examenfinal2015-06aunque-diga-en...

Anónimo

Algoritmos y Estructuras de Datos II

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de Murcia

RESOLUCIÓN EXAMEN AED2 JUNIO 2015

NOTA: en rojo correcciones desde la 1ª vez que subí la 1ª vez.

(1) *Mejor caso: people[index] ≤ 1 ∀ index, no entra al mientras.

$$t_m(n) = \begin{cases} 1+1+1, & n<4 \\ 3+3+t_m\left(\frac{n}{4}\right)+5+2\,t_m\left(\frac{n}{2}\right), & n\geq 4. \end{cases}$$

$n = 2^k$, $k = \log_2 n$., $t'_m(k) = t_m(2^k)$.

$t'_m(k) - 2\,t'_m(k-1) - t'_m(k-2) = 13$

EC. CAR: $(x^2-2x-1)\underbrace{(x-1)} = 0$, $x = \frac{2 \pm \sqrt{4+4}}{2} = 1 \pm \sqrt{2}$, $x = 1$.

SOL. GEN (en k): $t'_m(k) = c_1 + c_2(1+\sqrt{2})^k + c_3(1-\sqrt{2})^k$.

SOL. GEN (en n): $t_m(n) = c_1 + c_2(1+\sqrt{2})^{\log_2 n} + c_3(1-\sqrt{2})^{\log_2 n}$.

$\hookrightarrow t_m(n) \in \Theta\left(n^{\log_2(1+\sqrt{2})} \mid n = 2^k\right)$

CASOS BASE: 3, por ejemplo: n = 1, 2, 4.

* Peor caso: people[index] > 1 ∀ index, siempre entramos al mientras.

$$t_M(n) = \begin{cases} 2+4.\log_2 n. \\ 2+3\log_2 n + 10 + 2\,t_M\left(\frac{n}{2}\right) + t_M\left(\frac{n}{4}\right). \end{cases}$$

$1 + \sum_{i=1}^{\log_2 n - 1} 3. =$

$= 1 + 3\log_2 n - 3$

$= 3\log_2 n. - 2$

$n = 2^k$, $k = \log_2 n$, $t'_M(k) = t_M(2^k)$.

$t'_M(k) - 2\,t'_M\left(\frac{n}{2}\right) - t'_M\left(\frac{n}{4}\right) = 10 + 3.k.$

EC. CAR: $(x^2-2x-1)(x-1)^2 = 0$, $x = \begin{cases} 1+\sqrt{2} \\ 1-\sqrt{2} \\ 1, \text{ 2 veces} \end{cases}$

SOL. GEN (en k): $t'_M(k) = c_1 + c_2 k + c_3(1+\sqrt{2})^k + c_4.(1-\sqrt{2})^k$.

SOL. GEN (en n): $t_M(n) = c_1 + c_2.\log_2 n + c_3.n^{\log_2(1+\sqrt{2})} + c_4.(1-\sqrt{2})^{\log_2 n}$.

CASOS BASE: 4, n = 1, 2, 4, 8

$\hookrightarrow t_n(n) \in \Theta\left(n^{\log_2(1+\sqrt{2})} \mid n = 2^k\right)$ $\hookrightarrow := a.$

SÍ EXISTE ORDEN EXACTO EN TODOS LOS CASOS. $t(n) \in$

(2) Es necesario un array A donde para cada tipo de bombón se va contabilizando su nº de apariciones.
Al final, caja aburrida sii ∃ tipo i / A[i] > n/2.

Por eficiencia consideraré A global: A[1..n] int.

