Dividir y conquistar

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos
Tecnológico de Monterrey

pperezm@tec.mx

06-2020

Contenido

Definición

Es una técnica que permite encontrar la solución de un problema descomponiéndolo en subproblemas más pequeños (dividir) y que tienen la misma naturaleza del problema original, es decir, son similares a este. Luego resuelve cada uno de los subproblemas recursivamente hasta llegar a problemas de solución trivial o conocida con antelación (conquistar) para, finalmente, unir las diferentes soluciones (combinar) y así conformar la solución global al problema.

Forma general

Procedure 1 DIVIDE_AND_CONQUER

Input: X
  if X is simple or known **then**
    return $SOLUTION(X)$
  else
    Decompose X into smaller problems $x_1, x_2, ..., x_n$
    for $i \leftarrow 1$ **to** n **do**
      $y_i \leftarrow DIVIDE\_AND\_CONQUER(X_i)$
    end for
    Combine the y_i to get the Y that is solution of X
    return Y
  end if

Búsqueda binaria

Buscar un elemento x en un arreglo ordenado A de n elementos.

Procedure 2 BINARY_SEARCH

Input: $A : Array, low : Index, high : Index, k : Key$

```
if low > high then
    return −1
else
    mid ← FLOOR((high + low)/2)
    if k = A[mid] then
        return mid
    else if k < A[mid] then
        return BINARY_SEARCH(A, low, mid − 1, key)
    else if k > A[mid] then
        return BINARY_SEARCH(A, mid + 1, high, key)
    end if
end if
```

Internal Rate Return [1]

In finance, Internal Rate of Return (IRR) is the discount rate of an investment when NPV equals zero. Formally, given T, CF_0, CF_1, ..., CF_T, then IRR is the solution to the following equation:

$$NPV = CF_0 + \frac{CF_1}{1+IRR} + \frac{CF_2}{(1+IRR)^2} + \ldots + \frac{CF_T}{(1+IRR)^T} = 0$$

Your task is to find all valid IRRs. In this problem, the initial cash-flow $CF_0 < 0$, while other cash-flows are all positive ($CF_i > 0$ for all $i = 1, 2, \ldots$).

Important: IRR can be negative, but it must be satisfied that $IRR > -1$.

Input

There will be at most 25 test cases. Each test case contains two lines. The first line contains a single integer T ($1 \leq T \leq 10$), the number of positive cash-flows. The second line contains $T+1$ integers: CF_0, CF_1, CF_2, ..., CF_T, where $CF_0 < 0$, $0 < CF_i < 10000$ ($i = 1, 2, \ldots, T$). The input terminates by $T = 0$.

[1] https://onlinejudge.org/external/118/11881.pdf

Output

For each test case, print a single line, containing the value of IRR, rounded to two decimal points. If no IRR exists, print '`No`' (without quotes); if there are multiple IRRs, print '`Too many`' (without quotes).

Sample Input

```
1
-1 2
2
-8 6 9
0
```

Sample Output

```
1.00
0.50
```

Small Factors [2]

Computation of **Discrete Fourier Transform** (DFT) is necessary in many computer programs that require some kind of digital signal processing, like audio/image processing and spectral analysis. The most popular algorithm for computing DFT is the **Fast Fourier Transform** algorithm (FFT), which takes profit from the factorization into small prime factors of the number of samples that the digital signal consists of (say, n). The most widely used variant of the FFT algorithm is the so called Radix-2 FFT, which assumes that n is a natural power of 2. Its main drawback is that the number of samples has to be increased (usually filled with zeroes) to the smallest natural power of 2 that is greater than or equal to n. This implies that given a number of samples n, the actual number of samples to be transformed may grow up to $2n$, which in practical terms means a 100% computation overload.

An alternative that substantially reduces this upper bound is using a mixed Radix-2/3 FFT algorithm. In this case, the number of samples to be transformed should be expressed as a product of prime factors 2 and 3:

$$C_{2,3} = \{n = 2^i \cdot 3^j, \text{ such that } i, j \text{ belong to } N\}$$

[2] https://onlinejudge.org/external/116/11621.pdf

Given a positive integer number m, find the smallest number n in the set $C_{2,3}$, as defined above, such that $n \geq m$. We will denote this number as $n = Next_{2,3}(m)$.

Input

The input consists of a sequence of positive integer numbers, m, one number per line. The end of the input is marked by a value $m = 0$. No input number m shall be greater than 2^{31}.

Output

For each non-zero input value m, the program is to write one line with the value of $Next_{2,3}(m)$, as defined above. No trailing/leading blank spaces should be written after/before any output number.

Sample Input

```
100
108
1000
3000
0
```

Sample Output

```
108
108
1024
3072
```

Permutaciones

Hallar todas las permutaciones de un número. Por ejemplo, las permutaciones de 123 son: {123, 231, 321, 312, 132, 213, 123}.

