Soluciones OIFem II Nivel 2

Entrenamiento 4

Interval Covering

Teoría

- Ordenamiento
- Algoritmos voraces (greedy)

Solución

En los problemas en los que debemos procesar una serie de valores numéricos, un pregunta con la que siempre vale la pena empezar es: ¿Me ayuda de algo si los valores estan ordenados? En la mayoría de los problemas, ordenar el input puede ayudar a encontrar una solución y en términos de complejidad, suele ser eficiente ya que hacer un sort solo cuesta $O(N \log N)$.

Entonces, asumiendo que tenemos la lista de números ordenada, podemos analizar el punto que esta más a la izquierda. ¿De qué forma podemos cubrir este punto y que a su vez nos ayude a minizmar la cantidad total de segmentos a usar? Lo más eficiente es usar un segmento cuyo extremo izquierdo empieze en dicho punto. Como contraejemplo, podemos notar que Si existiera otra solución que no cumpla dicha regla, siempre podremos mover el segmento para que empiece en dicho punto ya que, al ser el que está en el extremo izquierdo, no dejaremos ningún punto sin cubrir.

Luego de colocar dicho segmento, algunos puntos serán cubiertos, y podemos repetir la misma lógica con los puntos sin cubrir. Dando como resultado el algoritmo greedy para la solución.

> Nota
>
> El problema trabaja con valores enteros, es decir con decimales. Si bien la solución presentada usa comparaciones simples, como $a <= b$, y el código es aceptado por el juez, es posible que en otros casos comparar numeros decimales no funcione de la manera que esperamos y debamos comparar usando una tolerancia. Para mayor información puede consultar el siguiente enlace

Código

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
```

```
    int N;
    while (cin >> N) {
        vector<double> v(N);
        for (int i = 0; i < N; i++) cin >> v[i];
        sort(v.begin(), v.end());
        int pos = 0, ans = 0;
        while (pos < N) {
            ans++;
            double max_value = v[pos] + 1.0;
            while (pos < N and v[pos] <= max_value) pos++;
        }
        cout << ans << endl;
    }
}
```

Setting the Video

Teoría

- Ordenamiento
- Estructuras de datos: cola de prioridad
- Algoritmos voraces (greedy)

Solución

Similar al problema anterior, en este caso debemos procesar episodios en el tiempo. Nuevamente vale la pena preguntarse si al ordenarlos es posible encontrar alguna ventaja. Sin embargo, cuando se trabaja con intervalos en el tiempo, el ordenamiento no es directo ya que cada intervalo tiene un inicio y un final, y debemos decidir bajo que criterio queremos ordenarlos.

Para un episodio X en particular, conocemos el tiempo que empieza y el tiempo que acaba. Si quisieramos que este episodio forme parte de la solución, y además sea el último episodio que veamos, deberiamos saber cual es la máxima cantidad de episodios que podemos ver antes de que empiece el episodio X. Si tuvieramos este valor para todos los episodios, podríamos quedarnos con el máximo valor entre ellos, y esa sería la respuesta.

Ahora, si analizamos los episodios luego de ordernarlos por inicio, cuando toque analizar el episodio X, podemos notar que previamente ya se han analizado todos los episodios que terminaron antes de empiece X (La unica forma de que un episodio termine antes que X, es que también haya empezado antes que X).

Sin embargo, es posible que también hayamos analizado un episodio que empezó antes de X, pero que terminara luego de X. Entonces, solo debemos considerar aquellos episodios cuyo final es antes de ini. Para ello podemos usar una estructura de datos que mantenga los elementos ordenados por final, y antes de procesar el episodio X retiramos aquellos episodios que terminaron antes que X. Este es un caso de uso para un PriorityQueue, o cola de prioridad. Que permite insertar/eliminar elementos, guardarlos bajo un orden, y obtener el primer elemento bajo ese orden.

Código

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> ii;

int main() {
    int N;
    while (cin >> N) {
        vector<ii> v(N);
        for (int i = 0; i < N; i++) cin >> v[i].first >> v[i].second;
        sort(v.begin(), v.end());  // Ordenamos los episodios por inicio
        int ans = 0;
        // Definimos una cola de prioridad, por defecto esta estructura ordena
        // los elementos decresciente mente Pasando el 3er parametro greater<>
        // podemos cambiar el criterio a creciente
        priority_queue<ii, vector<ii>, greater<ii> > PQ;

        for (auto episode : v) {
```

```
            int ini = episode.first;
            int fin = episode.second;

            // Mientras haya elementos en la cola, agregar los episodios que
            // hayan terminado antes de ini
            while (!PQ.empty() and PQ.top().first < ini) {
                ii nxt = PQ.top();
                PQ.pop();  // eliminamos el elemento de la cola
                ans = max(ans, nxt.second);  // Guardamos la mejor respuesta
            }

            // Insertamos el episodio con su final, y la maxima cantidad de
            // episodios que se pueden ver terminando en este episodio
            PQ.push(ii(fin, ans + 1));
        }
        cout << ans + 1 << endl;
    }
}
```

