Conrad Lewin
CS 325
4/10/2016

1a) $T(n) = T(n-2)+n$

The recursion tree for T(*n*) is:

$$\begin{array}{ll}
\text{Lvl } 0 & cn \\
 & | \\
\text{Lvl } 1 & c(n\text{-}2) \\
 & | \\
\text{Lvl } 2 & c(n\text{-}4) \\
 & | \\
\text{Lvl } 3 & c(n\text{-}6) \\
 & | \\
 & . \\
 & . \\
 & . \\
\text{Lvl } \frac{n-1}{2} & \text{T}(1)
\end{array}$$

A cost pattern emerges at each level of the tree that can be used to decipher where the tree will end, namely that each node in the tree has a cost of $c(n-2d)$ where *d* is the depth of the node. Thus, in order to find out where the recursion bottoms out we can ignore the constant and solve for $n-2d = 1$.

$$n-2d = 1$$
$$n = 1+2d$$
$$n-1 = 2d$$
$$\frac{n-1}{2} = d$$

We now need to add up the costs of each recursive call in the tree by calculating $\sum_{d=1}^{\frac{n-1}{2}} c(n-2d)$.

$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{n-1}{2}\left(\frac{cn-2c+cn-2c\left(\frac{n-1}{2}\right)}{2}\right)$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{n-1}{2}\left(\frac{2cn-2c-c(n-1)}{2}\right)$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{n-1}{2}\left(\frac{2cn-2c-cn+c}{2}\right)$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{n-1}{2}\left(\frac{cn-c}{2}\right)$$

$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{cn^2-2cn+c}{4}$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{c(n^2-2n+1)}{4}$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = \frac{c}{4}(n^2-2n+1)$$
$$\sum_{d=1}^{\frac{n-1}{2}} c(n-2d) = O(n^2)$$

We can now check this assertion using the substitution method. Our guess is that T(n) = O(n^2) and so we must show that T(n) ≤ cn^2. Therefore, our inductive hypothesis is T(n-2) ≤ c(n-2)^2. Using substitution we get

$$T(n) \le c(n-2)^2+n$$
$$T(n) \le c(n^2-4n+4)+n$$
$$T(n) \le cn^2-4cn+4c+n$$

The residual, 4cn + 4c + n, is positive for all $c \ge 0$; however, T(n) ≤ cn^2 - 4cn + 4c + n ≤ cn^2 requires that $c \ge 1$ for the inequality to hold. Furthermore, we can prove that T(n) = Ω(n^2) using the same inductive process. This time, however, we must show that T(n) ≥ c(n - 2)^2 + n which is true for all $0 \le c < 1$ and all $n > 1$. Therefore, T(n) = Θ(n^2).

1b) $T(n) = T(n-1)+3$

The recursion tree for T(n) is:

Lvl 0 3
|
Lvl 1 T(n-1) = 3
|
Lvl 2 T(n-2) = 3
|
Lvl 3 T(n-3) = 3
|
.
.
.
Lvl $n-2$ T(n-d) = 3
|
Lvl $n-1$ T(1)

Since each recursive call is only reducing the size of the input by one 1, the point at which n = 1 can be calculated by solving *n* - *d* = 1, where *d* is the depth of a node, for *d*. We can find the last level of the tree thusly,

$$n-d = 1$$
$$n = 1+d$$
$$n-1 = d$$

The amount of work done in this tree is the sum of the cost at each level from 0 to *n* - 2, 3(*n* - 2 + 1) = 3(*n* - 1), plus a constant time operation at level *n* - 1. Thus,

$$3(n-1)+T(1) = 3n-3+c = O(n)$$

We can now check this assertion using the substitution method. Our guess is that T(*n*) = O(*n*) and so we must show that T(*n*) ≤ *cn*. Therefore, our inductive hypothesis is T(*n*-1) ≤ c(*n*-1). Using substitution we get

$$T(n) \le c(n-1)+3$$
$$T(n) \le cn-c+3$$

Note that the residual, *c* + 3, will be positive when *c* ≤ 3. Also, note that we can prove T(*n*) = Ω(*n*) using a similar inductive process in which T(*n*) ≥ *c*(*n* - 1) + 3 for all *c* > 3. Therefore, T(*n*) = Θ(*n*).

1c) $T(n) = 2T\left(\frac{n}{4}\right)+n$

Using the master theorem, *a* = 2, *b* = 4 and *f*(*n*) = *n* such that $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$. Since $n^{0.5} < n$, and furthermore, since $n^{0.5}$ is polynomially smaller than *f*(*n*), insofar as $n^{0.5+\varepsilon} < n$ where ε = 0.1, we need only satisfy the regularity condition $af\left(\frac{n}{b}\right) \le cf(n)$ for some constant *c* < 1. Thus,

$$2\left(\frac{n}{4}\right) \le cn$$
$$\frac{2n}{4} \le cn$$
$$\frac{1}{2}n \le cn$$

If we let *c* = 0.9, though there are other possibilities, then this inequality holds, the regularity conditions is satisfied and T(*n*) = Θ(*n*).

