        else
                                j--;
                }
        }
        return false;
}

int main() {
        int N, X;
        cin >> N >> X;
        vector<pair<int, int>> numeros(N, pair<int, int>());
        for (int i = 0; i < N; i++) {
                // en cada posición de números, guardamos también el índice
                ↪  inicial
                cin >> numeros[i].first;
                numeros[i].second = i+1;
        }
        if (!haySolucion(X, numeros)) cout << "IMPOSSIBLE\n";
        return 0;
}
```

Stove

Teoría

- Ordenamiento en orden descendente
- Algoritmos voraces

Solución

Encontramos las diferencias entre el momento en el que se apaga la estufa tras el i-ésimo invitado, en el caso en el que se encienda solo para él, y cuando llega el (i+1)-ésimo invitado para todos los invitados de la lista.

Ordenamos estos huecos de mayor a menor. Obtenemos la respuesta calculando cuánto tiempo ahorramos apagando la estufa en los $K-1$ mayores huecos.

Código

C++

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

int solucion(int K, vector<int> & momentos) {
        int N = (int) momentos.size();
        vector<int> huecos(N-1);
        // calculamos las diferencias entre cuándo llegan los distintos invitados,
        ↪  teniendo en cuenta que las estufas están encendidas al menos 1 segundo
        for (int i = 0; i < N-1; i++)
                huecos[i] = momentos[i+1]-momentos[i]-1;
        // ordenamos las diferencias de mayor a menor
        sort(huecos.begin(), huecos.end(), greater<int>());
        // apagamos las estufas durante los K-1 mayores huecos
        int respuesta = momentos[N-1]+1 - momentos[0];
        for (int i = 0; i < K-1; i++) {
                respuesta -= huecos[i];
        }
        return respuesta;
}
```

```
int main() {
        int N, K;
        vector<int> momentos;
        cin >> N >> K;
        momentos = vector<int>(N);
        for (int i = 0; i < N; i++)
                cin >> momentos[i];
        cout << solucion(K, momentos) << '\n';
        return 0;
}
```

Paired Up

Teoría

- Pairs
- Ordenamiento

Solución

La idea es ordenar las vacas por orden de producción y realizar emparejamientos mayor-menor sucesivamente. Nótese que es posible, cuando se tiene un número grande de vacas con el mismo valor, realizar emparejamientos "en bloque" en vez de uno en uno.

Código

C++

```cpp
using namespace std;

int main(void) {
  vector<pair<int, int>> v;
  int N;
  ifstream fin("pairup.in"); // para leer input de un archivo de texto
  ofstream fout("pairup.out"); // para hacer output a un archivo de texto
  fin >> N;
  for (int i = 0; i < N; i++) {
    int x, y;
    fin >> x >> y;
    v.push_back(make_pair(y, x));  // {valor, numero de instancias}
  }
  // Ordenar las vacas por "valor".
  sort(v.begin(), v.end());
  int M = 0, i = 0, j = N - 1;

  // Realizar los emparejamientos.
  while (i <= j) {
    int x = min(v[i].second, v[j].second);
    if (i == j)
        x /= 2;
    M = max(M, v[i].first + v[j].first);
    v[i].second -= x;
    v[j].second -= x;
    if (v[i].second == 0)
        i++;
    if (v[j].second == 0)
        j--;
  }
  fout << M << "\n";
  return 0;
}
```

Mazmorra

Teoría

- Ordenamiento

Solución

Ordenamos las fuerzas de los monstruos de menor a mayor. Vamos monstruo a monstruo en este orden, comparando nuestra fuerza con la del monstruo. Si nuestra fuerza es menor, paramos de contar, ya que los monstruos restantes tienen fuerzas mayores o iguales. Si este no es el caso, incrementamos nuestra fuerza por 1 y vamos a por el siguiente monstruo. Seguimos así hasta que nos topamos con un monstruo que nos hace parar de contar o hasta luchar contra todos.

