Olimpiada Peruana de Informática Fase 2 - Solucionario

Federación Olímpica Peruana de Informática

26 de febrero de 2023

Tutorial del problema: "Inserción Laboral"

Autor(es): Miguel Miní

Grupo 1 (Complejidad Esperada: $O(n2^n)$)

Para el primer subtask, podemos hallar el óptimo con programación dinámica, eligiendo el orden correcto cada vez, lo que tardaría $O(\sum_{i=1}^{n} i \times 2^i) = O(n2^n)$.

Grupo 2 (Complejidad Esperada: $O(n^2 \log n)/O(n^2)$)

Para mimimizar la distancia, suponiendo que ambas secuencias esten ordenadas, viene dada por $\max(x_i + y_{n-i+1})$. basta notar que para $i < j$ y $l < r$:

$$\max(x_i + y_r, x_j + y_l) \leq x_j + y_r \leq \max(x_i + y_l, x_j + y_r)$$

Lo que nos deja una complejidad de $O(n^2 \log n)/O(n^2)$ dependiendo de la implementación.

Grupo 3 (Complejidad Esperada: $100n + O(n)$)

Usando lo anterior, notemos que podemos ordenarlo por *counting sort*, y tener una liste de 100 valores, además de agregar un nuevo elemento al orden en $O(1)$. Si trabajamos con estos arrays ordenados, podemos tener una complejidad de $100n + O(n)$.

Tutorial del problema: "Majora's Mask"

Autor(es): Miguel Miní

En primer lugar, el problema se puede reformular como: Dado un árbol con n nodos, hallar la mayor distancia luego de retirar y volver a colocar una arista, de tal forma que siga siendo un árbol.

Grupo 1 (Complejidad Esperada: $O(n^4)$)

Pordemos retirar una arista del arbol y volver a colocar otra en $O(n^3)$, luego calcular el diametro en $O(n)$. Lo que nos deja una complejidad $O(n^4)$.

Grupo 1 (Complejidad Esperada: $O(n \log n)/O(n)$)

Notemos que la respuesta siempre es el diámetro del árbol que se forma. Si no retiramos aristas, la respuesta es el diámetro inicial, así que si queremos colocar una arista, queremos que el diámetro pase por ahí. Por eso mismo, se deben unir los árboles por sus diámetros. Así, podemos resolverlo en dos partes:

- rooteamos el árbol en uno de sus diámetros, por cada subárbol que sale del diámetro, calculamos con dp su diámetro. Entonces, la respuesta es la suma de ese diámetro y el diámetro inicial.
- para la segunda parte, debemos retirar una arista del diámetro, por cada nodo del diámetro calculamos la máxima distancia fuera del diámetro, y el diámetro respectivo en ese subárbol, luego la respuesta puede ser calculada por un segment tree para los rangos $[1, i]$ y $[i + 1, n]$ del diámetro.
- (bonus) Si rooteamos en ambos extremos del diámetro, podemos calcular la respuesta en $O(n)$.

Grupo 3 (Complejidad Esperada: $O(n)$)

Podemos resolver ambos casos, haciendo rerooting, donde para cada nodo debemos calcular el diámetro dentro del subárbol y el resto, para ello, para un nodo hijo debemos unir el diámetro de afuera con los vecinos de dicho hijo. La técnica es conocida como in/out.

```
//dp para cada subárbol
S dfs(int x, int p) {
    dp[x] = S(0, 0);
    for (Node& e : g[x]) {
        if (e.u == p) {
            swap(e, g[x].back());
            if (e.u == p) continue;
        }
        S dp_u = dfs(e.u, x);
        dp_u.add_edge();
        dp[x] = dp[x] + dp_u;
    }
    if (p != 0) g[x].pop_back();
    return dp[x];
}

void dfs_reroot(int x, int p, S out) {
    vector<S> suf; //calculamos el sufijo de los nodos hermanos
    for (Node e : g[x]) {
        S dp_u = dp[e.u];
        dp_u.add_edge();
        suf.push_back(dp_u);
    }
    suf.push_back(S(0, 0));
    for (int i=(int)suf.size()-2; i>=0; --i) {
        suf[i] = suf[i] + suf[i+1];
    }
    S pref(0, 0); //calculamos el prefijo
    for (int i=0; i<g[x].size(); ++i) {
        auto [u, id] = g[x][i];
        S new_out = out + (pref + suf[i+1]);  //unimos los diametros externos
        S dp_u = dp[u];
        //calculamos la respuesta
        ans[id] = new_out.diam + 1 + dp_u.diam;
        new_out.add_edge();
        dfs_reroot(u, x, new_out); //enviamos el diametro externo
        dp_u.add_edge();
        pref = pref + dp_u;
    }
}
```

Tutorial del problema: "Candy Crush"

Autor(es): Miguel Miní

Grupo 1 (Complejidad Esperada: $O(n!)$)

Podemos simular el proceso de selección, donde la complejidad viene dada por:

$$T(n) \leq nT(n-1)$$

Lo que nos da una complejidad $O(n!)$.

