CSCE750

ByY. Meng   gmail: y(dot)meng201011

★ In the search problem, the input is a sourted array A of size n along with a search key k. The output is an integer i such that $A[i] = k$, or -1 if k is not in A. Prove, using the decision tree method, that any correct algorithm for this problem based on comparison ($<$, $>$, $\leq$, $\geq$, and $=$) between elements take $\Omega(\lg n)$ time.

We can build a (binary) decision tree with the sorted array A as following.
1. Find the median of array and make it as root.
2. Recursively find the median for both left part and right part of array A.
   2.1. Find the median of left part and make it as $root'.left$.
   2.2. Find the median of right part and make it as $root'.right$.
   2.3. Return $root'$ ($root'$ is the median found in upper level).

Then, height of this decision tree is $\lceil \frac{n}{2} \rceil$. And, for any given k, we need to compare it with at most $\lceil \frac{n}{2} \rceil$ (a path from root to any leaf) nodes to determin the certain $A[i]$ that equals k or k is not in array A (none node in this path equals k). Therefore, the running time is $\Omega(\lg n)$.

Note: Each round of comparison, there are 3 results for given number: falls in left subtree, falls in right subtree, equals the root of current subtree. Thus, the worst case happens when searching for a leaf or the number is not in array.

(P219-220) exe. 9.2.3 Write an iterative version of RANDOMIZED-SELECT.

In [32]:

```
# RECURSIVE
import random

def Swap (a, b):
    return b, a

def Partition (A, p, r):
    key = A[r]
    i = p - 1
    for j in range (p, r):
        if A[j] <= key:
            i = i + 1
            A[i], A[j] = Swap (A[i], A[j])
    i = i + 1
    A[i], A[r] = Swap (A[i], A[r])
    return i

def Randomized_Partition (A, p, r):
    i = random.randint(p, r)
    print "i = ", i
    A[i], A[r] = Swap (A[i], A[r])
    return Partition (A, p, r)

def Randomized_Select_Recursively (A, p, r, i):
    if (p == r):
        return A[p]
    q = Randomized_Partition (A, p, r)
    k = q - p + 1
    print p, r, i, ", ", q
    print A
    if (i == k):
        return A[q]
    elif (i < k):
        return Randomized_Select_Recursively (A, p, q - 1, i)
    else:
        return Randomized_Select_Recursively (A, q + 1, r, i - k)
```

In [33]:

```
testA = [5, 4, 2, 9, 6, 7, 0, -3, 1, -8]
print testA, "\n"
print Randomized_Select_Recursively (testA, 0, len(testA) - 1, 3)
print "\n"
print testA
```

```
[5, 4, 2, 9, 6, 7, 0, -3, 1, -8]

i =  3
0 9 3 ,  9
[5, 4, 2, -8, 6, 7, 0, -3, 1, 9]
i =  8
0 8 3 ,  3
[-8, 0, -3, 1, 6, 7, 4, 2, 5, 9]
i =  1
0 2 3 ,  2
[-8, -3, 0, 1, 6, 7, 4, 2, 5, 9]
0



[-8, -3, 0, 1, 6, 7, 4, 2, 5, 9]
```

In [16]:

```
# ITERATIVE
import random

def Swap (a, b):
    return b, a

def Partition (A, p, r):
    key = A[r]
    i = p - 1
    for j in range (p, r):
        if A[j] <= key:
            i = i + 1
            A[i], A[j] = Swap (A[i], A[j])
    i = i + 1
    A[i], A[r] = Swap (A[i], A[r])
    return i

def Randomized_Partition (A, p, r):
    i = random.randint(p, r)
    A[i], A[r] = Swap (A[i], A[r])
    return Partition (A, p, r)

def Randomized_Select_Iteratively (A, p, r, i):
    while (p < r):

        q = Randomized_Partition (A, p, r)
        k = q - p + 1
        print p, r, i, ", ", q
        print A
        if (k == i):
            return A[q]
        elif (k < i):
            p = q + 1
            i = i - k
        else:
            r = q - 1
    return A[p]
```