1d) $T(n) = 4T\left(\frac{n}{2}\right)+n^2\sqrt{n}$

Using the master theorem, *a* = 4, *b* = 2 and $f(n) = n^2\sqrt{n}$ such that $n^{\log_b a} = n^{\log_2 4} = n^2$. Since $n^2 <$ $n^2\sqrt{n} = n^{2.5}$, and furthermore, since n^2 is polynomially smaller than *f*(*n*), insofar as $n^{2+\varepsilon} < n$ where ε = 0.1, we need only satisfy the regularity condition $af\left(\frac{n}{b}\right) \le cf(n)$ for some constant *c* < 1.

Thus,

$$4\left(\frac{n}{2}\right)^{2.5} \le cn^{2.5}$$
$$4\left(\frac{n^{2.5}}{2^{2.5}}\right) \le cn^{2.5}$$
$$\approx 0.7071n^{2.5} \le cn^{2.5}$$

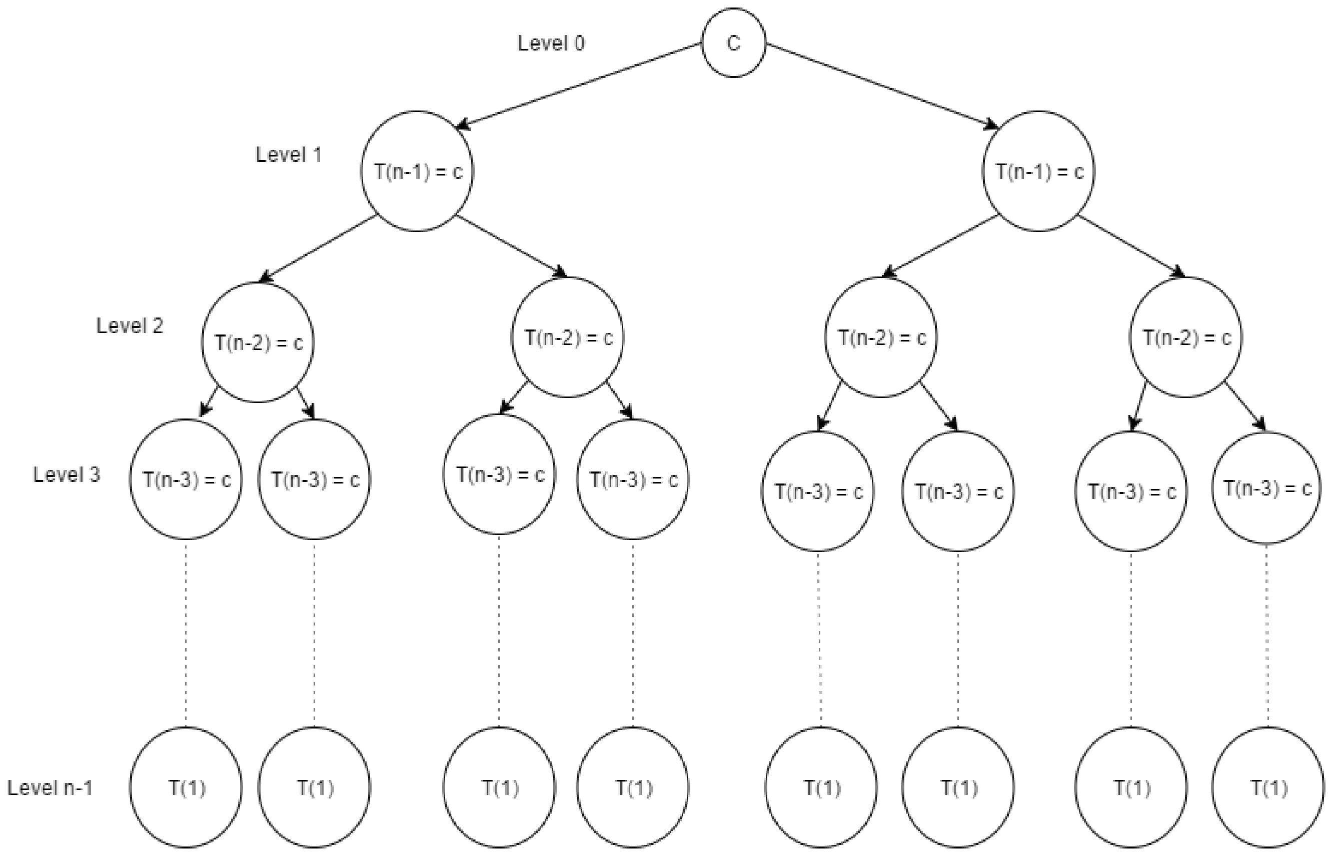
If we let c = 0.9, though there are other possibilities, then this inequality holds, the regularity conditions is satisfied and T(n) = Θ($n^2\sqrt{n}$) or T(n) = Θ($n^{2.5}$).

2) Algorithm A is $T(n) = 5T\left(\frac{n}{2}\right)+n$. Using the master theorem,

$$n^{\log_2 5}\approx n^{2.3}$$
$$n^{2.3-\varepsilon} \ge n \;\; where \;\; \varepsilon = 1$$
$$T(n) = \Theta(n^{2.3})$$

Algorithm B is $T(n) = 2T(n-1)+c$.

