Apuntes OIFem II Nivel 2

Búsqueda binaria, Divide y vencerás

Búsqueda binaria

Supongamos que queremos encontrar la posición de un elemento x en un vector A que tiene N elementos $(A[0], \ldots, A[N-1])$, o indicar que no existe.

Es fácil ver que el único algoritmo factible (asumiendo que no hemos procesado previamente este vector) es mirar en todas las posiciones en tiempo $O(N)$. Si tu algoritmo no mira alguna posición, un vector que tiene el valor x solo en esa posición haría fallar tu algoritmo.

Muchas veces no sólo sabemos que tenemos un vector de un cierto tamaño, sino que también sabemos que está ordenado. Esta información nos permite buscar el elemento de una forma más inteligente:

Asumimos que está ordenado de forma no decreciente ($A[i] \leq A[i+1]$). Tendremos dos variables L, R que indicarán el intervalo donde podría encontrarse x, incialmente $L = 0$ y $R = N-1$. Repetimos el siguiente proceso mientras $L \leq R$:
- Miramos el elemento de la posición central M $= \lfloor (L+R)/2 \rfloor$.
- Si $A[M] < x$, sabemos que a la izquierda de M aún son más pequeños los elementos, así que podemos descartar los elementos entre L y M, por lo que asignamos $L = M+1$.
- Si $A[M] > x$, de forma similar al caso anterior, descartamos entre M y R, asignando $R = M-1$.
- Si $A[M] = x$, hemos acabado.
Si el proceso sale del bucle sin haber encontrado x podemos garantizar que no existe.

A cada paso nos quedamos con la mitad de elementos, por lo que tras k pasos nos quedan $N/2^k$ elementos. Vemos que si $k > log_2(N)$ entonces el intervalo ya es más pequeño que 1 y por tanto ha finalizado, por lo que se ejecuta en tiempo $O(\log N)$.

```
int bs(const vector<int>& A, int x) {
    int N = A.size(), L = 0, R = N-1;
    while (L <= R) {
        int M = (L+R)/2;
        if (A[M] < x) L = M+1;
        else if (A[M] > x) R = M-1;
        else return M; // A[M] == x
    }
    return -1; // x no está en el vector
}
```

Generalmente las explicaciones sobre búsqueda binaria suelen quedarse aquí, pero es una técnica mucho más potente.

Un ejemplo de problema distinto sería el siguiente: Supongamos que queremos encontrar una x entera que cumpla $ax^3 + bx^2 + cx = d$, y sabemos que los valores de a, b, c y d son enteros positivos.
Si somos capaces de encontrar una propiedad que nos separe los valores en dos lados, podremos hacer búsqueda binaria (esto aplica a todos los problemas en los que funciona la búsqueda binaria).
Un ejemplo es agrupar en un grupo "1" los valores que cumplen $ax^3 + bx^2 + cx \geq d$ y en un grupo "0" los que no.

Por ejemplo, imaginemos que queremos resolver $2x^3 + 3x^2 + 4x = 861$.

$$\begin{aligned} x: &\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9, 10, 11 \ldots \\ \text{cumple la condición}: &\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 1,\ 1 \ldots \end{aligned}$$

Tenemos que haber elegido la condición de forma inteligente, de manera que la solución al problema, si existe, se encuentre en los elementos de la frontera entre los grupos 0 y 1.
La idea será que si caemos en un elemento de 0 saltamos a la derecha y si caemos en un elemento de 1 saltamos a la izquierda, lo que inevitablemente nos llevará a ese punto en la frontera entre 0s y 1s (y de forma eficiente!). Este salto se traduce en acotar el intervalo que mantenemos con las variables L y R como en la búsqueda binaria estándard.

Problema: https://jutge.org/problems/P78497_es/statement

```
ll a, b, c, d, n;
while (cin >> a >> b >> c >> d >> n) {
    ll l = 0, r = n, ans = -1; // el enunciado nos asegura que si hay solucion es < n
    while (l <= r) {
        ll x = (l+r)/2; // x hace de m
        ll f_x = a*x*x*x + b*x*x + c*x;
        if (f_x >= d) {
            if (f_x == d) ans = x;
            r = x-1;
        }
        else {
            l = x+1;
        }
    }
    cout << ans << endl;
}
```

Otro ejemplo de problema sería: Dado un vector con valores positivos, encontrar el subvector (consecutivo) de suma mínima que llegue a una cierta constante u.
Es decir, queremos encontrar i y j tal que $A[i] + \cdots + A[j] \geq u$ pero que se pase lo mínimo posible.

