Adina Edwards
8th Oct. 2017
CS325

HW 2

1. Give the asymptotic bounds for $T(n)$ in each of the following recurrences. Make your bounds as tight as possible and justify your answers. Assume the base cases $T(0) = 1$ or $T(1) = 1$.
   a. $T(n) = 2T(n-2)+1$

      Muster Theorem

      $a = 2,\ b = 2,\ n^d = 1$ therefore $d = 0$. The answer is in the form $O(n^d a^{n/b})$.

      Because $f(n) = \theta(n^d)$, we know the shortest this recurrence could take is $\Omega(n^d)$ when $2T(n-2) = 0$ at $n = 2$. This means we know the lower bound on $T(n)$ in addition to the upper bound. The answer is $T(n) = \theta(2^{\frac{n}{2}})$.
   b. $T(n) = T(n-1) + n^3$

      Muster Theorem

      $a = 1,\ b = 1,\ d = 3$ The answer is in the form $O(n^{d+1})$.

      The shortest time this algorithm could take is $\Omega(n^4)$ as it has to follow the $f(n)$ and $T(n-1)$ terms. Therefore the answer is $T(n) = \Theta(n^4)$.
   c. $T(n) = 2T(\frac{n}{6}) + 2n^2$

      Master Theorem, case 3

      $n^{log6(2)} = n^{.387} < 2n^2$

      If $2f(\frac{n}{6}) \leq cf(n)$ then $T(n) = \theta(n^2)$.

      $2 * 2(\frac{n}{6})^2 \leq c * 2n^2$

      $\frac{n^2}{18} \leq cn^2$

      $\frac{1}{18} \leq c$

      Therefore $T(n) = \theta(n^2)$.
2. The quaternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-fourth.
   a. Verbally describe and write the pseudocode for the quaternary search algorithm.

      The quaternary search algorithm will take a set of size n and will recursively divide it into four parts of equal size. Once at the base case of $n = 1$, it will return true if the value to be found is in the array, and false if it is not.

      QuaternarySearch(A[0….n], b)

```
If (sizeof(A) == 0)
        return false
if(sizeof(A) == 1) //subproblem of size 1
        if(A[i] == b)
                Return true
        Else
```

```
                    Return false
if(b == A[n/4] || b == A[n/2] || b == A[3n/4])
        Return true
Else if (b < A[n/4]) return QuaternarySearch(A[0...n/4], b)
Else if (b < A[n/2]) return QuaternarySearch(A[n/4+1...n/2], b)
Else if(b < A[3n/4]) return QuaternarySearch(A[n/2+1...3n/4], b)
Else return QuanternarySearch(A[3n/4+1...n], b)
```

Recursive stuff in blue Nonrecursive stuff in green

b. Give the recurrence for the quaternary search algorithm.
   $T(n) \leq T(\frac{n}{4}) + c$
   $T(n)$ is bounded above by $T(\frac{n}{4}) + c$. There is some cost, c, to perform all the nonrecursive stuff such as checking to make sure the size of the input array is greater than zero. The cost to recurse through the array is $T(\frac{n}{4})$ as we divide the problem space into four different parts and recurse over one and only one of those parts. Because it is possible to find the value b early on, possibly before even recursing a single time, we cannot state that the time is equal to this bound, but is less than or equal to this bound.
c. Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the quaternary search algorithm compare to that of the binary search algorithm?
   $T(n) \leq T(\frac{n}{4}) + c$
   Using the Master Theorem, $a = 1, b = 4$ and $f(n) = cn^0$.
   $n^{log4(1)} = n^0$ so we have Case 2 and $T(n) = O(lgn)$.

   The first case will be a problem of size n with cost c. The next case will be a problem of size n/4 also with a nonrecursive cost c. In the worst case, we recurse until we reach a subproblem of size zero. At this point, the height of the tree will be lgn. Because this algorithm could end before reaching the full height of the tree, we cannot use θ. Instead, we know the maximum time it could take, which uses O form.

   Therefore, $T(n) = O(lgn)$.
   This is the same running time as binary search. It probably provides little value (other than educational) to use quaternary search over binary search as the algorithm is more complex and therefore more likely to be implemented incorrectly.