Llamo C al array caja, tal que C[i] indica el tipo del bombón iésimo.

```
bool llamada inicial ()  // asumo A, C y n globales.
  int i;                 // A inicializado a ∅.
  DyV(1, n);
  PARA i = 1 .. n
    SI A[i] > n/2 retorna TRUE.
  FIN PARA.
  retorna FALSE.
```

```
DyV ( int: i, j )  // i...j = rango del subproblema.
  int k;
  SI i == j        // pequeño.
    A[C[i]]++      // solución directa.
  SINO.            // dividir.
    k = (i+j)/2 ;
    DyV(i, k);     ) // llamada recursiva
    DyV(k+1, j);   )
  // no hay combinar.
```

---

(3) En este problema la recurrencia necesita de cuatro variables: presupuesto disponible (p), nº de jugadores por completar (j), nº de jugadores disponibles (u), y un booleano que indique si yo se lo he elegido portero.

Se puede simplificar usando las tres primeras y dos funciones diferentes { alineación ≡ A, cuando aún no elegido portero, y será el primer jugador a elegir; A(p, up), up = nº porteros disponibles
alineación no portero ≡ ANP, cuando ya elegido portero. → ANP(p, j, u).

Inicialmente: p = P, j = 4, u = N = nº jugadores disponibles
up = Np = nº porteros disponibles

* A(p, up) no es recursiva, sino que llama a ANP...

$$A(p, up) = \max_{i=1..up} \left(ANP(p - Pp[i], 4, N) \right).$$

NOTA: asumo datos previo y valoración separados para porteros en arrays Pp y Vp.

* ANP(p, j, u) sí es recursiva...

- decisión, coger o no jugador u-ésimo:

$$ANP(p, j, u) = MAX \begin{cases} \text{NO COJO } j: \\ \quad ANP(p, j, u-1) \\ \text{SÍ COJO } j: \\ \quad ANP(p - V[j], j-1, u-1) + V[j]. \end{cases}$$

- CASOS BASE: { j = 0 → 0 } si no hay ... puestos, ∅.
{ IMPOSIBLE (−∞): { j < 0, u < 0, p < 0 → −∞ (fuera de tabla)
j > 0 y u = 0 → no jugadores para cubrir
j > 0 y p = 0 → no dinero ... }

T: TABLA 3D { p: 0 — P
j: 0 — 4.
u: 0 — N.

RELLENAR:

// casos base + imposibles.

```
PARA p = 0 .. P.
   PARA u = 0 .. N
      T[p, 0, u] = 0
   FIN PARA
FIN PARA.
```

// j = 0.

```
PARA j = 1 .. 4
   PARA p = 1 .. P
      T[p, j, 0] = -∞   // u = 0, j > 0, p > 0.
   FIN PARA.
   PARA u = 1 .. N.
      T[0, j, u] = -∞   // p = 0, j > 0, u > 0
   FIN PARA.
   T[0, j, 0] = -∞
FINPARA.
```

// RECURRENCIA...

```
PARA p = 1 .. P
   PARA j = 1 .. 4.
      PARA u = 1 .. N.
         T[p, j, u] = max( T[p, j, u-1],
                           T[p-V[j], j-1, u-1] + V[j] )
```

RECONSTRUIR SOLUCIÓN

* La i de la llamada a $A(P, N_p)$ nos da el portero. La solución para ANP está en $T[p - Pp[i], 4, N]$. (el valor de la solución).

* Para reconstruir la tupla ~~de~~ solución (los 4 jugadores elegidos), ir hacia atrás en la recurrencia, viendo cuál de los dos argumentos de MAX se aplicó, coger/no coger. Cuando se coge un jugador, añadirlo a la tupla solución

Descansito &Coins

GARNIER

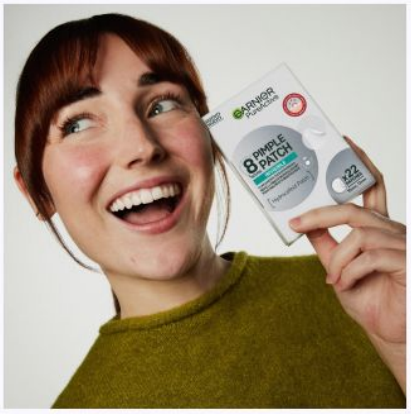
Descubre la solución para el enigma y para los granos con este juego

