Algoritmos Aleatorizados

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

12-2022

Contenido

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"
```

Procedure 1 FIND_ELEMENT

Input: A : *Array*

```
while TRUE do
    i ← rand() mód A.length
    j ← rand() mód A.length
    if i ≠ 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 ← |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
```

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

```
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| ≥ |S|/4 and |S_greater| ≥ |S|/4 then
        a_i is the central pivot
      end if
    end while
    QUICKSORT(S_lower)
    QUICKSORT(S_greater)
  end if
```