3. Design and analyze a divide and conquer algorithm that determines the minimum and maximum value in an unsorted list (array).
   a. Verbally describe and write pseudocode for the min_and_max algorithm.

The base case happens when there is a single element. This element is one of three things: lower than the current minimum, higher than the current maximum, or neither. If the single element falls into either of the first two cases, we update the current minimum or maximum as appropriate.
If we are not in the base case, we should cut the problem size in half, and recurse on each subproblem. Each recursion returns a list of length two, where the first element is the smallest value found throughout that side of recursion and the second is the largest value found.
If the returned value from recursing down the left side of the array is lower than the current minimum, then the current minimum gets updated. If the returned value from recursing down the right side of the array is lower than the current minimum, then the current minimum gets updated. The same goes for the current maximum: it is compared to the returned values from the left and right sides and updated if there is a greater maximum.

```
min_and_max(A, low_i=0, high_i=n, currentMin=NULL, currentMax=NULL)
        if(low_i == high_i) //single element
                If currentMin > A[low_i]
                        currentMin = A[low_i]
                If currentMax < A[low_i]
                        currentMax = A[low_i]
        Else //more than one element
                Mid_i = low_i + (high_i-low-i)/2
                Left = min_and_max(A, low_i, mid_i, currentMin, currentMax)
                Right = min_and_max(A, mid_i+1, high_i, currentMin, currentMax)
                If (Left[0] < currentMin)
                        currentMin = Left[0]
                If (Right[0] < currentMin)
                        currentMin = Right[0]
                If (Left[1] < currentMax)
                        currentMax = Left[1]
                If (Right[1] < currentMax)
                        currentMax = Right[1]
        Return [currentMin, currentMax]
```

b. Give the recurrence.
   $T(n) = 2T(\frac{n}{2}) + \theta(1)$
c. Solve the recurrence to determine the asymptotic running time of the algorithm. How does the theoretical running time of the recursive min_and_max algorithm compare to that of an iterative algorithm for finding the minimum and maximum values of an array?

By the Master Theorem, case 1, $n^{log2(2)} = n^1 > n^0$. Therefore, $T(n) = \theta(n)$. In both the best case and the worst case, this algorithm will search through the entire array. This means we stay with theta notation.

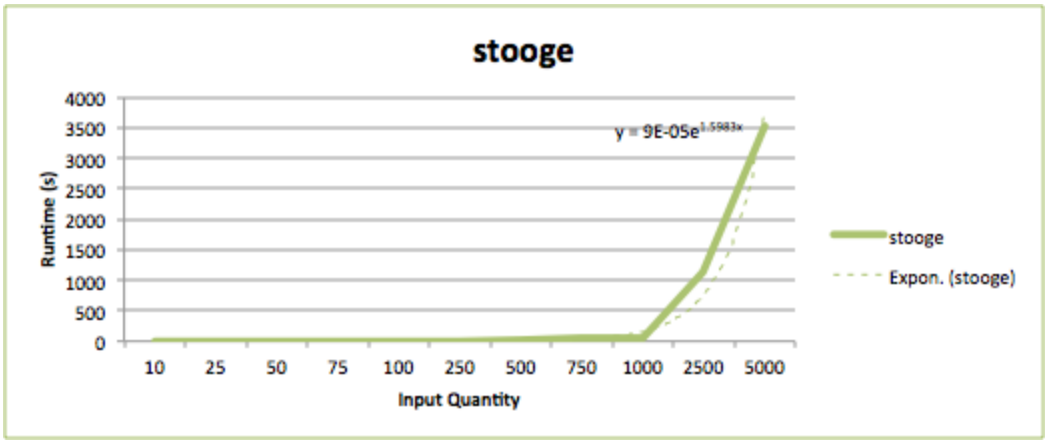
4. Consider the following pseudocode for a sorting algorithm.
    a. Verbally describe how the STOOGESORT algorithm sorts its input.
    Stoogesort divides the input up into three parts and recurses over each of them in a specific order. First, it sorts the lower third, then the upper third, and then the lower third again. By doing so, it can move large values up the array, and small values down the array in steps.
    b. Would STOOGESORT still sort correctly if we replaced $m = ceiling(\frac{2n}{3})$ with $m = floor(\frac{2n}{3})$? If yes, prove; if no, give a counterexample. (Hint: what happens when $n = 4$?)
    When $n = 4$, $m = floor(\frac{2*4}{3}) = 2$
    The first call would be on StoogeSort(A[0, 1]). The second call on StoogeSort(A[2, 4]). The third call on StoogeSort(A[0, 1]). At no point is there an opportunity for a potentially small value to leave the larger portion of A, nor for a large value to work its way out of the smaller part of the array. StoogeSort depends upon the ceiling function.
    c. State a recurrence for the number of comparisons executed by StoogeSort.
    $T(n) = 3T(\frac{2n}{3}) + \theta(1)$
    Each time StoogeSort recurses, it spawns three sub problems. Each subproblem is two thirds the size of the original problem. There is a constant time cost, which does not depend on the problem size, each time StoogeSort is called.
    d. Solve the recurrence to determine the asymptotic running time.
    By the Master Theorem, case 1, $n^{log\frac{3}{2}(3)} = n^{2.709} > n^0$ Therefore $T(n) = \theta(n^{2.709})$.
5. Implementation
    a. Implement StoogeSort.
    See submission on TEACH.
    b. Collect running time data. Use at least 7 different values of n where T(n)>0.