¿Granito inesperado? No te rayes, ponte un patch

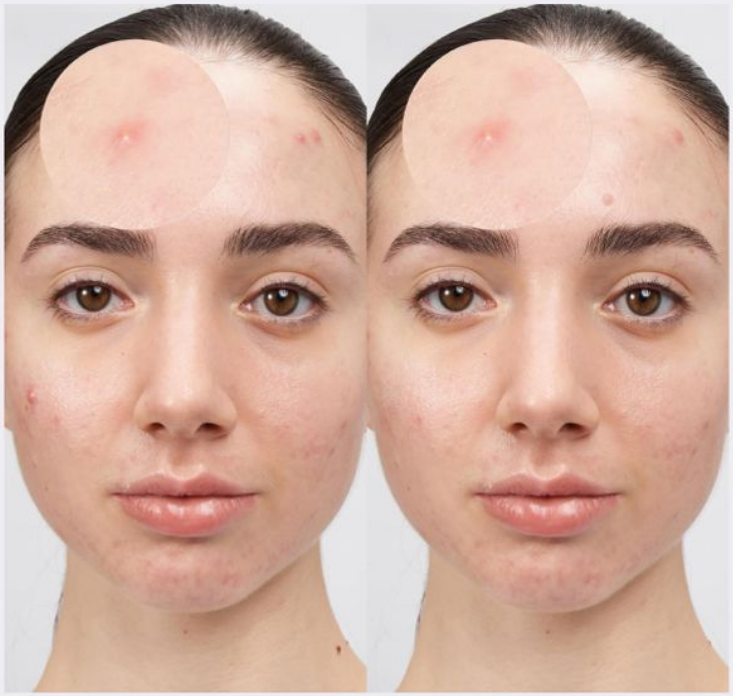
REGLAS

1. Atento a las pistas.
2. Encuentra las diferencias.
3. Gana Wuolah Coins y úsalas para descargar sin publi 🎉

Fácil | 5



JUGAR



Encuentra las diferencias y márcalas, ponle tu propio Pimple Patch.

BLABLAGRAPH

Para los tres algoritmos, la tupla solución es un vector de n-1 posiciones; cada una es un nº que indica la ciudad de la etapa correspondiente que fue elegida. Se puede pensar en Murcia y Berlín como 2 posiciones más, de valor fijo.

(4) VORAZ.

La función selección elegirá la ciudad alcanzable desde la actual que dé mayor beneficio en el siguiente paso, excepto en la etapa n-2, donde se tendrán en cuenta 2 pasos, siendo el 2º no elegible. Factible no existe, una vez elegida una ciudad (alcanzable) siempre se puede llegar a ella.

```
Vorazt (n) → int. SOL [0..n]
  (i,j) = (0,1); // ciudad actual.
  sol[0] = 0; // Murcia
  MIENTRAS i ≠ n   // NO SOLUCIÓN.
      k = SELECCIONAR (i, j);
      // NO HAY FACTIBLE.
      SOL[i+1] = k;   // INSERTAR.
      i = i+1; j = k; // actualizar ciudad actual.
  FIN MIENTRAS.
  sol[n+1] = 1; // Berlín.
```

Usaré una función C(i,j,k,l) que me devuelve el nº posej. desde c[i,j] a c[k,l], -∞ si no hay conexión

```
SELECCIONAR (i, j)
  int l, M; ind;
  M = -∞;
  SI i < n-1 // aún no última elección.
      PARA l = 1..n.
          SI (C(i,j,i+1,l) > M
              ind = l; M = C(i,j,i+1,l);
          FINSI
      FIN PARA.
  SINO
      PARA l = 1..n.
          SI C(i,j,i+1,l) + ((i+1,l,i+2,1) > M
              ind = l; M = ⇩
  FINSI, FINPARA, FINSI
```

→ retaem ind;

(5) BACKTRACKING

→ Porrijo del original Berlín para simplificar.
tupla solución: S[0, n-1], enteros.
(árbol n-1 ario). Soluciones en hojas.

