پویا آقاحسینی ۹۵۱۳۰۰۶

در این سوال از روش Selection in worst-case linear time در بخش ۹.۳ کتاب CLRS و تابع Partition در QuickSort که در بخش ۷.۱ کتاب هست استفاده میکنیم :

تابع Partition به این گونه است که آخرین عنصر را به عنوان پیووت انتخاب میکند و با انجام جابجایی در آرایه عناصر کوچتر از آن را قبل آن و بزرگتر را پس از آن قرار میدهد :

```
partition(array, l, r)
      {
      x ← array.end
      i ← l
         for j in range(l,r - 1)
            if jth element is less than x
            {
               swap i with j
               i++
            }
         }
          swap i with r
         return i                // Returns the position of pivot after partitioning
      }
```

روش Selection in worst-case linear time در کتاب به این گونه بود که عناصر را پنج تا پنج تا تقسیم میکرد و برای هر کدام میانه را پیدا میکرد و برای همه ی میانه ها نیز یک میانه پیدا میکرد و سپس تابع پارتیشن را با پیووت مدینِ مدین ها صدا میزد تا ۶ – n ۳/۱۰ تا از عناصر حداقل کوچکتر از آن و 6- 7n/10 تا از عناصر از آن بزرگتر بودند و در بدترین حالت تابع را دوباره برای عناصر بیشتر صدا میزدیم :

```
 kthSmallest(array,l ,r ,k)
{
     if k in range of array
     n ←  r-l+1
     median[(n+4)/5]
     for i in range of 5
          median[i] ←  findMedian(arr+l+i*5, 5)
          if i*5 is less than n //Last Group of elements
               median[i] = findMedian(arr+l+i*5, n%5)
               i++
     medianOfMedians ← kthSmallest(median, 0, i-1, i/2)
     position ← partition(arr, l, r, medianOfMedians)
     If position is same as k :
```

```
        return arr[pos];
     If position is more :
        return kthSmallest(arr, l, pos-1, k) // recur for left
      else
         return kthSmallest(arr, pos+1, r, k-pos+l-1) // recur for
right subarray
```

حال برای حل سوال ابتدا فرض میکنیم که برای مقایسه یک تابع Compare(x,y) داریم که ترتیب عناصری که در لیست زیبا داریم را میتوانیم مقایسه کنیم، همچنین یک تابع برای دریافت مولفه دوم که تعداد تکرار شدن هر عنصر هست به نام frequency(x) داریم.

همچنین یک تابع frequencySum(array,l,r) داریم که تعداد تکرار شدن های عناصر تا خانه pos را جمع میزند ، این تابع در قسمت if های سبز رنگ سودوکد بالا استفاده میشود چون میخواهیم k امین عنصر را پیدا کنیم در حالی که عناصر امکان تکرار پذیری دارند بنابر این باید تعداد تکرار شدن آن ها را نیز لحاظ کنیم.

```
frequency(x):
return frequency of element x
```

```
frequencySum(array,l,r):
for i in range l to r:
     s + = frequency(array[i])
 return s
```

```
Pos ← partition(array, l, r, medianOfMedians)
s ← frequencySum(array,0,pos)
if s=k
     return array[pos]
else if s>k
     return kthSmallest(arr, l, pos-1, k) // recur for left
else
     return kthSmallest(arr, pos+1, r, k-pos+l-1) // recur for right
     subarray
```

پیچیدگی زمانی را برای تابع compare , frequency ، را O(1) در نظر میگیریم اما تابع frequencySum در بدترین حالت همه ی عناصر را اگر جمع بزند O(m) برای آن داریم و چون m عنصر در لیست زیبا داریم و بقیه عملیات ها درست مانند حل مسئله برای آرایه عادی است که O(n) بود ، یک ضریبی از O(m) برای این برنامه داریم که در نتیجه O(m) میشود. در خصوص پیچیدگی زمانی به صفحات ۲۲۱ و ۲۲۲ کتاب CLRS ارجاع داده شده است.

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \ge 140 . \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that $T(n) \le cn$ for some suitably large constant c and all $n > 0$. We begin by assuming that $T(n) \le cn$ for some suitably large constant c and all $n < 140$; this assumption holds if c is large enough. We also pick a constant a such that the function described by the $O(n)$ term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by an for all $n > 0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) , \end{aligned}$$

which is at most cn if

$$-cn/10 + 7c + an \le 0 . \tag{9.2}$$