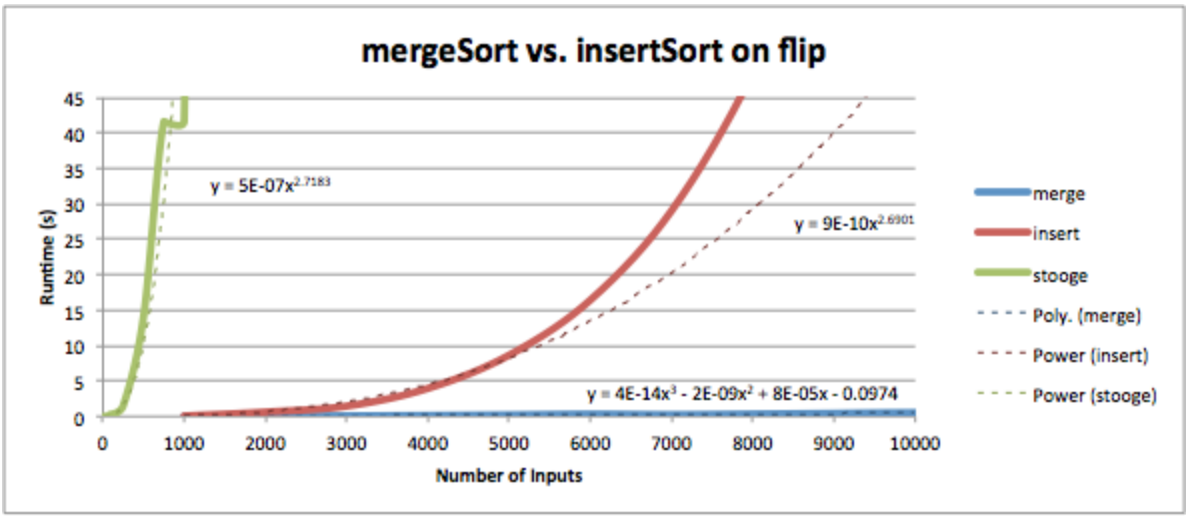
stoogeSort

Input Size	Runtime
10	0.000242949
25	0.002086878
50	0.01891613
75	0.061486959
100	0.176692963

250	1.534494877
500	14.11524582
750	41.6247611
1000	41.71062493
2500	1140.00299
5000	3530.880204

c. Plot the running time data on an individual graph. Plot the running time data on the same graph as the results from merge sort and insert sort.





d. Which type of curve best fits the StoogeSort data collected? How does the experimental running time compare to the theoretical running time?
The best fit curve for the data was an exponential function, when data points up to n=5000 are considered. This function grows more quickly than the theoretical

runtime. This difference could be accounted for by caching architectures and the use of python.