GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: LA SELECCIÓN RÁPIDA (QUICKSELECT)

1. Introducción

Dado un arreglo A de tamaño n y un número k. El problema es encontrar K-th más grande o K-th más pequeño en el arreglo, es decir, K-enésimo elemento con respecto a un criterio de ordenación.

2. Conocimientos previos

2.1. Ordenamiento Rápido *QuickSort*

QuickSort es un algoritmo de clasificación eficiente y de uso general . Quicksort fue desarrollado por el informático británico Tony Hoare en 1959 y publicado en 1961. Todavía es un algoritmo de uso común para clasificar. En general, es un poco más rápido que combinar ordenación y heapsort para datos aleatorios, particularmente en distribuciones más grandes

QuickSort es un algoritmo de divide y vencerás . Funciona seleccionando un elemento *pivote* del arreglo y dividiendo los otros elementos en dos sub-arreglos, según sean menores o mayores que el pivote. Por esta razón, a veces se le llama clasificación de intercambio de partición. A continuación, los subconjuntos se ordenan recursivamente . Esto se puede hacer en el lugar , lo que requiere pequeñas cantidades adicionales de memoria para realizar la clasificación.

3. Desarrollo

La idea básica es utilizar la idea del algoritmo de *QuickSort*. En realidad, el algoritmo es simple.

La selección rápida (*QuickSelect*) es un algoritmo de selección para encontrar el k- ésimo elemento más pequeño en una lista desordenada, también conocida como estadística de k- ésimo orden . Al igual que el algoritmo de clasificación Quicksort relacionado , fue desarrollado por Tony Hoare y, por lo tanto, también se conoce como algoritmo de selección de Hoare.

QuickSelect utiliza el mismo enfoque general que quicksort, eligiendo un elemento como pivote y dividiendo los datos en dos según el pivote, según corresponda, como menor o mayor que el pivote. Sin embargo, en lugar de recurrir a ambos lados, como en la ordenación rápida, la selección rápida solo recurre a un lado: el lado con el elemento que está buscando.

En *QuickSort*, hay un subprocedimiento llamado partitionque puede, en tiempo lineal, agrupar una lista (desde índices lefthasta right) en dos partes: las menores que cierto elemento y las mayores o iguales que el elemento. Aquí hay un pseudocódigo que realiza una partición sobre el elemento list[pivotIndex]

```
function partition(list, left, right, pivotIndex) is
   pivotValue := list[pivotIndex]
   swap list[pivotIndex] and list[right]//Mueve el pivote hasta el final
   storeIndex := left
   for i from left to right-1 do
      if list[i] < pivotValue then
```

```
        swap list[storeIndex] and list[i]
        increment storeIndex
    swap list[right] and list[storeIndex]// Mover el pivote a su lugar final
    return storeIndex
```

Esto se conoce como el esquema de partición de Lomuto , que es más simple pero menos eficiente que el esquema de partición original de Hoare .

En *QuickSort*, ordenamos recursivamente ambas ramas, lo que lleva al mejor de los casos O($n \log n$) tiempo. Sin embargo, al hacer la selección, ya sabemos en qué partición se encuentra nuestro elemento deseado, ya que el pivote está en su posición final ordenada, con todos los que le preceden en un orden no ordenado y todos los que le siguen en un orden no ordenado. Por lo tanto, una única llamada recursiva localiza el elemento deseado en la partición correcta y nos basamos en esto para la selección rápida:

```
// Devuelve el k-esimo elemento menor de la lista dentro de left..right
    inclusivo
// (es decir, left <= k <= right)
function select(list, left, right, k) is
    if left = right then // Si la lista contiene solo un elemento,
        return list[left] // retorno ese elemento
    pivotIndex  := ...    // selecciona un pivotIndex entre la izquierda y
        derecha
                          // p. ej., izquierda + piso(rand()  % (derecha -
                              izquierda + 1))
    pivotIndex  := partition(list, left, right, pivotIndex)
    // El pivote esta en su posicion ordenada final
    if k = pivotIndex then
        return list[k]
    else if k < pivotIndex then
        return select(list, left, pivotIndex - 1,k)
    else
        return select(list, pivotIndex + 1, right, k)
```

