UNIVERSIDAD DE MÁLAGA

Análisis y Diseño de Algoritmos

Práctica 4

1.- Dado un array de *N* de enteros no negativos y un número *K*, dividir los elementos del array en dos conjuntos de longitud *K* y *N – K* de forma que la diferencia de la suma de los elementos de ambos conjuntos sea máxima.
Ejemplos:

- Para arr[] = {8, 4, 5, 2, 10} y k = 2 se pueden construir los conjuntos {4, 2} y {8, 5, 10}. La diferencia (8 + 5 + 10) - (4 + 2) = 17 es la máxima que se puede conseguir.
- Si arr[] = {1, 1, 1, 1, 1, 1, 1, 1} y k = 3, los conjuntos para obtener la máxima diferencia son {1, 1, 1, 1, 1} y {1, 1, 1}.

a) Implementar un algoritmo de fuerza bruta que resuelva el problema. Estimar su orden de complejidad.
Se utilizará una máscara binaria (bitmask), que no es más que un número cuya representación binaria representa una solución al problema. En el caso de conjuntos, el bit ***i*** indica si el elemento de la posición ***i*** pertenece (1) o no (0) al conjunto solución. Por ejemplo, el número 7, cuya representación binaria es 1001, indicaría que el elemento 0 y el 3 pertenecen al conjunto solución, mientras que los elementos 1 y 2 no.
Se utilizarán operadores que trabajan con bits (&, |, ^, ~, <<, >>) para modificar las máscaras. Dichas operaciones tienen baja precedencia por lo que a veces será necesario utilizar paréntesis. Las operaciones más útiles son:
1) Activa el bit *i* : M = M | (1<< i)
2) Oculta el bit *i*: M = M & ~(1<< i)
3) Comprueba si *i* está activo: M & (1<<i)
Si el bit *i* está activo se obtiene un número distinto de *0*. En otro caso, *0*
4)* Conmuta *el bit i (de 0 a 1 o viceversa): *M =M ^ (1<<i)*
5)* Activa todos los bits de un conjunto de tamaño *n: *M = (1<<n)-1*

La desventaja de esta técnica se encuentra en que el tamaño de los tipos numéricos (int, long) limita su aplicación cuando el número de elementos es muy grande.

b) Implementar un algoritmo voraz más eficiente que el algoritmo anterior.

2.- Dado un array *A* de tamaño *n* que cumple lo siguiente:

i. Cada posición el array contiene un policía o un ladrón.
ii. Cada policía sólo puede atrapar a un ladrón.
iii. Un policía no puede atrapar a ladrones que están a más de *k* posiciones de distancia del policía.

Se desea calcular el máximo número de ladrones que pueden atraparse.
Ejemplos:

- Si A = {'P', 'T', 'T', 'P', 'T'} y k = 1, el resultado es 2. El primer policía atrapa al primer ladrón y el segundo policía al segundo ladrón o al tercero.
- Si A = {'T', 'T', 'P', 'P', 'T', 'P'} y k = 2, el resultado es 3.
- Si A = {'P', 'T', 'P', 'T', 'T', 'P'} y k = 3, el resultado es 3.

Un algoritmo de fuerza bruta formaría todos los conjuntos viables de combinaciones de policía y ladrón y devolvería aquél que tuviera máxima cardinalidad. La complejidad temporal de ese enfoque sería exponencial, pero se puede encontrar una solución más eficiente utilizando algoritmos voraces.

a) Estrategia 1: Intentar que cada policía atrape al ladrón más cercano a él.
b) Estrategia 2: Intentar que cada policía atrape al ladrón más alejado a él dentro del rango de *k* posiciones.

c) Estrategia 3: centrarse en las capturas y no en los policías, e intentar realizarlas en cuanto sea posible:
    1. Obtener los índices *p* del policía más a la izquierda y *t* del ladrón más a la izquierda.
    2. Si $|p-t|<=k$, incrementar la captura e incrementar *p* y *t*. En otro caso, incrementa el mínimo índice entre p y *t* al del siguiente policía o ladrón según corresponda.
    3. Repetir los pasos previos hasta que no haya más policías o ladrones.
    4. Devolver el número de capturas

Implementar las tres estrategias y estudiar la corrección de las mismas.