4.4.2)

Design a decrease-by-half algorithm for computing log2 n and determine its time efficiency.

Algorithm ComputeLog(n)

// Input: positive int n > 1

// Output: log b2 of n

If n != 1 return ComputeLog(floor(n/2)) + 1

Else return 0

Since A(1) = 0

A(n) = A(floor(n/2)) + 1

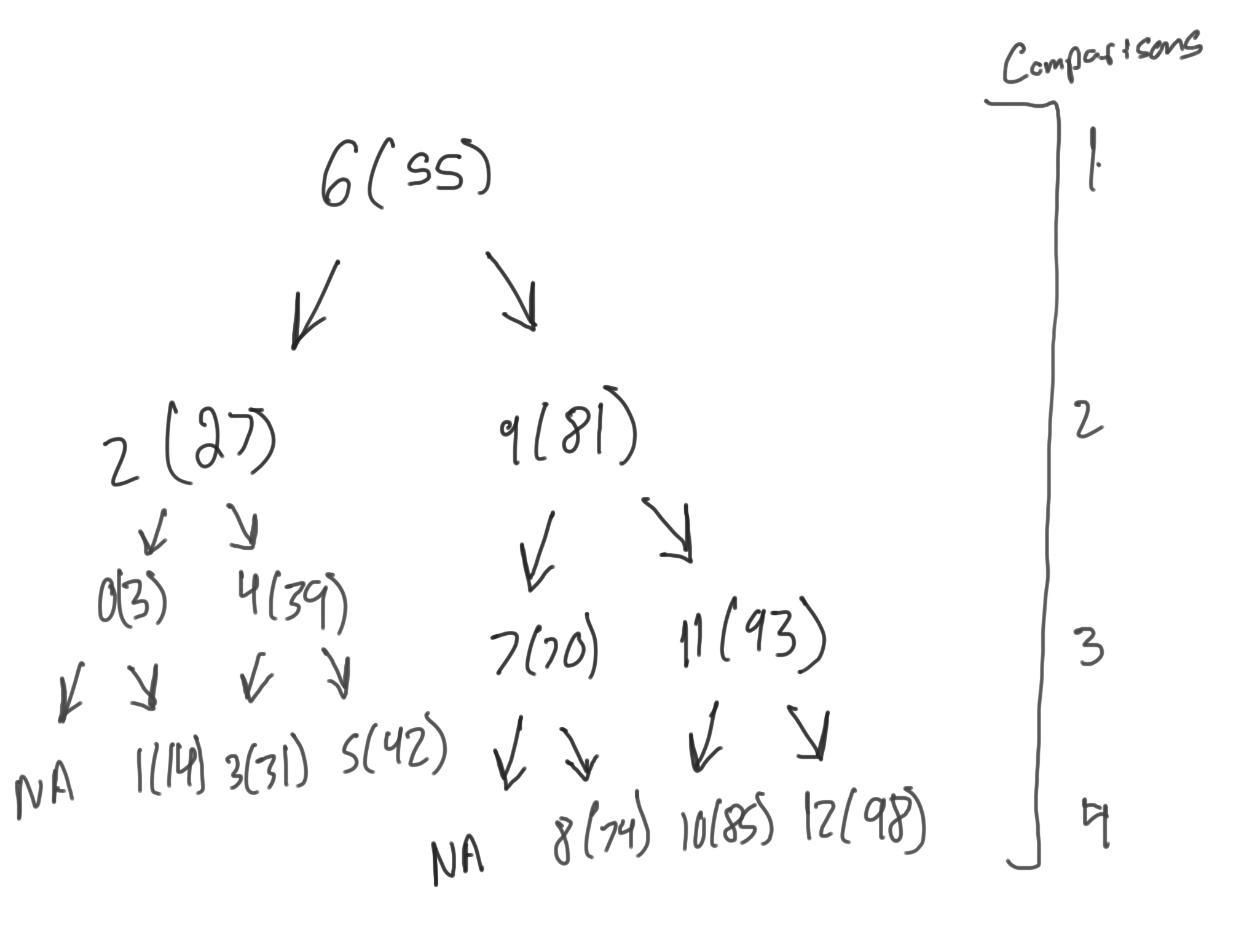
Making the time efficiency of A(n) = floor (log2n) in the domain of Θ(log n)

4.4.3)

1. What is the largest number of key comparisons made by binary search in searching for a key in the following array?  
      
   [3 14 27 31 39 42 55 70 74 81 85 93 98]

The largest number of key comparisons is 4, this is because binary search has a big O of log base 2 of n making the formula for this ceil(log b2 14) (13 plus the first comparison). Which makes the largest number of searches equal to 4.

1. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.



4.4.5)

The binary search algorithm would be a poor and inefficient way to search a sorted linked list. Unlike arrays, which allow for any element in the array to be accessed in a set time, finding the middle of a linked listed requires parsing the whole linked list which gives it Θ(n). This means that using linked lists in binary search would take more time and be less efficient than using an array.

4.4.8)

1. What design technique is this algorithm based on?

This is a decrease-by-a-constant-factor algorithm

1. Set up a recurrence for the number of key comparisons in the worst case.

C(n) = 2 + C(n/3) C(1) = 1 for n = 3k, k > 0

1. Solve the recurrence for n = 3k.

C(3k) = 2 + C(3k/3 = 2 + C(3k - 1)

C(3k - 1) = 2 + C(3k - 2)

…

C(3k - (k + 1)) = 2 + C(3k - k)

C(3k - k) = 1

= 2 log3(n + 1)

1. Compare this algorithm’s efficiency with that of binary search.

Since the previous equation is assuming a worst case scenario for the ternary algorithm, the worst case for binary search, log2(n + 1), will be assumed as well:

2 log3(n + 1) = 2 \* log2 n / log2 3 + 1 = (2 / log2 3 ) \* log2 n + 1

2 > (2 / log2 3 ) so we can assume that binary search will consistently be faster

4.4.10)

1. Write pseudo code for the divide-into-three algorithm for the fake-coin problem. Make sure that your algorithm handles properly all values of *n*, not only those that are multiple of 3.

Assuming the fake coin is lighter

If n = 1 the coin is fake

Split coins into three piles ⌈n/3⌉, ⌈n/3⌉, and ⌈n/3⌉

Weigh two piles

If they weigh the same continue with the third pile

Else continue with the lighter pile

1. Set up a recurrence relation for the number of weighings in the divide-into-three algorithm for the fake-coin problem and solve it for *n* = 3k.

C(n) = C(⌈n/3⌉) + 1 for n > 1 C(1) = 0

C(3k) = C(3k/3) + 1 = C(3k - 1) + 1

C(3k - 1) = C(3k - 2) + 1

…

C(3k - (k + 1)) = C(3k - k) + 1

C(3k - k) = 0

= log3(n)

1. For large values of *n*, about how many times faster is this algorithm than the one based on dividing coins into two piles? Your answer should not depend on *n*.

log2n / log3n = log2n / log32 log2n = 1 / log32 = 1.5849625

4.5.2) Apply quickselect to find the median of the list of numbers

9 12 5 17 20 30 8 n = 7 k = ⌈7 / 2⌉ = 4

partition

pivot = 9

s i

9 12 5 17 20 30 8

s i

9 12 5 17 20 30 8

9 5 12 17 20 30 8 swap since i < pivot

s i

9 5 12 17 20 30 8

s i

9 5 12 17 20 30 8

s i

9 5 12 17 20 30 8

s i

9 5 12 17 20 30 8

9 5 8 17 20 30 12 swap since i < pivot

s

8 5 9 17 20 30 12 swap A[l], A[s]

return s = 2

s < 4 - 1 use right half of list

17 20 30 12

pivot = 17

s i

17 20 30 12

s i

17 20 30 12

s i

17 20 30 12

17 12 30 20 swap since i < pivot

s

12 17 30 17 swap A[l], A[s]

return s = 4

s == k - 1 return A[s] = A[4] = 12

4.5.8)

* 1. If the key is a left, make a pointer to the parent null
  2. If it has 1 child, make the pointer to its parent point to its child
  3. If it has 2 children, find the smaller node on the right of the key, switch their positions, then remove it using one of the above cases
  4. This would not be a variable size decrease algorithm

1. The worst case is , as in the would need to follow n-2 pointer when you are finding the smallest right subtree

4.5.10)

Player 1 loses the game iff n % (m+1) = 1, otherwise the player can win by taking (n-1) mod (m+1) chips. If they make any other move they risk losing the game.