Código

C++

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


int main() {
    int t, n, miFuerza;
    vector<int> fuerza;
    cin >> t;
    while(t--) {
        cin >> n;
        fuerza.assign(n, 0);
        for (int i = 0; i < n; i++)
            cin >> fuerza[i];
        sort(fuerza.begin(), fuerza.end()); // ordenamos las fuerzas de menor a
        ↪  mayor
        miFuerza = 0;
        for (int i = 0; i < (int) n; i++) {
            if (miFuerza < fuerza[i]) {
                // paramos de contar, ya que los monstruos restantes tienen
                ↪  fuerzas mayores o iguales a fuerza[i]
                break;
            }
            miFuerza++;
        }
        cout << miFuerza << '\n';
    }
    return 0;
}
```

Detecting Molecules

Teoría

- Ordenación.
- Voraz.

Solución

La idea principal es utilizar una estrategia voraz tras haber ordenado las moléculas. Tras esto, iremos acumulando los valores. Si la suma acumulada excede la cota superior, podemos descartar los valores de menor tamaño considerados, de esta forma satisfaciendo la condición impuesta por la cota inferior. Asimismo. al comprobar si también se satisface la condición de la cota superior, habremos encontrado una solución válida.

Código

C++

```cpp
using namespace std;

// Podemos utilizar una estrategia voraz.
vector<int> find_subset(int l, int u, vector<int> w) {
  int n = w.size();
  vector<pair<int, int>> a(n);
  for (int i = 0; i < n; i++) a[i] = {w[i], i};
  // ordenemos los valores de menor a mayor.
  sort(a.begin(), a.end());
  int k = 0;
  long long sum = 0;
  for (int i = 0; i < (int) a.size(); i++) {
    sum += a[i].first;
    // Si nos "hemos pasado", descontar por los valores de menor tamaño hasta
    // estar de nuevo dentro del rango permitido.
    while (sum > u) {
      sum -= a[k++].first;
    }
    // En este punto se cumple sum <= u.
    if (sum >= l) {
      // Si ademas se cumple l <= sum, tenemos l <= sum <= u, una solución
      // válida: retornarla.
      vector<int> ans;
      for (int j = k; j <= i; j++) {
        ans.push_back(a[j].second);
      }
      return ans;
    }
  }
  return {};
}
```

Rice Hub

Teoría

- Búsqueda binaria.

Solución

La idea principal es darnos cuenta de que una solución válida consistirá en una secuencia de campos de arroz consecutiva, y el almacén (hub) tendrá que estar situado en el punto centro de esta secuencia —en otras palabras, en la mediana—. Para determinar la longitud y posición de este segmento solución podemos utilizar búsqueda binaria.

Código

C++

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

bool valida(int len, long long budget, int *x, int n) {
  long long cost = 0;
  int a = 0;
  int b = len - 1;
  int mid = (a + b) / 2;
  long long left = 0;
  long long right = 0;
  for (int i = a; i <= b; i++) {
    cost += abs(x[i] - x[mid]);
  }
  left = mid - a;
  right = b - mid;
  if (cost <= budget) return true;
  while (b + 1 < n) {
    cost -= x[mid] - x[a];
    cost += left * (x[mid + 1] - x[mid]);
    cost -= right * (x[mid + 1] - x[mid]);
    mid++;
    a++;
    b++;
    cost += x[b] - x[mid];
    if (cost <= budget) return true;
  }
  return false;
}
```

```
// Utilizar búsqueda binaria sobre la longitud del rango de campos de arroz.
// Para cada longitud candidato, representada por el punto medio "mid" habitual
// en búsqueda binaria, comprobamos si respresenta un solución válida.
int besthub(int n, int l, int x[], long long budget) {
  int a = 1;
  int b = n;
  int res = -1;
  while (a <= b) {
    int mid = (a + b) / 2;
    if (valida(mid, budget, x, n)) {
      res = mid;
      a = mid + 1;
    } else {
      b = mid - 1;
    }
  }
  return res;
}
```