Product Trees

Teoría

- Definición de árboles binarios
- Matemática
- Algoritmos voraces (greedy)

Solución

En problemas en donde debemos construir una posible solución, a veces también llamados algoritmos constructivos, se recomienda empezar con casos pequeños, y luego intentar generalizar una solución.

Para el caso más simple, cuando n=1, la única solución es tener un árbol de 1 nodo. Para el caso n=2, la única solución es tener un arbol con una raíz y dos hojas ya que no importa en que orden esten las hojas.

Lo interesante empieza a ocurrir en caso n=3, la estructura del árbol es única: Un nodo raíz con dos hijos, uno de los hijos tiene 2 hojas y el otro hijo es una hoja. Sin embargo, aquí debemos encontrar la manera óptima de asignar los 3 valores a las 3 hojas. Sean estos 3 valores a, b, c con $a <= b <= c$. Existen 3 posibles soluciones dependiendo de que número se asigne a la hoja de menor profundidad

- $a^3 * b^3 * c^2$
- $a^3 * c^3 * b^2$
- $b^3 * c^3 * a^2$

Dado que todos los números son mayores o iguales a 1, lo óptimo será tomar que los dos números menores sean los que esten elevados a la potencia 3, y el mayor sea elevado a la potencia 2. Entonces lo mejor será la primera alternativa.

Para n=4 la cantidad de casos se hace más grande, pero podemos notar un patrón del caso anterior. Si tomamos dos números, sean a y b, y los reemplazamos por un número cuyo valor es $a * b$, entonces volveremos a caer al caso n=3 para el cual ya tenemos la solución. ¿Qué números debemos elegir? Como queremos minimizar el producto total, lo mejor será tomar los dos número más pequeños nuevamente. Ese sería nuestro algoritmo voraz.

Podemos ayudarnos de una cola de prioridad para mantener los números en un orden creciente y siempre tomar los dos más pequeños.

Código

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    int N;
    while (cin >> N and N) {
        double ans = 1.0;

        // La cola de prioridad debe estar en orden decreciente
        priority_queue<double, vector<double>, greater<double> > PQ;
```

```
12
13          for (int i = 0; i < N; i++) {
14              double aux;
15              cin >> aux;
16              ans *= aux;
17              PQ.push(aux);
18          }
19
20          while (PQ.size() >= 2) {
21              double l = PQ.top();
22              PQ.pop();
23              double r = PQ.top();
24              PQ.pop();
25
26              ans *= (l * r);
27              PQ.push(l * r);
28          }
29
30          // Solo nos importan 7 cifras significativas
31          while (ans >= 10000000) ans /= 10.0;
32
33          // Casteamos el double a long para retirar los decimalos
34          // Importante notar que esta operacion no redondea el decimal
35          cout << long(ans) << endl;
36      }
37  }
```

Non-overlapping Intervals

Teoría

- Ordenamiento
- Estructuras de datos: cola de prioridad
- Algoritmos voraces (greedy)

Solución

El problema es muy parecido a la pregunta "Setting the Video" que ya se analizó. En este caso nos piden cual es la mínima cantidad de intervalos que debemos remover para que que los restantes no se superpongan. Es pregunta es equivalente a cual es la máxima cantidad de intervalos que no se superponen, que es lo mismo que nos pedían en la pregunta anterior. Entonces nuestro algoritmo debería funcionar y solo deberíamos devolver la cantidad total de intervalos menos la cantidad máxima de intervalos que no se superponen.

La única diferencia es que en esta pregunta es posible que los extremos de dos intervalos coincidan, lo cual no era permitido en "Setting the Video". Solo basta cambiar la comparación al momento de procesar los elementos de la cola de prioridad.

Código