ESQUEMA: (MAXIMIZACIÓN)

nivel = 1;
soa = ∅; voa = -∞; s = SINICIAL = todo a ∅.; s[0] = 1; //Marca.
pact = 0; // nº pasajeros actual.

REPETIR.
  GENERAR (nivel, s, pact);
  SI SOLUCIÓN (nivel, s) Y pact > voa.
    soa = s;
    voa = pact;
  SINO // como SOLS en hojas, uso SINO.
    SI CRITERIO (nivel, s)
      nivel = nivel + 1
    SINO MIENTRAS (NO MASHIJOSMANOS (nivel, s) Y
                   nivel > 0).
      RETROCESOR (nivel, s).
HASTA nivel > 0
S[n] = 1; // Berlín

FUNCIONES:

* GENERAR (nivel, s, pact)
  s[nivel]++
  → pact = pact + C(nivel-1, s[nivel-1], nivel, s[nivel]);
  SI nivel > 1,
    pact = pact - C(nivel-1, s[nivel-1], nivel, s[nivel]-1);

* SOLUCIÓN (nivel, s)

    return nivel = n-1;

* CRITERIO (nivel, s)

    return. nivel < n-1;

* MASHERMANOS (nivel, s)

    SI s[nivel] = n return FALSO.

    PARA i = s[nivel] .. n.

        SI C(nivel-1, s[nivel-1], nivel, i) > -∞

            return TRUE.

    FIN PARA.

    return FALSO.

* RETROCESO (s, nivel, pact)

    pact := pact - C(nivel-1, s[nivel-1], nivel, s[nivel])

    s[nivel] = 0;

    nivel--;

---

* POSIBLES MEJORAS:

Podemos podar por criterio de optimización.

Posibles cotas: pact + { 3 * (n - nivel) // t. constante.
                       { voraz, // t. lineal.
                       { (q en cada nivel use máximo entre ciudades de ese nivel.

Añadir esto a criterio y añadir criterio 2.

en la condición del mientras, podar si esta cota no supera a voa.

(6) Ramificación y Poda.

Misma tupla solución que en BT.

Nodo { Tupla = todo ceros
Nivel. = 0
Pact = 0
CI, BE, CS.

Dado nodo x, para obtener su descendencia:

```
PARA i = 1 .. n.
    y.nivel = x.nivel + 1;
    y.tupla = x.tupla;
    y.tupla[y.nivel] = i;
    y.pact[y.nivel] += C(y.nivel-1, y.tupla[nivel-1],
                         y.nivel, y.tupla[nivel]);
    y.CI = ... ; y.BE = .. ; y.CS = ... ; // invocar funciones calculo
```

Un nodo X es solución si x.nivel = n-1 (Berlín se añade al final).

¿COTAS / BE:

CI: trivial, pact, t. constante.
+ elaborada: voraz de ejercicio 4, t. lineal.

CS: trivial: 3 * (n - x.nivel), t. constante.
+ elaborada: voraz usado en BT.

BE: $\frac{CI + CS}{2}$.

* Estrategia de ramificación: MB - LIFO
  → Favorecer profundidad porque soluciones están en hojas.
* Estrategia de poda: (maximización).
  C = max(soluciones, CI(nodos))
  podar y si y.CS < C.
  └ Todos tienen solución.

* ESQUEMA:

```
RyP(n: int,          vars: Nodo).
  LNV := {raiz}; // Pancia.
  C = raiz.CI;
  MIENTRAS LNV ≠ ∅
    x := SELECCIONAR(LNV); // MB-LIFO.
    SI x.CS ≥ C
      ∀y hijo de x
        · SI SOLUCIÓN(y) Y (y.pact > s.pact)
            s = y
            C = max(C, y.pact)
          SINO SI NO SOLUCIÓN(y) Y (y.CS ≥ C)
            LNV = LNV + {y}
            C = max(C, y.CI).
          FINSI.
      FIN ∀
  FIN MIENTRAS.
```