CSCI 3104 Fall 2017 HW 4
Austin Griffith (9/24)

---

1. *You are given n metal balls B1, . . . , Bn, each having a different weight. You can compare the weights of any two balls by comparing their weights using a balance to find which one is heavier.*

   (a) *Consider the following algorithm to find the heaviest ball: i. Divide the n balls into n 2 pairs of balls. ii. Compare each ball with its pair, and retain the heavier of the two. iii. Repeat this process until just one ball remains. Illustrate the comparisons that the algorithm will do for the following n = 8 input: B1 : 3, B2 : 5, B3 : 1, B4 : 2, B5 : 4, B6 : 1 2 , B7 : 5 2 , B8 : 9 2*

   n = 8

   ```
   1st itt: [3 , 5]    [1 , 2]    [4 , 1/2]  [5/2 , 9/2]
   2nd itt:  [5 , 2]    [4 , 9/2]
   3rd itt:  [5 , 9/2]
   ```

   It took 7 total comparisons and 3 itterations to find the heaviest ball = 5 = B_2

   (b) *Show that for n balls, the algorithm (1a) uses at most n comparisons.*

   If you have n balls in the set the algorithm must make n comparisons at most. If the set of balls is of size 8 (n=8) then n/2 = 8/2 = 4 so 3 iterations of comparisons will occur which is less than n=8.

   ```
   For n = 8 so that log_2 n = 2^i = n will result in 2^i = 8
   2^3 = 8  for the levels of comparison in the algorithm
   We also see that n/(2^i) = (8/(2^3)) + (8/(2^2)) + (8/(2^1))
   = 1 + 2 + 3 = 6 < 8  (n=8)
   n over 2^i comparisons for the set must be < 8
   ```

(c) *Describe an algorithm that uses the results of (1a) to find the second heaviest ball, using at most log2 n additional comparisons. There is no need for pseudocode; just write out the steps of the algorithm like we have written in (1a). Hint: if you follow sports, especially wrestling, read about the repechage.*

i) divide n balls into n/2 pairs of balls
ii) copare each ball with its pair and retain the heavier of the two
iii) repeat process is until one ball (heaviest ball) is remains keeping track of the elements that the heaviest found ball was compared too.
iv) compare those elements that were compared to the heaviest ball in pairs until only one ball is left
v) 2nd heaviest ball is found

(d) *Show the additional comparisons that your algorithm in (1c) will perform for the input given in (1a).*

```
It took log_2 n levels to find the heaviest ball when n = 8
which (log_2 8 = 3) so 3 iterations of comparisons.
```

The additional comparisons that will be performed are as follows:

```
1st comparison  [3 , 2]
2nd comparison  [3 , 9/2]
```

B_8 : 9/2 is the 2nd heaviest ball

2. *An array is almost k sorted if every element is no more than k positions away from where it would be if the array were actually sorted in ascending order.*

   (a) *Write down pseudocode for an algorithm that sorts the original array in place in time n k log k. Your algorithm can use a function sort(A, l, r) that sorts the subarray A[l], . . . , A[r] Note: you will be working on this problem in recitation this week.*

   Goal: O(nk log k) run time
   Sort(A[ ], l, r) —-> sorts A[l]....A[r]

```
A = [1 4 2 3 5 15 5]
k=2
AlmostSort(A[ ] , k)
{
    limit = A.length - k
    for(i=0 , i <= limit , i++ )  //loops runs n-k imes
    {
        sort(A[], i , i + k)  //runs in O(n log n) time
    }
}

sorts first 3 elements and then continues
1st itt: i=0  k=2  Sort(A[],0,2)
2nd itt: i=1  k=2  Sort(A[],1,3)
3rd itt: i=2  k=2  Sort(A[],2,4)
4th itt: i=3  k=2  Sort(A[],3,5)
Final sorted array:  A = [1 2 3 4 5 15]
```

3. *Consider the following strategy for choosing a pivot element for the Partition subroutine of QuickSort, applied to an array A.*

   (a) *If the element m obtained as the median of S is used as the pivot, what can we say about the sizes of the two partitions of the array A?*

   The worst possible case is when n = 16 and the partitions are 3 and n = 16 - 3
   The Best case is when the partions are $\sqrt{n}$ and n - $\sqrt{n}$

   (b) *How much time does it take to sort S and find its median? Give a bound.*

   Insertion sort takes $\Theta(n^2)$ time to sort an array of n elements
   The subarray S has floor(2$\sqrt{n}$) elements in its array
   Thus to sort S will take $\Theta(\sqrt{n}^2) = \Theta(n)$ time and to find median will take one operation n/2.

   (c) *Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.*

   There will be two cases for the two partitions:
   T(n) = 2T[$\sqrt{n}$/n] + Θ(n)
   (n) = 2T[n-$\sqrt{n}$/n] + Θ(n)
   Where a = 2 because the array is divided into two subarrays, b = $\sqrt{n}$ and n-$\sqrt{n}$ the size of the two partitions, and f(n) = Θ(n) as proven in previous portion.

4. *Let A and B be arrays of integers. Each array contains n elements, and each array is in sorted order (ascending). A and B do not share any elements in common. Give a O(lg n)-time algorithm which finds the median of A U B and prove that it is correct. This algorithm will thus find the median of the 2n elements that would result from putting A and B together into one array. (Note: define the median to be the average of the two middle values of a list with an even number of elements.)*

```
// returns median of A[] and B[]
//Assumes both A[] & B[] are sorted arrays and both have n elements
calcMedian(A[], B[], n)
{
    if(n == 0)  //nothing in array
    {print "invalid input}
    if(n == 1)   //one thing in array
    {return A[0] or B[0]}
    if(n == 2)  //two things in array
    {return max(A[0], B[0]) + min(A[1], B[1]) / 2 }  //end of base cases

    if(n % 2 == 0) //even case
    {
        median1 = (A[n/2] + A[n/2-1]) / 2
        median2 = (B[n/2] + B[n/2-1]) /2
    }
    else  //odd case
    {
        median1 = A[n/2]
        median2 = B[n/2]
    }
    if(median1 == median2)  //case where medians are equal
    { return median1 }

    if(median1 < median2)
    {
        if(n % 2 == 0)
        {
            return calcMedian(A[median1...end], B[], n- n/2 +1)
        }
        return calcMedian(A[median1...end], B[], n - n/2)
    }

    if(n % 2 == 0)
    {
        return calcMedian(B[median2...end], A[], n - n/2 + 1)
    }
    return calcMedian(B[median2...end, A[], n - n/2])
}
```

This algorithms is a modification of Binary search algorithm which runs in O(log n) time so therefore this algorithm runs in O(log n) time complexity. The way this is done is by first checking the base cases of 0, 1 or 2 values in each array A and B. The algorithm then calculates the median of both input arrays and then checks if they are equal to return that value. If the median is not found thus far the algorithm then uses the first calculated median of the two arrays and and then starts cutting them in sub-arrays and uses the technique that Binary search uses. From there the function is recursively called until the median is found.