Solucions de la Final

Olimpíada Informàtica Catalana 2019

Problema 1: Endevineu les edats

Sigui x l'edat de la Marta, y la de l'Anna i z la de la Ivet. Tenim que y, z, $y+24$ i $z+24$ són primers. També sabem que $y-3=2(z-3)$. Finalment, $x=4(y+z)$, amb $x<100$. És evident que el repte del problema esdevé trobar les edats de les dues filles, ja que l'última equació ens dóna trivialment l'edat de la Marta sabent y i z.

Podem resoldre el problema fent un programa que calculi els nombres primers i que per a cada parella comprovi si satisfan les condicions de y i z. Tot i això, donat que els nombres són molt petits és possiblement més ràpid probar valors raonant una mica.

Com $x<100$, tenim que $y+z<25$. De la primera equació obtenim que $y=2z-3$ i per tant $3z-3<25$, el que implica que $z\leq 9$. Tenim doncs pocs primers vàlids per z: 2, 3, 5 i 7. Com $z+24$ també ha de ser primer, els dos únics valors restants per z són 5 i 7. La seva relació amb y i el fet que $y+24$ és primer ens marca que z ha de ser 5, y ha de ser 7 i x 48.

Problema 2: Primers de Fermat

Per resoldre aquest problema cal fer un programa que iteri sobre els possibles candidats (fins a passar-se de 10^{18}) i miri si són primers i en cas afirmatiu els multipliqui. Podeu trobar un codi que fa això a `pb2.py`.

Per aquelles persones amants de Bash, podem fer servir `factor`, que factoritza els nombres donats, per resoldre el problema en una línia.

```
for i in {1..64}; do echo "2^$i + 1" | bc | factor | sed -E -n "s/(.*): \1$/\1/p" ; done | paste -sd* | bc
```

Problema 3: Nombres taxicab

La solució esperada consisteix a iterar sobre tots els parells $\{i,j\}$ tals que $i^3+j^3<10^6$ i sumar una descomposició al nombre resultant. Com que els nombres són petits, podem guardar-ho en un vector o diccionari/map. A `pb3.cpp` teniu un codi que fa exactament això.

Problema 4: Caixes senars

La clau per aquest problema és veure primer que els números no són importants, només la seva paritat. Per tant podem considerar que cada posició val 0 o 1 i fer les sumes mòdul 2. A més, cal veure que fixada la base, la resta de posicions venen determinades. Per tant, podem iterar sobre totes les possibles bases (2^{25}) i comptar quants nombres senars apareixen.

Aquest tipus de codi resulta molt fàcil de fer si es treballa en binari fent servir màscares de bits, tal i com es fa a `pb4.cpp`.

Problema 5: Màxim nombre de combinacions

Per resoldre aquest problema cal programar un backtracking que per a cada moneda (tenint en compte que de cada tipus en tenim dues) provi a agafar-la i a no fer-ho, per acabar generant tots els subconjunts de monedes possibles i calcular la seva suma.

Com que el nombre de monedes diferents és 16, el nombre de total de subconjunts és 2^{16}, que és prou petit com per simplement generar-los tots tal i com es fa a `pb5.cpp`.

Problema 6: Semàfors

Per resoldre aquest problema només cal pintar un semàfor amb els colors i mides indicats a l'enunciat. Cal anar amb compte amb les coordenades a l'hora de dibuixar les diferents figures. Podeu consultar la solució oficial a `pb6.py`.

Problema 7: Olles

En aquest problema ens demanen pintar una volta de la trajectòria que se'ns descriu a l'enunciat. Donat que les voltes depenen només de v_1, podem anar iterant en t fins que $t \cdot v_1 > 360$, tal i com detalla l'enunciat. Per a cada punt cal aplicar la fórmula tal i com es descriu per evitar problemes de precisió. Podeu trobar una solució a `pb7.py`.

Problema 8: Pac-man

Aquest problema té diverses parts que hem de considerar per poder fer un codi que funcioni en tots els casos. En primer lloc, cal implementar una funció `Rainbow` tal i com es descriu a l'enunciat, amb cura de no fer errors en el tractament de casos.

En segon lloc, per poder pintar el color de cada posició necessitem saber calcular la distància tal i com es defineix a l'enunciat. Se'ns presenta un món toroïdal, on els costats estan connectats dos a dos. Això implica que sempre que volem anar a un cert punt, podem fer-ho tant per la dreta com per l'esquerra (i tant per dalt com per baix, equivalentment). Així doncs, com que el Pac-man es troba a (x, y), la distància al punt $P(a, b)$ és

$$d(P) = \min(|x - a|, n - |x - a|) + \min(|y - b|, m - |y - b|)$$

Per últim, cal recordar pintar el propi Pac-man de la forma que se'ns indica. Teniu la solució oficial a `pb8.py`.