```cpp
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {

        sort(intervals.begin(), intervals.end());
        int ans = 0;
        priority_queue<pair<int,int>, vector< pair<int,int> >, greater< pair<int,int> > >
        ↪ PQ;
        for (auto interval : intervals) {
            int ini = episode[0];
            int fin = episode[1];

            // Aqui la comparacion cambia de < a <= para considerar que es posible que los
            ↪ extremos se superpongan
            while (!PQ.empty() and PQ.top().first <= ini) {
                pair<int,int> nxt = PQ.top();
                PQ.pop();
                ans = max(ans, nxt.second);
            }
            PQ.push(make_pair(fin, ans + 1));
        }
        return intervals.size() - ans - 1;

    }
};
```

Largest Number

Teoría

- Libreria STL
- Simulación

Solución

En este problema vamos a empezar construyendo una solución, que tal vez no se la mejor, pero veremos como podemos mejorarla. Para simplificar el análisis digamos que todos los números son de 1 dígito.

Si tenemos 7, 4, 3, 8, armemos el número 7438. ¿Cómo podemos mejorar esta respuesta? Notamos que entre los dos últimos elementos (3,8) nos conviene más si el 8 va delante, ya que tiene mayor valor. Entonces nos quedaría 7483. Podemos repetir el mísmos análisis hasta llegar a 8743.

Pero esta comparación no puede ser númerica, ¿qué pasa cuando tenemos número de distinta cantidad de cifras? Si tenemos 4 y 13, podríamos armar 134 si los comparamos por valor, pero notamos que es mejor el valor 413.

Entonces aquí podemos definir nuestro criterio de comparación, sea la operación concat(a,b) que da como resultado el concatenar a y b, si para dos número consecutivos a y b en la solución encontramos que concat(b,a) ¿ concat(a,b) entonces combiene intercambiarlos.

El procedimiento antes descrito corresponde a un algoritmo de ordenamiento: Bubble Sort pero con una función de comparación personalizada. No necesitamos usar necesariamente Bubble Sort, ya que es ineficiente, ya que Usando la librería STL podemos definir la función de comparación para el método ”sort”.

Es importante notar que el enunciado del problema indica ”número no negativos”, lo que significa que hay un caso en el que todos los números son 0. Nuestro algoritmo generará un string de muchos ceros para este caso, cuando la respuesta deber ser un único 0.

Código

```cpp
// Pasamos los parametros por referencia, porque al ser strings pasarlos por valor es costoso
// Nuestra funcion de comparacion debe devoler verdadero cuando a debe ir antes que b en el arreglo ordenado
// Para mayor informacion de funciones de comparacion personalizadas
// https://www.geeksforgeeks.org/sort-an-array-of-dates-in-ascending-order-using-custom-comparator/
bool cmp(string &a, string &b) {
    return (a+b)>(b+a);
}

class Solution {
public:
    string largestNumber(vector<int>& nums) {

        vector<string> strings;

        // Usamos la funcion to_string que funciona bien para enteros
        // https://www.geeksforgeeks.org/stdto_string-in-cpp/
```

```
        for(auto n: nums) strings.push_back(to_string(n));

        // El 3er argumento es la funcion de comparacion
        sort(strings.begin(), strings.end(), cmp);

        // Si luego de ordenar nuestro string mas significativo es 0
        // significa que estamos en el caso de puros ceros
        if (strings[0] == "0") return "0";

        string ans = "";
        for(auto s: strings) ans += s;
        return ans;