El primer algoritmo que se nos puede ocurrir es mirar desde cada posición i todos los subvectores hasta otras posiciones $j > i$. Esto es un doble for hasta N, se ejecutaría en tiempo $O(N^2)$. Podemos hacer algo parecido, pero añadiendo el ingrediente de búsqueda binaria:
Fijamos el extremo izquierdo del subvector i, y fijémonos que la suma de i a j es creciente, ya que los elementos del vector son positivos. Podemos hacer una búsqueda binaria sobre j para cada i. Pero necesitamos poder calcular la suma $A[i] + \cdots + A[j]$ sin recorrer los elementos.
Para eso definimos un nuevo vector de sumas de prefijos, P, tal que $P[i] = A[0] + \cdots + A[i]$.
Así tendremos que $A[i] + \cdots + A[j] = P[j] - P[i-1]$, y podemos consultar la suma de i a j en tiempo constante. Con todo esto, hacemos N búsquedas binarias, por lo que es un algoritmo $O(N \log N)$.

Problema: https://www.aceptaelreto.com/problem/statement.php?id=590

```cpp
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); // suma de v[l]+...+v[r] = pre[r]-pre[l-1]
    pre[0] = v[0];
    for (int i = 1; i < n; ++i) {
        pre[i] = pre[i-1] + v[i];
    }
    int ans = pre[n-1]; // una primera posible solución es coger todo
    for (int i = 0; i < n; ++i) {
        int l = i, 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); // tratamos de minimizar la solución
                r = m-1;
            }
            else {
                l = m+1;
            }
        }
    }
    if (ans < u)
        cout << "IMPOSIBLE" << endl;
    else
        cout << ans << endl;
}
```

Divide y vencerás

Los algoritmos divide y vencerás se basan en la idea de que al partir un problema en problemas más pequeños puede hacer que resolver el problema original sea más fácil a partir de las soluciones de estos problemas pequeños.

Un ejemplo básico de esta idea consiste en calcular $A^B \mod M$.
Cuando B es par, $A^B \mod M = (A^{B/2} \mod M) \times (A^{B/2} \mod M) \mod M$.
Si B es impar, $A^B \mod M = (A^{(B-1)/2} \mod M) \times (A^{(B-1)/2} \mod M) \times A \mod M$. En ambos casos partimos el problema en problemas pequeños, pero esos problemas pequeños son dos veces el mismo problema! Así que reutilizamos la solución de uno simplemente elevando al cuadrado y tenemos un algoritmo eficiente.

Problema: https://jutge.org/problems/P29212_en/statement

```
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;
}
```

Un ejemplo bastante conocido y aplicable de divide y vencerás es el algoritmo de Karatsuba, un método eficiente para multiplicar números de N cifras.

Paréntesis histórico: El matemático Kolmogorov organizó un seminario donde planteó que no se podría encontrar algoritmos que multiplicaran en menos de $O(N^2)$ operaciones. Invitó a su estudiante, Karatsuba, que encontró un algoritmo $O(N^{\log_2 3}) \approx O(N^{1.58})$ en una semana y Kolmogorov lo hizo público el último día de su seminario. (https://es.wikipedia.org/wiki/Algoritmo_de_Karatsuba#Historia)

Vamos a expresar nuestra multiplicación como números separados por la mitad en dos partes. Supongamos que son números de $2N$ cifras por simplificar las expresiones.

Queremos calcular $R = X \times Y = \underbrace{(10^N A + B)}_{X} \times \underbrace{(10^N C + D)}_{Y}$. Desarollando el producto,

$$R = 10^{2N} AC + 10^N (AD + BC) + BD$$

Los productos AC, AD, BC y BD los calculamos aplicando el algoritmo recursivamente.
Si la complejidad de nuestro algoritmo es $T(N)$, cumple:

$$T(N) = 4T(N/2) + O(N)$$

Ya que aplicamos el algoritmo recursivamente 4 veces y hacemos sumas en tiempo lineal. Si resolvemos esta recurrencia obtenemos $T(N) = O(N^2)$:(

No ha servido de nada partir en problemas pequeños? La respuesta es que no solo importa partir en problemas pequeños, sino hacerlo de forma eficiente, aprovechando al máximo los cálculos.

Karatsuba se fijó en lo siguiente: él quería $10^{2N}AC + 10^N(AD + BC) + BD$, que está en función de 4 términos, pero y si pudiera calcular $AD + BC$ directamente, sin multiplicar por separado dos veces y luego sumar?

Así que primero calcula AC y BD recursivamente. Luego plantea el producto $(A + B)(C + D) = AC + AD + BC + BD$. Resulta que $(A + B)(C + D) - AC - BD = AD + BC$, el término que buscaba, y puede obtenerlo ya que previamente ha calculado AC y BD.

Solo ha empleado una tercera multiplicación en lugar de llegar a hacer 4 multiplicaciones recursivas, por lo que la recurrencia del tiempo empleado es ahora

$$T(N) = 3T(N/2) + O(N) = O(N^{\log_2 3}) \approx O(N^{1.58})$$