Grupo 2 (Complejidad Esperada: $O(n^4)$)

Podemos plantear un dp en rango, donde llevemos en cada estado una recolección parcial de elementos que queremos tomar, y luego decidir un bloque que se quiere saltar (como una parentización). $O(n^4)$

$$dp(l, r, k) = dp(l + 1, r, 0) + 1$$
$$dp(l, r, sum) = dp(l + 1, i) + dp(i + 1, r, sum + 1)$$

Grupo 3 (Complejidad Esperada: $O\left(n^3k\right)$)

Para mejorar la idea anterior, podemos restringir la suma a k, con esto tendremos un $O(n^3k)$

Grupo 4 (Complejidad Esperada: $O\left(n^3 + n^2k\right)$)

(Bonus) De forma óptima, solo partimos el bloque si es que necesitamos agregar 1 a nuestra suma. Por tanto, los estados donde la suma tiene valor son cuando el extremo l contiene los elementos que conforman la suma. con lo que nos queda $O(n^3 + n^2k)$. Si además, comprimimos el array inicial logramos un $O(\frac{n^3}{c} + n^2k)$. Donde c es la cantidad de colores distintos.

Tutorial del problema: "Dijkstra Mul"

Autor(es): Miguel Miní

Primero notemos que la solución siempre es un camino simple, en otro caso podemos quitar un ciclo y así reducir alguna de las sumas, mejorando la respuesta.

Grupo 1 (Complejidad Esperada: $O(n^2)/O(n + m \log n)$)

En este caso $x_i = y_i$, por tanto, el problema es un simple Dijkstra.

Grupo 2 (Complejidad Esperada: $O(\max(t)nm \log n)$)

Actualizando por cada vez que encontramos una relajación, debemos, en una cantidad máxima de 2000 pasos hallar cada camino.

Grupo 3 (Complejidad Esperada: $O((n \max(t))^{\frac{2}{3}} m \log n)$)

Notemos que si fijamos los nodos como $(v_i, \sum t_j)$ y corriendo un Dijkstra normal, debemos obtener $O(nm \max t \log m)$ ya que $\sum t_j \leq n \max t$. Por otro lado, si optimizamos por t_i y consideramos el orden natural para los pares (t, c), de tal forma que para un t mayor siempre minimizamos c Nuestra solución será: $O((n \max t)^{\frac{2}{3}} m \log n)$ con una constante dependiendo de la implementación.

```cpp
vll dikjstra(int s) {
    priority_queue<iii, vector<iii>, greater<iii>> Q;
    vll res(n+1, LLONG_MAX);
    vi tim(n+1, INT_MAX);
    Q.push({{0, 0}, s});
    while (!Q.empty()) {
        auto q = Q.top(); Q.pop();
        int v = q.second;
        int t = q.first.second, c = q.first.first;
        res[v] = min(res[v], t*1ll*c);
        //vamos tomando los nodos del convex hull.
        if (tim[v] < t) continue;
        tim[v] = t;
        trav(e, g[v]) {
            Q.push({{c+e.c, t+e.t}, e.to});
        }
    }
}
```

Grupo 4 (Complejidad Esperada: $O((n \max(t))^{\frac{2}{3}} m \log n)$)

La solución anterior, también se puede aplicar en este caso, ocurre que hay muchos cálculos que se sobrestiman, por otro lado, para poder explicar porque sucede esta complejidad, primero debemos notar que para cualquier segmento desde (a, b) hasta (x, y), el mínimo siempre está en un extremo:

$$(ta + (1-t)x)(tb + (1-t)y) =$$

$$t^2ab + t(1-t)(ay + bx) + (1-t)^2xy \geq$$

$$t^2ab + 2t(1-t)\sqrt{aybx} + (1-t)^2xy =$$

$$(t\sqrt{ab} + (1-t)\sqrt{xy})^2$$

Sin perdida de generalidad, sea $xy \leq ab$, por tanto:

$$(t\sqrt{ab} + (1-t)\sqrt{xy})^2 \geq$$

$$(t\sqrt{xy} + (1-t)\sqrt{xy})^2 = xy$$

De esta forma, uno puede probar que para un polígono convexo, la solución se encuentra en un vértice extremo. Note que con el algoritmo descrito en 3. nosotros estamos recorriendo la cápsula convexa (quizás con algunos puntos extras). Finalmente, en un rectángulo $[1, N]^2$ cualquier polígono con coordenadas enteras puede tener a lo más $O(N^{\frac{2}{3}})$ puntos. Luego, como detalle, existe otra forma de recorrer todos los puntos de la cápsula convexa, de una manera más certera, pero no se abordará en esta editorial.