    }
};
```

Maximum Number of Consecutive Values You Can Make

Teoría

- Algoritmos voraces

Solución

Se busca tener la máxima cantidad de números consecutivos. Una forma de analizar es tratando de generarlos en orden. Siempre tendremos el número cero, ya que es el conjunto vacío. La única forma de que podamos formar el número 1, es que tengamos el 1 en el arreglo. Luego, para formar el número 2, dado que ya formamos el 0 y el 1, existen 2 opciones: que tengamos otro 1, o que tengamos el 2. Luego, para formar el número 3, dado que ya formamos el 0, 1 y 2, existen 3 opciones: que tengamos otro 1, o un 2 o un 3. Podemos seguir esta misma lógica y notamos que en cada paso vamos a necesitar un número mayor, pero siempre en orden creciente.

Si tenemos armados los números hasta 100, y el siguiente número, considerando que estan en orden, es 102, no hay forma de armar el número 101. Entonces ahí encontramos una limitante. Por ello convendrá procesar los números en orden.

Código

```cpp
class Solution {
public:
    int getMaximumConsecutive(vector<int>& coins) {
        sort(coins.begin(), coins.end());
        int ans = 0;
        for(auto coin: coins) {
            if (coin > ans + 1) break;
            ans +=coin;
        }
        return ans+1;
    }
};
```

Minimum Cost to Hire K Workers

Teoría

- Ordenamiento

Solución

Imaginemos que decidimos invertir un monto T para pagar a los k trabajadores elegidos. Y lo que nos piden es minimizar K. Según la primera primera condición, el salario de cada trabajador debe ser proporcional a su calidad. Entonces, debemos repartir T de manera proporcional a las calidades.

Si tenemos tres trabajadoes cuyas calidades son 1, 2 y 3. Entonces sus salarios deben ser k, 2k y 3k respectivamente, siendo la suma 6k = T. Aun no sabemos el valor exacto de k, pero debe ser tal que se minimize T. Como T es directamente proporcional a k, entonces minizar T es equivalente a minimizar k.

Ahora, según la segunda condición, cada trabajador tiene una expectativa mínima, sean estas expectativas e1, e2 y e3 para los tres trabajadores. Entonces se debe cumplir que $k >= e1, 2k >= e2, 3k >= e3$.

Entonces, ¿qué valor de k debemos elegir? El mínimo valor k, tal que se cumplan las tres equaciones. Para dicho valor al menos el salario de uno de los trabajadores será igual a su expectativa mínima. Caso contrario aun sería posible reducirles el sueldo a todos. Entonces, debemos elegir a que trabajador vamos a pagarle el sueldo mínimo, y en base a eso ver si es posible cumplir la 2da condición para el resto de trabajadores.

> Nota
>
> Analizar las restricciones a los datos de entrada puede darnos pistas de la solución esperada. En este caso, el valor de $n <= 1e4$ el cual, por experiencia, indica que la solución puede ser cuadrática. Para valores mas grandes como 1e5 o 1e6, estamos restringidos a encontrar una solución logarítmica o lineal.
> Para mayor referencias pueden consultar el siguiente enlace

Código

```cpp
class Solution {
public:
    double mincostToHireWorkers(vector<int>& quality, vector<int>& wage, int k) {
        int n = quality.size();
        double ans = INT_MAX; // Un valor muy grande

        // leader es el trabajador para quien fijaremos el salario minimo
        for (int leader = 0; leader < n; leader++) {
            // multiplicamos * 1.0 para convertir el valor a double, y no perder los
            ↪  decimales
            // sino la operacion seria entre dos enteros

            double k = (wage[leader] * 1.0) / quality[leader];

            vector<double> potential_wages;

            for(int worker = 0; worker < n; worker++) {
                double potential_wage = optimal_ratio * quality[worker];

```

```
                // Si el salario no llega a su expectativa, no lo consideramos
                if(potential_wage < wage[worker]) continue;

                potential_wages.push_back(potential_wage);
            }

            // Si no hay suficientes trabajadores, saltamos al siguiente leader
            if(potential_wages.size() < k) continue;

