Soluciones OIFem II Nivel 2

Entrenamiento 1

Modular exponentiation

Teoría

- Aritmética modular
- Divide y vencerás

Solución

Como explicamos en clase, aprovechamos que $a^b = a^{\frac{b}{2}} \times a^{\frac{b}{2}} = (a^{\frac{b}{2}})^2$

Código

```
int mpow(int a, int b, int m) {
    if (b == 0) return 1;
    if (b % 2 == 0) {
        int t = mpow(a, b/2, m);
        return t*t % m;
    }
    else {
        int t = mpow(a, (b-1)/2, m);
        return t*t % m * a % m;
    }
}

// Piensa por qué este código es equivalente al de arriba!
int mpow(int a, int b, int m) {
    if (b == 0) return 1;
    int t = mpow(a, b/2, m);
    if (b % 2 == 0) return t*t % m;
    return t*t % m * a % m;
}
```

Find first and last position of element in sorted array

Teoría

- Búsqueda binaria

Solución

Como explicamos en clase, hacer una búsqueda binaria para ambos elementos.

Código

```cpp
vector<int> searchRange(vector<int>& nums, int target) {
    int n = nums.size();
    int l = 0;
    int r = n-1;
    int starting = -1;
    while (l <= r) {
        int m = (l+r)/2;
        if (nums[m] >= target) {
            if (nums[m] == target) starting = m;
            r = m-1;
        }
        else {
            l = m+1;
        }
    }

    int ending = -1;
    l = 0;
    r = n-1;
    while (l <= r) {
        int m = (l+r)/2;
        if (nums[m] <= target) {
            if (nums[m] == target) ending = m;
            l = m+1;
        }
        else {
            r = m-1;
        }
    }

    return {starting, ending};
}
```

Ecuación de tercer grado

Teoría

- Búsqueda binaria

Solución

Como explicamos en clase, esta ecuación cumple $f(x+1) > f(x)$ para todo x y por tanto podemos aplicar búsqueda binaria.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    ll a, b, c, d, n;
    while (cin >> a >> b >> c >> d >> n) {
        ll l = 0;
        ll r = n;
        ll ans = -1;
        while (l <= r) {
            ll m = (l+r)/2;
            ll x = m;
            ll f_x = a*x*x*x + b*x*x + c*x;
            if (f_x >= d) {
                if (f_x == d) ans = x;
                r = m-1;
            }
            else {
                l = m+1;
            }
        }
        cout << ans << endl;
    }
}
```

A caballo por el viñedo

Teoría

- Búsqueda binaria

Solución

Como explicamos en clase, podemos aplicar búsqueda binaria consultando las sumas de los subarrays (usando sumas de prefijos por eficiencia).

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
using vi = vector<int>;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, u;
    while (cin >> n >> u) {
        if (n == 0 and u == 0) break;

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

        vi pre(n);
        pre[0] = v[0];
        for (int i = 1; i < n; ++i) {
            pre[i] = pre[i-1] + v[i];
        }


        // suma de v[l]+...+v[r] = pre[r]-pre[l-1]

        int ans = pre[n-1];

        for (int i = 0; i < n; ++i) {

            int l = i;
            int r = n-1;
            while (l <= r) {
                int m = (l+r)/2;
                int suma = pre[m];
                if (i > 0) suma -= pre[i-1];
                if (suma >= u) {
                    ans = min(ans, suma);
                    r = m-1;
                }
                else {
```

```
                    l = m+1;
                }
            }

        }

        if (ans < u)
            cout << "IMPOSIBLE" << endl;
        else
            cout << ans << endl;
    }
}
```