Tenga en cuenta la similitud con la ordenación rápida: así como el algoritmo de selección basado en mínimos es una ordenación de selección parcial, esta es una ordenación rápida parcial, que solo genera y divide O($\log n$)de su O(n)particiones Este procedimiento simple tiene un rendimiento lineal esperado y, al igual que la ordenación rápida, tiene un rendimiento bastante bueno en la práctica. También es un algoritmo en el lugar , que solo requiere una sobrecarga de memoria constante si la optimización de llamadas de cola está disponible, o si se elimina la recursividad de cola con un bucle:

```
function select(list, left, right, k) is
    loop
        if left = right then
            return list[left]
        pivotIndex := ... // selecciona pivotIndex entre izquierda y derecha
```

```
      pivotIndex := partition(list, left, right, pivotIndex)
      if k = pivotIndex then
         return list[k]
      else if k < pivotIndex then
         right := pivotIndex - 1
      else
         left := pivotIndex + 1
```

4. Implementación

4.1. C++

```
int partition(int arr[], int l, int r){
   int x = arr[r], i = l;
   for (int j = l; j <= r - 1; j++) {
      if (arr[j] <= x) {
         swap(arr[i], arr[j]);i++;
      }
   }
   swap(arr[i], arr[r]); return i;
}

int kthSmallest(int arr[], int l, int r, int k){
   if(k > 0 && k <= r - l + 1) {
      int index = partition(arr, l, r);
      if (index - l == k - 1) return arr[index];
      if (index - l > k - 1) return kthSmallest(arr, l, index - 1, k);
          return kthSmallest(arr, index + 1, r, k - index + l - 1);
   }
   return INT_MAX;
}
```

Se implementa una solución lineal determinista en la biblioteca estándar C ++ como std::nth_element.

4.2. Java

```
public int nth_element(int[] a, int low, int high, int n) {
   if (low == high - 1) return low;
   int q = randomizedPartition(a, low, high);
   int k = q - low;
   if (n < k) return nth_element(a, low, q, n);
   if (n > k) return nth_element(a, q + 1, high, n - k - 1);
   return q;
}

public int randomizedPartition(int[] a, int low, int high) {
```

```
    swap(a, low + rnd.nextInt(high - low), high - 1);
    int x = a[high - 1]; int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] <= x) {
            ++i;swap(a, i, j);
        }
    }
    return i;
}

public void swap(int[] a, int i, int j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;
}
```

5. Complejidad

Al igual que *QuickSort*, es eficiente en la práctica y tiene un buen desempeño en el caso promedio, pero tiene un desempeño deficiente en el peor de los casos. La variación que realiza con respecto al *QuickSort* reduce la complejidad media de O($n \log n$) a O(n), con el peor caso de O (n^2).Es sensible al pivote que se elige. Si se eligen buenos pivotes, es decir, aquellos que disminuyen consistentemente el conjunto de búsqueda en una fracción dada, entonces el conjunto de búsqueda disminuye exponencialmente en tamaño y por inducción (o sumando la serie geométrica) uno ve que el rendimiento es lineal, ya que cada paso es lineal y el tiempo total es una constante por esto (dependiendo de qué tan rápido se reduzca el conjunto de búsqueda). Sin embargo, si los pivotes malos se eligen constantemente, como la disminución de un solo elemento cada vez, entonces el rendimiento en el peor de los casos es cuadrático: O(n^2).Esto ocurre, por ejemplo, al buscar el elemento máximo de un conjunto, utilizando el primer elemento como pivote y ordenando los datos.

6. Aplicaciones

Quickselect y sus variantes son los algoritmos de selección más utilizados en implementaciones eficientes del mundo real.

7. Ejercicios propuestos

Aquí están un grupo de ejercicios que se pueden resolver con este algoritmo:

- CODECHEF: Median