            // Ordenamos para quedarnos con los k menores
            sort(potential_wages.begin(), potential_wages.end());

            double candidate_answer = 0;
            for(int i = 0; i < k; i++) {
                candidate_answer += potential_wages[i];
            }

            ans = min(ans, candidate_answer);
        }

        return ans;
    }
};
```

Array Destruction

Teoría

- Simulación
- Ordenamiento
- Estructuras STL

Solución

El problema nos plantea un procedimiento que debemos simular. En este tipo de problemas se recomienda seguir el proceso en papel, para al menos los casos de prueba que tenemos o alguno que podamos crear, y así encontrar alguna observación que nos permita solucionarlo.

Debemos definir un número x elegiremos 2 números a y b tal que a + b = x. La primera observación importante es que todos los números son positivos, es decir que a y b son menores que x, y como debemos reemplazar x por uno de los números, entonces x va decrecer en cada paso.

Otra observación es que si elegimos dos números a y b, y dejamos un número c tal que c es el mayor de los tres, no será posible eliminar c. Esto debido a que x será reemplazado por a o por b, y luego de eso no será posible encontrar una pareja para c tal que sumen a o b. Es decir, que en cada operación siempre debemos tomar el número de mayor valor.

La última observación es que, una vez definamos un número X, el proceso queda completamente definido y puede simularse:

- Definimos un número X
- Tomamos el mayor elemento del arreglo, sea este max
- Buscamos si el número (x-max) esta en arreglo. Si existe eliminamos ambos números y reemplazamos x por el máximo entre ellos. Si no existe significa que no hay solución.
- Repetimos hasta que el arreglo quede vacío o no encontremos solución.

Entonces, el problema queda en definir el valor de X. ¿Cuántos posibles valores de X tenemos? No muchos, porque X debe ser la suma entre el mayor número del arreglo, y otro número. Es decir que a lo más tenemos (n-1) valores distintos de X.

Como el valor de n es 1e3, podemos plantear una solución cuadrática en complejidad. Si probamos con todos los valores de X, entonces debemos simular el proceso descrito anteriormente con complejidad lineal o logarítmica. Podemos ayudarnos de las estructuras de la librería STL para ello.

Código

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<int> v;

vector<pair<int, int>> simulate(int X) {
    vector<pair<int, int>> ans, empty;
```

```cpp

    // Usamos una estructura multiset que nos permite guardar los elementos, considerando repetidos
    // en order y agreagrlos/retirarlos en complejidad logaritmica
    // https://www.geeksforgeeks.org/multiset-in-cpp-stl/
    multiset<int, greater<int>> MS(v.begin(), v.end());

    while (!MS.empty()) {
        // Tomamos el valor maximo y lo quitamos del multiset
        int maxi = *MS.begin();
        MS.erase(MS.begin());

        int complement = X - maxi;
        // Si el complemento no existe, no podemos continuar
        if (MS.count(complement) == 0) return empty;

        // Quitamos el complemento
        // Importante notar el uso de .find(), caso contrario eliminaremos todas las ocurrencias
        // en caso haya repetidos
        MS.erase(MS.find(complement));
        ans.push_back(make_pair(maxi, complement));

        // Actualizamos el valor de X
        X = maxi;
    }
    return ans;
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        int N;
        cin >> N;
        // Definimos v como una variable global para poder usarla en una funcion
        // auxiliar
        v.resize(2 * N);
        for (int i = 0; i < 2 * N; i++) cin >> v[i];

        // Si usamos los iteradores rbegin y rend, el ordenamiento sera
        // decreciente
        sort(v.rbegin(), v.rend());
        int X;
        vector<pair<int, int>> solution;
        // Probaremos todas las posible parejas de numeros
        for (int i = 1; i < 2 * N; i++) {
            int initialX = v[0] + v[i];
            solution = simulate(initialX);
            // encontramos una solucion
            if (!solution.empty()) {
                X = initialX;
                break;
            }
        }
        if (!solution.empty()) {
            cout << "YES" << endl;
            cout << X << endl;
            for (auto p : solution) {
                cout << p.first << " " << p.second << endl;
```

```
            }
        } else {
            cout << "NO" << endl;
        }
    }
}
```