Deadline

Teoría

- Búsqueda ternaria

Solución

Como explicamos en clase, $x + \left\lceil \frac{d}{x} \right\rceil$ es decreciente hasta un punto y luego creciente, por lo que cumple la propiedad necesaria para aplicar búsqueda ternaria.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using ld = long double;
using ll = long long;

ll n, d;

ld f(ld x) {
    return x + ceil(d/(x+1));
}

ll f(ll x) {
    return x + (d+x)/(x+1);
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int t;
    cin >> t;
    while (t--) {
        cin >> n >> d;
        ld l = 1;
        ld r = n;
        for (int i = 0; i < 200; ++i) { // con 200 iteraciones converge de sobras
            ld m1 = l + (r - l)/3;
            ld m2 = r - (r - l)/3;
            if (f(m1) < f(m2)) {
                r = m2;
            }
            else {
                l = m1;
            }
        }
        ll ans = 1e18;
        for (ll x = max(0ll, ll(r-10)); x <= ll(r+10); ++x) { // para evitar problemas de precisión
            ans = min(ans, f(x));
        }
        cout << (ans <= n ? "YES" : "NO") << endl;
```

```
42          }
43      }
```

Sqrt(x)

Teoría

- Búsqueda binaria

Solución

Como $\sqrt{x+1} > \sqrt{x}$, podemos aplicar búsqueda binaria. La condición sería $m \leq \sqrt{x}$, pero como estamos asumiendo que no podemos calcular raíces cuadradas (y por eficiencia), la expresamos de forma equivalente matemáticamente, pero mejor para el programa: $m^2 \leq x$.

Código

```
int mySqrt(int x) {
    int l = 0, r = 46340; // 46340 = raíz de 2^31-1 truncada
    while (l < r) {
        int m = (l+r+1)/2;
        if (m*m <= x) {
            l = m;
        }
        else {
            r = m-1;
        }
    }
    return l;
}
```

Arranging coins

Teoría

- Búsqueda binaria

Solución

El problema es equivalente a encontrar la máxima k tal que $1 + 2 + ... + k \leq n$. Esta suma tiene una fórmula cerrada: $f(k) = 1 + 2 + ... + k = \frac{k(k+1)}{2}$. Como $f(k+1) > f(k)$, podemos aplicar búsqueda binaria.

Código

```
int arrangeCoins(int n) {
    ll l = 0;
    ll r = 1e5;
    while (l < r) {
        ll m = (l+r+1)/2;
        if (m*(m+1)/2 <= n) {
            l = m;
        }
        else {
            r = m-1;
        }
    }
    return l;
}
```

A gas station too far

Teoría

- Búsqueda binaria

Solución

Supongamos que sabemos la capacidad del depósito. Entonces está claro que siempre queremos repostar solo cuando ya no llegaremos a la siguiente gasolinera, es decir, lo más tarde posible, y en tiempo $\mathcal{O}(n)$ podemos ver si llegamos o no.
Además, está claro que si llegamos con x litros, entonces llegamos con $x+1$, $x+2$, etc...
Está claro que no se puede llegar y a partir de ciertos litros siempre se puede, por lo que se puede aplicar búsqueda binaria. Haremos $\mathcal{O}(\log n)$ veces la comprobación de si llegamos con x litros (que cuesta $\mathcal{O}(n)$), por lo que en total hacemos $\mathcal{O}(n \log n)$ operaciones.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;

vi l;
int n, s;

bool funciona(int r) {
        int cnt = 0;
        int gas = r;
        for (int i = 0; i < n; ++i) {
                if (l[i] > r) return false;
                if (gas < l[i]) {
                        gas = r;
                        ++cnt;
                        if (cnt > s) return false;
                }
                gas -= l[i];
        }
        return true;
}

int main() {
        while (cin >> n >> s) {
                l = vi(n);
                for (int i = 0; i < n; ++i) {
                        cin >> l[i];
                }

                int lo = 1;
                int hi = 1e9+1;
                int ans = -1;
                while (lo <= hi) {
                        int mid = (lo+hi)/2;
                        if (funciona(mid)) {
                                ans = mid;
                                hi = mid - 1;

```

```
                        }
                        else {
                                lo = mid + 1;
                        }
                }
                cout << ans << endl;
        }
}
```