Problema 9: Mastermind

Conceptualment aquest problema no té massa complicacions, però és delicat de programar i és fàcil complicar-se si no es pensa una mica primer.

El primer que ens diu l'entrada és la mida de la imatge en funció del nombre de línies que ens donen. Cada línia esdevé llavors un problema diferent, on la part de l'esquerra ens la donen directament per pintar amb l'entrada i els cercles blancs i negres els hem de calcular. Per fer-ho, considerem l'algorisme següent per a cada color de la conjectura:

- Si el color coincideix amb el que va en aquella posició, llavors cal afegir un cercle negre.

- Si el color no coincideix, iterem sobre la resposta correcta i mirem si apareix en alguna altra posició on no haguem encertat el color. En aquest cas afegim un cercle blanc.
- Altrament, no s'afegeix res.

Podeu trobar els detalls d'una possible implementació a `pb9.py`.

Problema 10: Pobles d'un país

En aquest problema cal identificar els components d'un graf del qual se'ns donen explícitament els nodes i ens diuen que dos nodes adjacents estan connectats.

Com que cada component s'ha de pintar d'un color determinat, el més senzill és iterar sobre els nodes en l'ordre en que es determinen els colors: primer el node que està més amunt i en cas d'empat, més a l'esquerra. Cada vegada que trobem un node per al qual no sabem el seu component, fem un DFS per trobar tots els nodes del component, assignant-los el color que els pertoqui.

Per pintar les arestes, podem modificar el DFS lleugerament per considerar-les totes (sense repetir els nodes, però) o bé simplement pintar-les totes després, com es fa a la solució oficial que podeu trobar a `pb10.py`.

Problema 11: Taula de multiplicar

En aquest problema cal imprimir les taules de multiplicar de l'1 al 10 segons l'entrada, però escrivint els nombres amb lletres. La dificultat del problema es troba doncs en fer una funció que converteixi un nombre a lletres en català. Potser la manera més senzilla consisteix a partir el nombre en desenes i unitats i, un cop tractats els casos especials, imprimir les desenes i les unitats per separat. Trobareu a `pb11.py` i `pb11.cpp` solucions per aquest problema.

Problema 12: Àlbum de cromos (2)

Si considerem les posicions ordenades dels cromos, es pot veure que la solució òptima consisteix a enganxar des del principi un cert nombre de cromos en ordre (possiblement zero) i enganxar la resta comptant des del final. Podem doncs iterar sobre la posició a la qual dividim els cromos i mirar quina és la que millor funciona, maximitzant els espais que no es compten al mig, com a `pb12.cpp` o directament comptant els que sí, com a `pb12.py`.

Problema 13: Palíndroms de Morse

En aquest problema tenim dos subproblemes: transformar una paraula a Morse i dir si una certa paraula (en Morse o no) és un palíndrom. Per a la primera part el més recomenable és crear un vector amb tot l'alfabet en Morse per anar convertint caràcter a caràcter. La segona és un problema molt clàssic per al qual només cal anar comparant els caràcters movent-nos a la vegada des del principi i des del final de la paraula fins a encreuar-nos o trobar una diferència. Teniu les solucions oficials a `pb13.cpp` i `pb13.py`.

Problema 14: La formiga prudent

Per obtenir tots els punts d'aquest problema cal fer una solució amb fórmula: no es pot simular el moviment de la formiga. Hi ha diferents maneres de calcular aquesta distància, com podeu veure a les solucions `pb14.cpp` i `pb14.py`. Donem doncs indicacions per trobar la fórmula:

- Si la distància en l'eix X és més gran o igual que la distància en l'eix Y, llavors la resposta és la suma de distàncies dels dos eixos per separat. Ho podem veure com que en aquest cas podem encabir tots els desplaçaments verticals entre els horitzontals.
- Altrament, la resposta és sempre el doble de la distància en l'eix Y més o menys 1 segons sigui necessari per les paritats de les coordenades. Ho podem entendre com abans: fiquem els desplaçaments horitzontals entre els verticals, però tenint en compte que potser al final cal desfer en l'eix X ($+1$), no fer res si acabem exactament al lloc desitjat, o potser un dels desplaçaments horitzontals que hem comptat no era necessari (-1).

