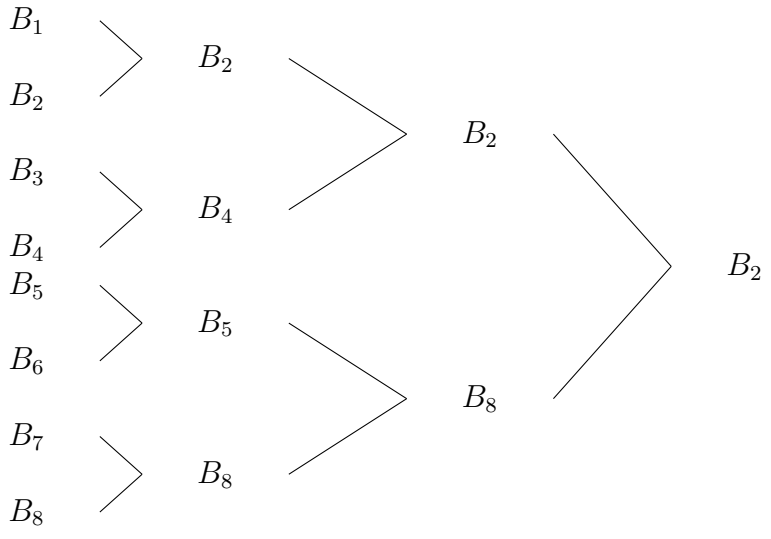
CSCI 3104, Fall 2017 Problem Set 4
Adam Ten Hoeve (10/02)
adam.tenhoeve@colorado.edu
Collaborated with Erin Ruby and Andrea DeVore

Problem 1

(10 pts) You are given n metal balls $B_1, \cdots, B_n$, 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 $\frac{n}{2}$ 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:

$$B_1 : 3,\ B_2 : 5,\ B_3 : 1,\ B_4 : 2,\ B_5 : 4,\ B_6 : \tfrac{1}{2},\ B_7 : \tfrac{5}{2},\ B_8 : \tfrac{9}{2}$$



Based on the algorithm, Ball 2 is the heaviest ball.

(B) *Show that for n balls, the algorithm* `(1A)` *uses at most n comparisons.*

Total comparisons for `1A` is

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + \frac{n}{2^k}$$

There is 1 comparison at the end of the algorithm to find the heaviest ball, therefore the last comparison occurs when

$$\frac{n}{2^k} = 1$$

Thus

$$k = \log_2 n$$

So, the total comparisons can be written as

$$\sum_{k=1}^{\log_2 n} \left(\frac{n}{2^k}\right) = n \sum_{k=1}^{\log_2 n} \left(\frac{1}{2^k}\right) = n \sum_{k=1}^{\log_2 n} \left(\frac{1}{2}\right)^k$$

We can solve for an infinite sum of a geometric series, therefore we can find an upper bound for number of comparisons for ∞ balls

$$n \sum_{k=1}^{\log_2 n} \left(\frac{1}{2}\right)^k \leq n \sum_{k=1}^{\infty} \left(\frac{1}{2}\right)^k$$

Recall that

$$\sum_{k=1}^{\infty} \left(\frac{1}{2}\right)^k = \frac{a}{1-r}$$

where a = first element and r = the common ratio. In our case, a = $\frac{1}{2}$ and r = $\frac{1}{2}$. Thus:

$$n \sum_{k=1}^{\log_2 n} \left(\frac{1}{2}\right)^k \leq n \cdot \frac{\frac{1}{2}}{\frac{1}{2}}$$

$$n \sum_{k=1}^{\log_2 n} \left(\frac{1}{2}\right)^k \leq n$$

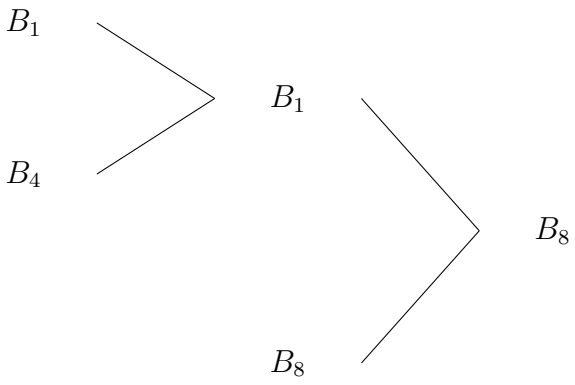
For n balls, the algorithm uses at most n comparisons

(C) *Describe an algorithm that uses the results of* `(1A)` *to find the second heaviest ball, using at most* $\log_2 n$ *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.*

After we find the heaviest ball, we will compare all of the balls that lost to the heaviest ball because at some point the heaviest ball must have compared against the second heaviest ball. We will start at the leaves of the tree with the ball the lost in the first comparison and compare that to the ball that lost in the second comparison. The heavier of those two will then compare to the next loser, and so on.

(D) *Show the additional comparisons that your algorithm in* `(1C)` *will perform for the input given in* `(1A)`.