In [19]:

```
testA = [5, 4, 2, 9, 6, 7, 0, -3, 1, -8]
print testA,"\n"
print Randomized_Select_Iteratively (testA, 0, len(testA) - 1, 3)
print "\n"
print testA
```

```
[5, 4, 2, 9, 6, 7, 0, -3, 1, -8]

0 9 3 ,  6
[-8, 4, 2, 0, -3, 1, 5, 6, 7, 9]
0 5 3 ,  5
[-8, 1, 2, 0, -3, 4, 5, 6, 7, 9]
0 4 3 ,  1
[-8, -3, 2, 0, 1, 4, 5, 6, 7, 9]
2 4 1 ,  4
[-8, -3, 1, 0, 2, 4, 5, 6, 7, 9]
2 3 1 ,  2
[-8, -3, 0, 1, 2, 4, 5, 6, 7, 9]
0


[-8, -3, 0, 1, 2, 4, 5, 6, 7, 9]
```

exe. 9.2.4 Suppose we use RANDOMIZED-SELECT to select the minimum element of the array A = [3, 2, 9, 0, 7, 5, 4, 6, 1]. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

The worst case is that in RANDOMIZED_PARTITION method, we get the random pivots in reverse order. That is, in given array A, the sequence of random pivot elements is 9-7-6-5-4-3-2-1-0.

[3, 2, 1, 0, 7, 5, 4, 6, **9**]
[3, 2, 1, 0, 6, 5, 4, **7**, 9]
[3, 2, 1, 0, 4, 5, **6**, 7, 9]
[3, 2, 1, 0, 4, **5**, 6, 7, 9]
[3, 2, 1, 0, **4**, 5, 6, 7, 9]
[0, 2, 1, **3**, 4, 5, 6, 7, 9]
[0, 1, **2**, 3, 4, 5, 6, 7, 9]
[0, **1**, 2, 3, 4, 5, 6, 7, 9]
[**0**, 1, 2, 3, 4, 5, 6, 7, 9] return minumum = 1.

(P223) exe. 9.3.3 Show how quicksort can be made to run in $O(n\lg n)$ time in the worst case, assuming that all elements are distinct. (write pseudocode for this quicksort variant.)

In PARTITION, randomly picking a pivot for array splitting cannot guarantee the worst case running in $O(n\lg n)$ time. But we can choose the median specifically each time, by a linear algorithm SELECT, as the pivot for array splitting. Which can make sure the array is split into two subarrays with same sizes or one might have one more element than another.

Let $T(n)$ be the total running time needed by the modified QUICKSORT given an array with n elements. Then, according to above new PARTITION based on SELECT (say the name of this method is SELECTED_PARTITION), we can derive the recurrence as follow:
$T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$.
And $n^{\log_2 2} = n^1 = f(n)$, thus, case 2 of Master Theorem applies. That is,
$T(n) = \Theta(n\lg n)$.

Pseudocode

```
SELECTED_PARTITION (A, p, r)
    $midIndex = \lceil \frac{r-p+1}{2} \rceil$
    $i = SELECT(A, p, r, midIndex)$// return median in subarray A[p..r]
    $EXCHANGE\, A[i]\, with\, A[r]$
    $return\, PARTITION(A, p\, r)$
```

---

exe. 9.3.8 Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order.

Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Extended binary search.
First, we need to make sure which array the median is in, X or Y.
After we have determined the median is in either X or Y, we can search for it in certain array using binary search, which is a linear searching algorithm.

```
TWO_ARRAY_MEDIAN (X, Y)
  $n \leftarrow length[X]$ // n also equals length[Y]
  $median \leftarrow FIND\_MEDIAN(X, Y, n, 1, n)$
  if $median = NOT\_FOUND$
    else $median \leftarrow FIND\_MEDIAN(X, Y, n, 1 n)$
  return $median$

FIND_MEDIAN (A, B, n, low, high)
  if $low > high$
    then return $NOT\_FOUND$
  else
    $k \leftarrow \lfloor \frac{low + high}{2} \rfloor$
    if $k = n$ and $A[n] \leq B[1]$
      then return $A[n]$
    elseif $k < n$ and $B[n - k] \leq B[n - k + 1]$
      then return $A[k]$
    elseif $A[k] > B[n - \ + 1]$
      then return $FIND\_MEDIAN(A, B, n, low, k - 1)$
    else
      return $FIND\_MEDIAN(A, B, n, k + 1, high)$
```

(P224) prob. 9.1 Largest i numbers in sorted order.