Problema 15: L'alpinista fatxenda

Sigui a_i l'alçada del pic i-èsim. Definim $\mathcal{T}_i$ de la següent forma:

$$\mathcal{T}_i = \{j \mid f(j) \leq f(i), j < i\}$$

Per tant, $\mathcal{T}(i)$ és el subconjunt de cims que hem vist abans del cim i i que no són més alts. Sigui ara $g(i)$ la màxima suma d'alçades que es pot aconseguir acabant al cim i. Volem veure que es compleix la fórmula següent:

$$g(i) = a_i + \max\{g(j) \mid j \in \mathcal{T}_i\}$$

És a dir, el millor que podem aconseguir triant pujar el cim i és la seva alçada més el millor que haguem pogut aconseguir amb un cim anterior que sigui més baix. Aquestes operacions es poden implementar directament amb un Treap, però adaptant una mica la idea es pot calcular ràpidament en `C++` fent servir un `std::map`, com es fa a `pb15.cpp`.

A Python no hi ha llibreries estàndar (fins a on arriba el nostre coneixement) que incloguin arbres binaris balancejats per poder resoldre aquest problema eficientment. Així doncs, cal implementar un Treap com a la solució `pb15.py` o qualsevol altre arbre binari balancejat.

Problema 16: L'investigador empanat

Aquest problema s'ha de resoldre amb un backtracking que generi totes les possibles permutacions. Per tal de que el codi sigui prou ràpid, cal anar generant només aquelles permutacions que siguin vàlides, altrament el nombre de permutacions creix molt ràpidament i el programa no serà prou ràpid. Podeu trobar solucions a `pb16.cpp` i `pb16.py`.

Problema 17: Distància de cavall màxima

Per poder calcular la mínima distància entre dos nodes qualssevol en un graf sense pesos tenim habitualment dues maneres de fer-ho: l'algorisme de Floyd-Warshall i fer un BFS des de cada node. Si tenim un graf de n nodes i m arestes, la primera solució té cost $\Theta(n^3)$

mentre que la segona té cost $\Theta(n(n+m))$. Com que, en el nostre cas, el nombre d'arestes és $m \approx 8n$, la solució òptima fa n BFSs, com a `pb17.cpp` i a `pb17.py`.

Problema 18: Rectangle màxim

Resoldrem aquest problema en dues parts. Primer veurem com reduir el problema a un de més fàcil i després veurem com resoldre la reducció.

Donada una fila, considerem la *skyline* formada amb les caselles lliures d'aquella fila i totes les superiors, fins a trobar un obstacle en la columna. Es pot veure que si fem això per a cada fila i trobem el rectangle més gran de l'*skyline*, la resposta serà el màxim de totes les files.

La transformació en un *skyline* és fàcil de fer: a la fila i, la columna j val 0 si no està lliure i 1 més el valor de la fila anterior en aquella mateixa columna en cas contrari:

.	X	.	.	.	.
.	X	.	.	.	.
.	.	X	.	.	X

$\rightarrow$

1	0	1	1	1	1
2	0	2	2	2	2
3	1	0	3	3	0

Per resoldre cada fila, simplement fem servir una pila: cada vegada que l'alçada augmenta es fica a la pila el seu inici i alçada, i quan baixa s'extreu de la pila i es mira l'àrea. Podeu veure detalls a `pb18.cpp` i a `pb18.py`.

Problema 19: Bonnie i Clide (2)

Aquest problema es pot resoldre amb programació dinàmica. Sigui s la suma de tots els diners i n el nombre de xecs. Definim $f(i, j, k)$ com una funció booleana que val cert si i només si d'entre els i primers xecs se'n poden agafar j tal que la seva suma sigui k. Clarament, la resposta del nostre problema és $f(n, \frac{n}{2}, \frac{s}{2})$.

Només ens cal definir f. Ignorant els casos base, podem veure que tenim la recurrència següent:

$$f(i, j, k) = f(i-1, j, k) \vee f(i-1, j-1, k-v_i)$$

Per consultar els detalls dels casos base, podeu consultar `pb19.cpp`. Per qüestions d'eficiència, aquest problema no es pot resoldre en Python al Jutge.

Problema 20: Samarretes de l'OIcat (2)

Sigui a_i el nombre de samarretes de mida i que es gasten per any i b_i el nombre de samarretes que ens queden després del primer any. Volem trobar un nombre c_i de samarretes a demanar de tipus i cada any de manera que després de x anys ens en quedin zero de cada tipus de samarreta. Això vol dir el següent:

$$xc_i + b_i - xa_i = 0$$

Això vol dir que

$$xc_i = xa_i - b_i$$

i per tant tenim que necessàriament x (el nombre d'anys) ha de dividir b_i per a tot i. Per tant, ha de dividir el màxim comú divisor de tots els b_i.

Així doncs, podem fer un programa que calculi tots els divisors d'aquest màxim comú divisor i de petit a gran vagi mirant si són solucions vàlides per x, és a dir, si tots els c_i resultants són positius. Podeu trobar les solucions oficials a `pb20.cpp` i `pb20.py`.