Procedure 3 PERMUTATION

Input: $S : String, pos : Integer$

```
if pos > 1 then
  for i ← 1 to pos do
    SWAP(number, i, pos)
    PERMUTATION(number, pos − 1)
    SWAP(number, i, pos)
  end for
else
  print S
end if
```

Exponenciación rápida

Dados dos números, x y n, calcular el resultado de x^n, haciendo uso de la técnica de dividir y conquistar.

Procedure 4 FAST_POW

Input: $x : Real, n : Integer$
  if $n < 0$ **then**
    return $FAST\_POW(1/x, -n)$
  else if $n == 0$ **then**
    return 1
  else if $n == 1$ **then**
    return x
  else if $n \mod 2 = 0$ **then**
    return $FAST\_POW(x * x, n/2)$
  else if $n \mod 2 = 1$ **then**
    return $x * FAST\_POW(x * x, (n-1)/2)$
  end if

Prefijo común más largo

Dado un conjunto de cadenas, encontrar el prefijo común más largo. Por ejemplo, dadas la siguiente cadenas: “geeksforgeeks”, “geeks”, “geek”, “geezer”, el resultado esperado es: "gee". [3]

[3] https://goo.gl/rqjV76

Procedure 5 FIND_PREFIX

Input: $A : String, B : String$

```
result ← ""
i ← 1
j ← 1
while i < A.length and j < B.length do
    if A[i] <> B[j] then
        break
    end if
    result ← result + A[i]
    i ← i + 1
    j ← j + 1
end while
return result
```

Procedure 6 COMMON_PREFIX

Input: $A : Array, low : Index, high : Index$
  if $low == high$ **then**
    return $A[low]$
  end if
  if $low < high$ **then**
    $mid \leftarrow FLOOR((high + low)/2)$
    $str1 \leftarrow COMMON\_PREFIX(A, low, mid)$
    $str2 \leftarrow COMMON\_PREFIX(A, mid + 1, high)$
    return $FIND\_PREFIX(str1, str2)$
  end if

Contando inversiones

Dado arreglo de números enteros distintos, A, determinar el número de inversiones que existen. Decimos que dos indices i $<$ j forma una inversión si $A[i] > A[i]$.
El número de inversiones de un arreglo indica: a qué distancia (o qué tan cerca) está el arreglo de estar ordenado. Si el arreglo ya está ordenado, el conteo de inversión es 0. Si el arreglo está ordenado en orden inverso, el conteo de inversión es el máximo.
Ejemplo: ¿cuántas inversiones hay en el siguiente arreglo: 2, 4, 1, 3, 5? Tiene tres inversiones (2, 1), (4, 1), (4, 3). [4]

[4] https://goo.gl/DxqxBe

Procedure 7 SORT_AND_COUNT

Input: $A, B : Array, low, high : Index$

$r \leftarrow 0$
$left \leftarrow 0$
$right \leftarrow 0$
if $(high - low + 1 = 1)$ **then**
  return 0
else
  $mid \leftarrow FLOOR((high + low)/2)$
  $left \leftarrow SORT\_AND\_COUNT(A, B, low, mid)$
  $right \leftarrow SORT\_AND\_COUNT(A, B, mid + 1, high)$
  $r \leftarrow MERGE\_AND\_COUNT(A, b, low, mid, high)$
  $COPY(A, B, low, high)$
end if
return $r + left + right$

Procedure 8 MERGE_AND_COUNT

Input: $A, B : Array.low, mid, high : Index$

```
...
while left ≤ mid AND right ≤ high do
    if A[left] < A[right] then
        B[i] ← A[left]
        left ← left + 1
    else
        B[j] ← A[right]
        right ← right + 1
        count ← count + (mid − left)
    end if
    i ← i + 1
end while
...
return count
```

Par más cercano

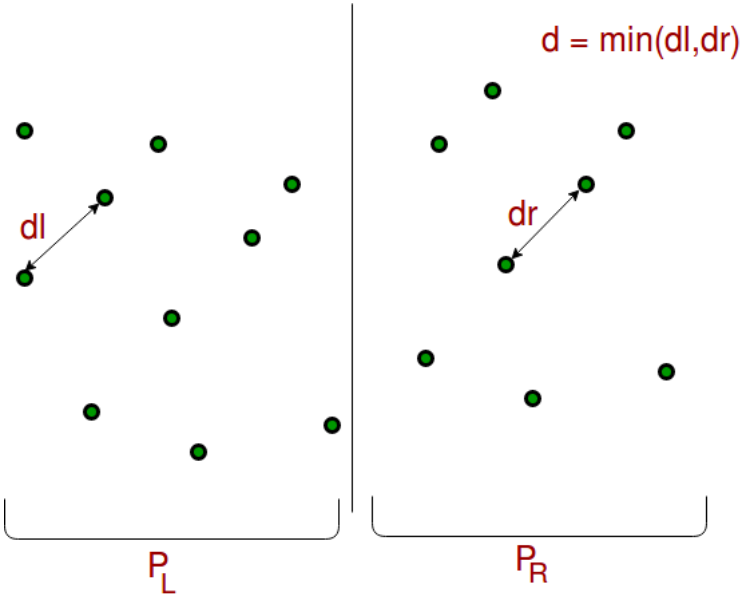
Dado un arreglo de N puntos en un plano, el problema es encontrar el par de puntos más cercano del arreglo. Este problema lo podemos encontrar en una serie de aplicaciones. Por ejemplo, en el control de tráfico áreo, se requiere monitorear los aviones que se acercan demasiado, ya que esto puede indicar una posible colisión.
La solución a fuerza bruta, de $O(n^2)$, calcula la distancia existente ente cada par de puntos y devuelve el más cercano. Podemos calcuolar la distancia más cercana en $O(n \log n)$ utilizando la estrategia de Dividir y Conquistar. [5]