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case runing time, and analyze the running times of the algorithms in terms of n and i. (express the run time of each solution in terms of both n and i.)

a. Sort the numbers, and list the i largest.

Sort the numbers using merge sort or heapsort, which take $\Theta(n\lg n)$ running time for worst case. After sorted, listing the ith largest element takes $\Theta(i)$ time, by outputing the ith element directly.
Thus, the total running time for worst case is $\Theta(i + n\lg n) = \Theta(n\lg n)$, since $i \leq n$.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i trimes.

Implement the priority queue as a heap. Build the heap using BUILD-HEAP, which takes $\Theta(n)$ time, then call HEAP-EXTRACT-MAX i times to get the ith largest elements, in $\Theta(i\lg n)$ worst-case time, and store them in reverse order of extraction in the output array. The worst-case extraction time is $\Theta(i\lg n)$ because

- i extractions from a heap with $O(n)$ elements takes $i * O(\lg n) = O(i\lg n)$ time, and
- half of the i extractions are from a heap with $\geq \frac{n}{2}$ elements, so those $\frac{i}{2}$ extractions take $(\frac{i}{2}\Omega(\lg\frac{n}{2})) = \Omega(i\lg n)$ time in the worst case.

Total worst-case running time would be $\Theta(n + i\lg n)$.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the ith largest numbers.

Use the SELECT algorithm to find the ith largest number in $\Theta(n)$ time. Partition around that number in $\Theta(n)$ time. Sort the i largest numbers in $\Theta(i\lg i)$ worst-case time (with merge sort or heapsort).
Total worst-case running time: $\Theta(n + i\lg i)$.

---

prob. 9.2 Weighted median.

For n distinct elements $x_1, x_2, \cdots, x_n$ with positive weights $w_1, w_2, \cdots, w_n$ such that $\underset{i}{W}^{n}_{i=1} \ = 1$, the *weighted*(*lower*)*median* is the element x_k satisfying ${\sideset{}{{*x*{i}\,\lt\,x*{k}}}\sum w{i}

\,\lt\,\frac{1}{2}}*and*{\sideset{}{{*x*{i}\,\gt\,x*{k}}}\sum w{i} \,\leq\,\frac{1}{2}}$

b. Show how to compute the weighted median of n elements in $O(n\lg n)$ worst-case time using sorting.

We first sort the n elements into increasing order by x_i values. Then we scan the array of sorted x_i's, starting with the smallest element and accumulating weights as we scan, until the total exceeds $\frac{1}{2}$. The last element, say x_k, whose weight caused the total to exceed $\frac{1}{2}$, is the weighted median. Notice that the total weight of all elements smaller than x_k is less than $\frac{1}{2}$, because x_k was the first element that caused the total weight to exceed $\frac{1}{2}$. Similarly, the total weight of all elements larger than x_k is also less than $\frac{1}{2}$, because the total weight of all the other elements exceeds $\frac{1}{2}$.

The sorting phase can be done in $O(n\lg n)$ worst-case time (using merge sort or heapsort), and the scanning phase takes $O(n)$ time. The total running time in the worst case, therefore, is $O(n\lg n)$.

c. Show how to compute the weighted median in $\Theta(n\lg n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The *post − officelocationproblem* is defined as follows. We are given n points $p_1, p_2, \cdots, p_n$ with associated weights $w_1, w_2, \cdots, w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$, where $d(a, b)$ is the distance betwen points a and b.

We find the weighted median in $\Theta(n)$ worst-case time using the $\Theta(n)$ worst-case median algorithm. (Although the first paragraph of the section only claims an $O(n)$ upper bound, it is easy to see that the more precise running time of $\Theta(n)$ applies as well, since steps 1, 2, and 4 of SELECT actually take $\Theta(n)$ time.)

The weighted-median algorithm works as follows. If $n \leq 2$, we just return the brute-force solution. Otherwise, we proceed as follows. We find the actual median x_k of the n elements and then partition around it. We then compute the total weights of the two halves. If the weights of the two halves are each strictly less than $\frac{1}{2}$, then the weighted median is x_k, and the search continues within the half that weighs more than $\frac{1}{2}$.

Pseudocode, takes as input a set $X = \{x_1, x_2, \cdots, x_n\}$.

In [ ]:

```

```