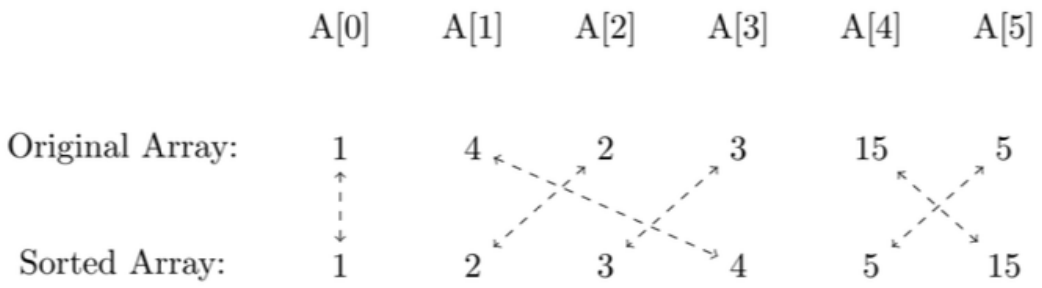
From part A, we know that B_2 is the heaviest ball. We now have to compare all of the balls that lost to B_2.



B_8 is the second heaviest ball.

Problem 2

(10 pts) 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.
As an example, here is an almost 2-sorted array:



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

```
moreSorting(A, l, r, k)
    for (i=0, i < r-k, i++)
        sort (A, i, i+k)
```

Assuming that the sort function has a runtime of $\Theta(n \log n)$, then we can reach a sorting algorithm of $\Theta(nk \log k)$ by simply running the sort algorithm through a for loop. Because we are running the for loop from 0...r-k, we can express this as n because it is roughly the length of the array. The sort function then has an order $\Theta(k \log k)$ because it sorts a subarray for the length of k, rather than the entire n of the larger array. By putting these two together processes together, we come to the conclusion that the algorithm runs on order $\Theta(nk \log k)$.

Problem 3

(20 pts) Consider the following strategy for choosing a pivot element for the `Partition` *subroutine of QuickSort, applied to an array A.*

- *Let n be the number of elements of the array A.*
- *If $n \leq 15$, perform an Insertion Sort of A and return.*
- *Otherwise, choose $2floor(\sqrt{n})$ elements at random from n; let S be the new list with the chosen elements, sort the list S using Insertion Sort and use the median m of S as a pivot element, partition using m as a pivot, and carry out QuickSort recursively on the two parts.*

(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?*

To determine the sizes of the subarrays, we must look at the worst case behavior of the algorithm. The worst case behavior would be that the random elements selected by the algorithm would be the first/last $2 * floor(\sqrt{n})$ elements of the array. The algorithm would then sort those elements and find the median m of S. By partitioning A at A[m] into two subarrays , we find that the two subarrays have lengths of $floor(\sqrt{n})$ and $n - floor(\sqrt{n})$. Becuase this is the worst possible behavior, all other subarrays would be larger than the smallest array in this case. Therefor, the sizes of the partitions must be at least $floor(\sqrt{n})$.

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

We are using insertion sort to sort S, so we know that it runs at n^2 because there is a nested loop in the algorithm. Finding the median value takes the two middle values and divides by 2, which has a constant run time. Therefor, the the run time of the algorithm is $\Theta(n^2) + C$.

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

$$T(n) = T(\sqrt{n}) + T(n - \sqrt{n}) + \Theta(n)$$

This is the recurrence relation because, in the worst case, the algorithm partitions A into two subarrays, one of size $\sqrt{n}$ and the other of size $n - \sqrt{n}$. Therefor the recurrence has to account for the varrying sizes of the subarrays.

Problem 4

(20 pts) 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 $\mathcal{O}(\log n)$-time algorithm which finds the median of $A \cup B$ 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.)

```
median(A, n)
    if n % 2 == 0
        return floor((A[n/2] + A[n/2+1]) / 2)
    else
        return A[n/2]

findMedian(A, B, n)
    if n == 1
        return floor((A[0] + B[0]) / 2)
    else if n == 2
        return floor((max(A[0],B[0]) + min(A[1],B[1])) / 2)
    mA = median(A, n)
    mB = median(B, n)
    if mA < mB
        return findMedian(A[mA...n], B[0...mB], n)
        // First half of A or last half of B
    else
        return findMedian(A[0...mA], B[mB...n], n)
        // First half of B or last half of A
```

Base Cases:

1. If n = 1, then returns the average of the value in each array because the median is the middle number between the arrays.
2. If n = 2, then returns the average between the maximum of the first element in A or B and the minimum of the second element.

Inductive Hypothesis: Assume the algorithm is true for any array of size n where $2 < n \leq k$.

1. If the median of A is less than the median of B, then the middle values of A are less than the middle values of B. Therefor the median of the union array must be from the last half of A and the first half of B. We then recursively call the function with these new parameters.

2. If the median of A is greater than the median of B, then the middle values of A are greater than the middle values of B. Therefor the median of the union array must be from the first half of A and the last half of B. We then recursively call the function with these new parameters.

This function is $\mathcal{O}(\log n)$ because it runs similar to binary search, which also has a runtime of $\mathcal{O}(\log n)$.