Semana 13

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

10-2021

Algoritmos aleatorizados

Algoritmos Aleatorizados

Un algoritmo aleatorio es un algoritmo cuyo comportamiento depende, en gran medida, del resultado de elecciones hechas al azar. Existen dos ventajas principales que suelen tener los algoritmos aleatorios:

- A menudo el tiempo de ejecución o el espacio utilizado en memoria es menor que el del mejor algoritmo determinista que se conozca para el mismo problema.
- Si observamos los diversos algoritmos aleatorios que se han inventado hasta ahora, descubrimos que, invariablemente, son extremadamente sencillos de comprender e interpretar.

- Además, existen algunos algoritmos deterministas, especialmente aquellos que exhiben un buen tiempo promedio de ejecución, que con la simple introducción de elecciones al azar es suficiente para convertir un algoritmo simple e ingenuo con un mal comportamiento en el peor de los casos en un algoritmo aleatorizado que funciona bien con alta probabilidad para todas las posibles entradas. Esto será evidente cuando estudiemos el algoritmo de ordenamiento conocido como Quicksort.
- Es muy importante considerar que el comportamiento de este tipo de algoritmos puede variar en diferentes ejecuciones aún cuando las entradas sean las mismas.

Supongamos que hay una arreglo con la longitud par n. La mitad de las entradas del arreglo son ceros y la mitad restante son unos. El objetivo aquí es encontrar un índice que contenga un 1.

- " Las Vegas" no juega con la corrección sino con el tiempo de ejecución.
- "Monte Carlo" no apuesta por el tiempo de ejecución, sino por la corrección.

```
//Las Vegas algorithm
repeat:
    k = RandInt(n)
    if A[k] == 1,
        return k;

//Monte Carlo algorithm
repeat 300 times:
    k = RandInt(n)
    if A[k] == 1
        return k
return "Failed"
```

Buscando un elemento repetido

Procedure 1 FIND_ELEMENT

Input: A : *Array*
  while *TRUE* **do**
    $i \leftarrow rand()$ mód $A.length$
    $j \leftarrow rand()$ mód $A.length$
    if $i \neq j$ **and** $A[i] = A[j]$ **then**
      return i
    end if
  end while

Encontrar la mediana

Supongamos que nos dan un conjunto S de n números. La mediana es el número que estaría en la posición media si tuviéramos que ordenarlos. Aunque, existe una "dificultad técnica" si n es par, ya que no existe una posición media; así que tendremos que ser más específicos: la mediana de S es igual al k-ésimo elemento más grande de S, en donde $k = \frac{n+1}{2}$ si es n es impar y $k = \frac{n}{2}$ si n es par. Por simplicidad de la explicación consideraremos que todos los números son distintos. Resulta evidente que si ordenamos los elementos resulta muy fácil encontrar la mediana en $O(n \log_2 n)$. Pero ¿sería posible hacerlo sin ordenar y logrando el mismo tiempo o incluso mejor? Bueno, pues lo intentaremos usando aleatorización con la técnica de "Divide y vencerás".

El primer paso clave es pasar del problema de la búsqueda de la mediana al problema más general de selección. Da un conjunto S de n números y un número k entre 1 y n, considera la función `SELECT (S, k)` que devuelve el k-ésimo elemento más grande en S. Como casos especiales, `SELECT` resuelve el problema de encontrar la mediana de S mediante `SELECT (S, n/2)` o `SELECT(S, (n + 1)/2)`. Además, también permite resolver los problemas de encontrar el mínimo, `SELECT (S, 1)`, y el máximo, `SELECT(S, n)`.

La estructura básica de este algoritmo es la siguiente: Vamos a elegir un elemento a_i del conjunto S. A partir de este "pivote", formaremos dos conjuntos S_{lower}, los elementos menores al pivote, y $S_{greater}$, los elementos mayores al pivote. Con esto, ya podemos determinar cuál de los dos conjuntos, S_{lower} y $S_{greater}$, contiene el k-ésimo elemento más grande y recorrer solo ese.

Procedure 2 SELECT

Input: $S : Array, k : Integer$

Choose a pivot, a_i, uniformly at random
for each element, a_j in S **do**
  if $a_j < a_i$ **then**
    Place a_j in S_{lower}
  end if
  if $a_j > a_i$ **then**
    Place a_j in $S_{greater}$
  end if
end for
$i \leftarrow |S_{lower}|$
if $i = k$ **then**
  return a_i
else if $i > k$ **then**
  $SELECT(S_{lower}, k)$
else
  $SELECT(S_{greater}, k - i - 1)$
end if

Quicksort

La técnica aleatoria de "Divide y vencerás" que usamos para encontrar la mediana será nuestra base del algoritmo Quicksort. Como antes, vamos a elegir un pivote para el conjunto de entrada S y separaremos S en los elementos menores y mayores al pivote. La diferencia es que, en lugar de buscar la mediana en un solo lado, tendremos que hacerlo para ambos lados de la recursión. Además, necesitamos incluir un caso base cuando el tamaño del conjunto es menor a 4.

