Problema 1: Kaio Ken

Explicación:

Tenemos N elementos y el problema consiste en dividir esos elementos en 2 grupos, no necesariamente la mitad para un lado y la mitad para el otro (aunque en la resolución del problema veremos que esto nos favorece), con el siguiente criterio de división: repartir dichos elementos en tantas instancias como se requieran para que cada uno esté “enfrentado” a cada otro elemento al menos una vez en todas las instancias de salida que arroja el problema, estas instancias deben ser mínimas. Al “enfrentarse” nos referimos a que si tenemos únicamente 2 elementos A y B entonces la manera de “enfrentarlos” va a ser repartiendo en un grupo el elemento A y en otro grupo el elemento B, aquí ampliamos con un ejemplo para N = 3:

Elementos:= ABC

Para resolver el problema podemos enfrentar a A con B y C, de esta manera nos queda únicamente enfrentar a B con C, entonces podemos dividir los 3 elementos en una segunda instancia pero esta vez con B por un lado, A y C por el otro. Cabe destacar que también podemos ubicar a C por un lado, y A y B por otro, y también cumpliríamos con la resolución, pero no sería minimal que es lo que pide el problema, en ese caso o bien optamos por poner a B solo o bien optamos por poner a C solo.

Complejidad: O(N Log2(N)) pues recorre todos los elementos del vector porque los tiene que repartir según corresponda, pero la serie de particiones sólo se dará una cantidad logarítmica de veces que es lo necesario para satisfacer las condiciones del problema.

Es un clásico problema de “Dividir y Conquistar”, el vector de elementos se va “partiendo” a la mitad en cada iteración para repartir cada elemento al grupo que corresponda.

Pseudocódigo:

Int rango := log2(n) + 2

If (n == 2(rango -2 )) rango := rango - 1

Para i := 1 hasta rango hacer

Para x := 0 hasta n hacer

Si x mod 2i < 2(i – 1) entonces

Imprimir “1”

Else

Imprimir “2”

Fin Si

X++

Fin Para

I++

Fin Para

Análisis experimental:

Para el análisis consideramos necesario experimentar únicamente aumentando el valor de entrada, ya que no tenemos ni mejor caso ni peor caso porque el algoritmo sin excepciones recorre todo el vector una cantidad logarítmica de veces según la entrada N, con lo cual presentamos un gráfico mostrando el valor de la entrada aumentando en potencias de 2. Con lo cual se deduce que, cuanto mayor es el tamaño de la entrada, más ciclos de CPU son necesarios y por tanto más tiempo.

Apéndice:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | for (i = 1; i < rango; i++) { | |  |  |  | | --- | |  | |  | |  | |  | | for (x = 0; x < n; x++) { | |  | |  | | if ((x % (int)pow(2, i)) < pow(2, i - 1)) { | |  | |  | | cout << GRUPO\_1; | |  | |  | | } | |  | |  | | else { | |  | |  | | cout << GRUPO\_2; | |  | |  | | } | |  | |  | | if (x < n - 1) cout << " "; | |  | |  | | } | |  | |  | | cout << endl; | |  |  |  | | --- | | } | |

Problema 2: Genkidama

Explicación:

Tenemos un plano del cual no nos importan sus dimensiones. Suponiendo que el plano en estado inicial está vacío y la entrada del problema consiste en una serie posiciones que cumplen que X1 > X2 > … > Xn >= 0 y 0 <= Y1 <= Y2 <= … <= Yn, entonces podemos considerar que esas posiciones en el plano están ocupadas.

El problema consiste en buscar el menor número de intentos necesarios para remover las posiciones ocupadas.

Consideramos que un punto se remueve si uno se “ubica” en esa posición y considera que se remueve ese punto y todos los demás puntos que están a una distancia T sobre ambos ejes, siendo T una entrada del programa que es un número entero positivo.

Supongamos N = 2, tenemos una grilla de 2x2, y suponiendo que tenemos los puntos ubicados de la siguiente manera:

Si T = 0 entonces el menor número de intentos que necesito para remover ambos puntos es 2. Si T fuera mayor o igual a 1 entonces con un solo intento ya removimos ambos puntos.

Correctitud:

Complejidad:

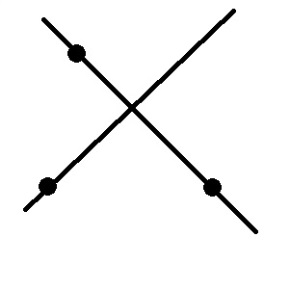
El algoritmo cuenta con una complejidad de O(N) esto es así porque

Problema 3: Kamehameha

Explicación:

Tenemos un plano infinito con N posiciones ocupadas. El problema consiste en remover dichos puntos dibujando semirrectas desde una posición inicial en una dirección indicada, los puntos que sean atravesados por la semirrecta se consideran removidos. La cantidad de semirrectas necesarias para remover todos los puntos debe ser la mínima posible.

Supongamos el siguiente plano con N = 3:



Entonces se ve claramente que con dos rectas alcanza para solucionar el problema.

Correctitud:

El algoritmo se basa en la técnica de backtracking, por lo tanto sin tener en cuenta podas ni complejidad, podemos decir lisa y llanamente que recorremos punto por punto buscando todas las posibles soluciones al problema. Esto es, por cada punto disponible, elegimos otro para trazar una recta y luego verificar cuántos puntos pasan por dicha recta, luego se aplica el mismo procedimiento con los puntos restantes, contando la cantidad de rectas necesarias. Una vez finalizado este proceso volvemos a tomar todos los puntos y nos paramos sobre el primer punto disponible, pero esta vez elegimos otro punto para trazar la recta distinto a la iteración anterior, verificamos cuántos puntos se destruyen y aplicando el mismo procedimiento.

Esto nos asegura evaluar todas las posibles soluciones para todos los puntos disponibles.

En cuanto a las podas, naturalmente verificamos si el punto ya fue destruido y además chequeamos no trazar una recta entre dos puntos en los cuales esos dos puntos sean el mismo, lo cual no sería una recta.

Hasta aquí seguimos analizando todas las posibilidades pues estamos descartando únicamente puntos ya destruidos y no procesar dos veces el mismo punto, por tanto seguimos garantizando el correcto funcionamiento del algoritmo.

Por último, para mejorar el rendimiento y además satisfacer los requerimientos del problema descartamos aquellas soluciones que llegado un momento sean peores que la mejor solución encontrada hasta el momento.

Con las podas correspondientes recién mencionadas podemos asegurar sin problemas que el algoritmo evalúa todas las soluciones factibles a ser la mejor solución y las demás las descarta.

Complejidad:

La complejidad del algoritmo sin podas recae en O(NN + 2) pues por cada punto recorremos todos para trazar todas las posibles rectas, entonces partiendo de un punto cualquiera trazamos rectas con todos los demás puntos en O(N), si esto lo vamos a hacer para N puntos entonces tenemos hasta aquí O(N2). Lo que resta es la recursividad del algoritmo pues si tenemos N puntos y trazamos una recta entre los 2 primeros, luego vamos a tener que calcular todas las posibles soluciones para los N – 2 puntos restantes (Esto sigue siendo O(N)). Con lo cual se concluye que para N puntos existentes cuyas rectas con otros puntos se calculan en O(N2) tendremos una productoria de N veces N2, lo que quedaría como N2 \* …(N - 2 veces)… \* N2 = NN + 2.