The lightest pearl

Solución

La solución óptima es partir en 3 grupos lo más parecidos posibles, por lo que hay siempre entre $\lfloor \frac{p}{3} \rfloor$ y $\lceil \frac{p}{3} \rceil$ en estos grupos. El peor caso siempre es que la que buscamos se encuentra en el mayor grupo, por lo que nos quedamos con el grupo más grande y repetimos recursivamente. A cada paso dividimos p entre 3 hasta llegar a 1, por lo que hacemos $\mathcal{O}(\log_3 p)$ operaciones.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

int f(int p) {
    if (p == 1) return 0;
    return f((p+2)/3) + 1; // (p+2)/3 da ceil(p/3)  (p/3 redondeado hacia arriba)
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int p;
    while (cin >> p) {
        cout << f(p) << endl;
    }
}
```

Find Right Interval

Teoría

- Binary search

Solución

Si ordenamos los intervalos es fácil ver que podemos aplicar búsqueda binaria para encontrar el que nos piden, ya que a partir de ese intervalo todos a la derecha lo cumplen también. Añadimos una tercera componente a los intervalos antes de ordenar para recordar su posición original y así reconstruir ese orden al acabar el proceso ya que tenemos que dar las respuestas en el orden original.

Código

```
vector<int> findRightInterval(vector<vector<int>>& intervals) {
    int n = intervals.size();
    for (int i = 0; i < n; ++i) {
        intervals[i].push_back(i);
    }
    sort(intervals.begin(), intervals.end());
    vector<int> ans(n, -1);
    for (int i = 0; i < n; ++i) {
        int l = 0;
        int r = n-1;
        while (l <= r) {
            int m = (l+r)/2;
            if (intervals[m][0] >= intervals[i][1]) {
                ans[i] = m;
                r = m-1;
            }
            else {
                l = m+1;
            }
        }
    }
    vector<int> ans_original_order(n);
    for (int i = 0; i < n; ++i) {
        ans_original_order[intervals[i][2]] = ans[i] == -1 ? -1 : intervals[ans[i]][2];
    }
    return ans_original_order;
}
```

Gallinas

Teoría

- Búsqueda binaria

Solución

Si asumimos que las gallinas pueden volar una distancia x, entonces podemos calcular cuántas parejas de gallinas amigas existen: Vamos sumando las gallinas en corrales consecutivos separados por distancia $\leq x$, cuando ya no podemos unir el siguiente corral contamos cuántas parejas han añadido estos corrales y seguimos. Si en estos corrales había n gallinas en total, entonces nos aportan $\binom{n}{2} = \frac{n(n-1)}{2}$ parejas. Sabiendo que $(a+b)^2 > a^2+b^2$ para números positivos, vemos que dos grupos juntos aportan más parejas que separados, por lo que podemos aplicar binaria respecto a la distancia x ya que la cantidad de parejas con distancia máxima $x+1$ será mayor o igual que con distancia máxima x.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
using vl = vector<ll>;

ll a;
int n;
vl x;

bool ok(ll dist) {
    ll adj = 1, cnt = 0;
    for (int i = 0; i < n-1; ++i) {
        if (x[i+1] - x[i] <= dist) {
            ++adj;
        }
        else {
            cnt += adj*(adj-1)/2;
            adj = 1;
        }
    }
    cnt += adj*(adj-1)/2;
    return cnt >= a;
}

int main()
{
    ios::sync_with_stdio(0);
    int t;
    cin >> t;
    while (t--) {
        cin >> n >> a;
        x = vl(n);
        for (int i = 0; i < n; ++i) cin >> x[i];
        sort(x.begin(), x.end());
        ll l = 0, r = x.back(), ans = 0;
        while (l <= r) {
            ll m = (l+r)/2;
```

```
            if (ok(m)) {
                ans = m;
                r = m-1;
            }
            else {
                l = m+1;
            }
        }
        cout << ans << endl;
    }
}
```