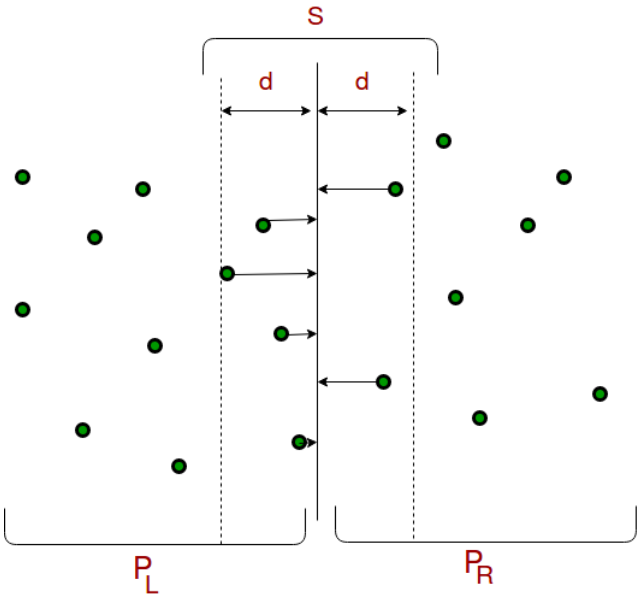
[5] https://goo.gl/36zwaA

Procedure 9 CLOSEST_PAIR

Input: $A : Array < Point >$

1. Ordenamos todos los puntos según sus coordenadas x.
2. Divide todos los puntos en dos mitades.
3. Encuentra, recursivamente, las distancias más pequeñas en ambas mitades.
4. Tomamos el mínimo de ambas distancias, d.
5. Crea un arreglo que almacene todos los puntos que se encuentran a una distancia máxima de la línea media que divide ambas mitades.
6. Encuentramos la distancia más corta de este subconjunto.
7. Devolvemos el mínimo, d, y la distancia más pequeña calculada en 4.


d = min(dl,dr)
dl
dr
P_L
P_R


S
d
d
P_L
P_R

Encontrar el número más cercano en el arreglo

Dado un arreglo de números enteros ordenados. Necesitamos encontrar el valor más cercano al número dado. El arreglo puede contener valores duplicados y números negativos. [6]

[6] https://goo.gl/XTgBzX

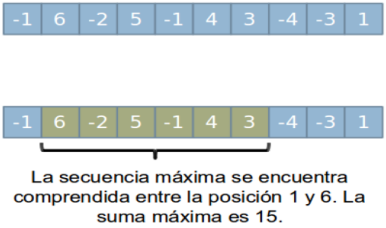
Procedure 10 FIND_CLOSEST

Input: $A : Array, target : Key$
  if $target < A[1]$ **then**
    return $A[1]$
  end if
  if $target > A[A.length]$ **then**
    return $A[A.length]$
  end if
  $low \leftarrow 1$
  $high \leftarrow A.length$
  while $low < high$ **do**
    NEXT SLIDE
  end while

$mid \leftarrow FLOOR((high + low)/2)$
if $target = A[mid]$ **then**
  return $A[mid]$
else if $target < A[mid]$ **then**
  if $mid > 0$ **and** $target > A[mid - 1]$ **then**
    return $CLOSEST(A[mid - 1], A[mid], target)$
  end if
  $high \leftarrow mid$
else
  if $mid < A.length$ **and** $target < A[mid + 1]$ **then**
    return $CLOSEST(A[mid], A[mid + 1], target)$
  end if
  $high \leftarrow mid$
end if

Secuencia de suma máxima

Dado un arreglo de n números enteros positivos y negativos, encontrar los i elementos del arreglo cuya suma se la máxima posible.



Procedure 11 MAX_SUM

Input: $A : Array, low, high : Index$

```
if (high − low + 1) = 1 then
    return A[low]
else
    mid ← FLOOR((high − low)/2)
    left ← MAX_SUM(A, low, mid)
    right ← MAX_SUM(A, mid + 1, high)
    center ← MAX_AUX(A, low, high)
    return MAX(left, right, center)
end if
```

Procedure 12 MAX_AUX

Input: $A : Array, low, high : Index$

```
left ← 0
acum ← 0
for i ← mid to low do
    acum ← acum + A[i]
    left ← MAX(left, acum)
end for
right ← 0
acum ← 0
for i ← mid + 1 to high do
    acum ← acum + A[i]
    right ← MAX(right, acum)
end for
return left + right
```