Procedure 3 QUICKSORT_V1

Input: S : *Array*
  if $|S| < 4$ **then**
    Sort S
  else
    while no center pivot found **do**
      Choose a pivot, a_i, uniformly at random
      for each element, a_j in S **do**
        if $a_j < a_i$ **then**
          Place a_j in S_{lower}
        end if
        if $a_j > a_i$ **then**
          Place a_j in $S_{greater}$
        end if
      end for
      if $|S_{lower}| \geq |S|/4$ **and** $|S_{greater}| \geq |S|/4$ **then**
        a_i is the central pivot
      end if
    end while
    $QUICKSORT(S_{lower})$
    $QUICKSORT(S_{greater})$
  end if

Técnicas de búsqueda avanzada

Técnicas de búsqueda avanzada

Los algoritmos que hemos analizado en capítulos anteriores se encuentran diseñados para explorar espacios de búsqueda de forma sistemática. No poseen información adicional sobre el problema más allá de la que se proporciona en la definición de este. Todo lo que pueden hacer es generar posibles soluciones parciales y determinar si se ha llegado, o no, al resultado final. A este tipo de búsqueda se les conoce como búsquedas ciegas.

Sin embargo, estos algoritmos pueden ser mejorados usando técnicas de programación y/o estructuras de datos que nos permiten realizar una búsqueda más informada o reducir la memoria empleada en el proceso de generación de la solución. Por ejemplo, las técnicas de Backtracking y Programación Dinámica se ven muy beneficias con el uso de máscaras de bits para almacenar información.

Backtracking con Bitmask

Backtracking con Bitmask

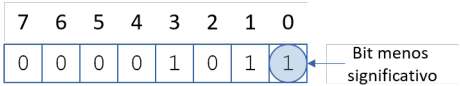
Retomemos el problemas de la suma de subconjuntos:

- Dado un conjunto S de N números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumple, se debe responder con el conjunto vacío.

Ver código "subsetv1.cpp"

Entonces, ¿qué podemos hacer para reducir este tiempo? La respuesta más sencilla sería substituir el método `insert` por alguna operación o función más eficiente. Quizás algo $O(1)$, para que el tiempo de ejecución no se vea tan afectado. Y es aquí donde entran las máscaras de bits. Las máscaras de bits se emplean en la solución de problemas que requieren de tiempo y/o memoria exponencial.

Una máscara de bits no es más que un número binario que representa algo. Tomemos como ejemplo el problema de la suma de subconjuntos. Dado un conjunto inicial, $A = \{x_1, x_2, \cdots, x_n\}$, queremos conocer qué elementos del conjunto suman la cantidad C. Es decir, cuales elementos han sido seleccionados. En este caso, supongamos que el conjunto inicial es $A = 2, 3, 7, 9$. Podemos representar cualquier subconjunto de A usando una máscara de 4 bits. En esta secuencia de bits, el elemento i-ésimo se considera seleccionado si y sólo si el i-ésimo bit de la máscara está prendido, es decir, es igual a 1.



Supongamos queremos representar el subconjunto $X = \{2, 3, 9\}$. Si consideramos que el elemento 1 es el bit menos significativo y así sucesivamente.

Las operaciones básicas que podemos realizar sobre una máscara son establecer (prender un bit), remover (apagar un bit) o verificar (determinar si un bit está prendido). Para realizar estas operaciones usaremos los operadores lógicos or (|), and (&), desplazamiento a la izquierda («) y desplazamiento a la derecha (»).

Digamos que tenemos la máscara del ejemplo anterior, b = 00001011,

- Si queremos establecer el *i*-ésimo bit de una máscara, usaremos la operación: b | (1 « i). Por ejemplo, si queremos establecer el tercer elemento, i = 2:

```
1 << 2 = 00000100
00001011 | 00000100 = 0001111
```

  Así que ahora, el subconjunto también incluye al tercer elemento, $X = \{2, 3, 7, 9\}$.

- Si queremos remover el *i*-ésimo bit de una máscara, debemos usar la siguiente operación: b & !(1 « i). Digamos que ahora queremos remover el segundo elemento, i = 1, del subconjunto:

```
1 << 1 = 00000010
!00000010 = 11111101
00001111 & 11111101 = 00001101
```

  De esta forma, la nueva máscara representará $X = \{2, 7, 9\}$.

- Si queremos verificar si el i-ésimo bit está prendido, usaremos la siguiente operación: b & (1 « i). Si el i-ésimo bit está prendido, el resultado será diferente de cero. Supongamos que queremos verificar si el cuarto elemento se encuentra en el subconjunto, i = 3:

```
        1 << 3 = 00001000
00001101 & 00001000 = 00001000
```

  Ya que el resultado es igual a 1, podemos determinar que el cuarto elemento se encuentra en el subconjunto X.

Ver código "subsetv2.cpp"