Using a recursion tree,



we can see that number of nodes at each level is 2^k where k is the depth and that the height of the tree is *n* - 1. Thus, the sum of costs at each level is

$$2^{n-1}T(1)+\sum_{k=0}^{n-1}2^k = 2^{n-1}c+\left(\frac{1+2^{n-1}}{1-2}\right)$$
$$2^{n-1}T(1)+\sum_{k=0}^{n-1}2^k = 2^{n-1}c-2^{n-1}-1$$
$$2^{n-1}T(1)+\sum_{k=0}^{n-1}2^k = 2^{n-1}(c-1)-1$$
$$2^{n-1}T(1)+\sum_{k=0}^{n-1}2^k = O(2^{n-1})$$

We can verify this by using the substitution method

$$guess:\ T(n) = O(2^{n-1})$$
$$show:\ T(n) \leq c(2^{n-1})$$
$$hypothesis: T(n-1) \leq c(2^{n-2})$$

$$T(n) \leq c(2^{n-2})+c$$
$$T(n) \leq 2^{n-2}c+c \leq c(2^{n-1})\ \ for\ \ all\ \ c \geq 0,\ n \geq 2$$

Furthermore, using a similar inductive process in which T(n) ≥ c(2^{n-2}) + c for all n < 2, we can show that T(n) = Ω(2^{n-1}). Therefore T(n) = Θ(2^{n-1}).

Algorithm C is $T(n) = 9T\left(\frac{n}{3}\right)+\Theta(n^2)$. Using the master theorem,

$$n^{\log_3 9} = n^2$$
$$n^2 = \Theta(n^2)$$
$$T(n) = \Theta(n^2\log_2 n)$$

The fastest algorithms in this set are A and C. However, algorithm A beats C with a running time of T(n) = Θ($n^{2.3}$) for all n > 1.

3) This algorithm makes 4 recursive calls, each one dividing *n* in half. Upon resolving each call, *n* work must be done to echo “Print”. The recurrence is then T(n) = 4T(n/2) + n. Using the master theorem,

$$n^{\log_2 4} = n^2$$
$$n^{2-\varepsilon} \geq \Theta(n)\ \ where\ \ \varepsilon = 0.9$$
$$T(n) = \Theta(n^2)$$

4) Pseudocode for ternary search:

```
ternarySearch(list,val,start,end):
    if(start < end)
        mid1 = (2 * start + end) / 3
        mid2 = (start + 2 * end) / 3

        if(val < list[mid1]):
            return ternarySearch(list,val,start,mid1)
        else if(val > list[mid1] and val < list[mid2])
            return ternarySearch(list,val,mid1,mid2)
        else if(val > list[mid2])
            return ternarySearch(list,val,mid2+1,end)
        else
            return true

    return false
```

The recurrence of this algorithm is T(*n*) = T(*n*/3) + c. Using the master theorem,

$$n^{\log_3 1} = n^0 = 1$$
$$1 = T(1)$$
$$T(n) = \Theta(\log_2 n)$$

This is similar to a recursive binary search algorithm, and the two differ only in the max number of comparisons they may have to make, insofar as this ternary search algortihm will have to make at most 3 comparisons, while a binary search will have to make at most 2, excluding in both cases the check that determines whether or not valid indices were passed to the function.

5a) Performing a cumulative merge on each successive array requires at worst *n* comparisons for the first two arrays, 2*n* comparisons for the newly merged array and the third array, 3*n* comparisons for the next pair of arrays, and so on until *k* - 1 merges have be completed. In this way, this algorithm will perform comparisons at an order of

$$n+2n+3n+\cdots+(k-1)n = n(1+2+3+\cdots+k-1)$$
$$n\sum_{i=1}^{k-1} i = n\left[k-1\left(\frac{1+k-1}{2}\right)\right]$$
$$n\sum_{i=1}^{k-1} i = n\left(\frac{k^2-k}{2}\right)$$
$$n\sum_{i=1}^{k-1} i = \left(\frac{k^2n-kn}{2}\right) = O(k^2n)$$

5b) To improve the run time of the algorithm examined in 5a, we can use a divide and conquer approach. Two recursive calls should be made at each level of recursion to divide the total number of arrays *k* in half until at last there is only one array left, which, by the virtue of already being sorted, can be said to be merged with itself. While the bottom level of the recursion incurs an operation at constant time, each prior level must do *kn* work, where *n* is the number of elements in each array, to merge various array pairs. Thus, the recurrence relation is

$$T(k) = 2T\left(\frac{k}{2}\right)+kn$$

Using the master theorem, this algorithm will have an asymptotic run time of

$$k^{\log_2 2} = k$$
$$k = kn \ (they\ are\ polynomial\ equivalents)$$
$$T(k) = \Theta(kn \log_2 kn)$$

6a) From page 27 of our textbook, we are given the worst case running time of insertion sort on a single list as $ak^2 + bk + c$ where *a*, *b* and *c* are constants. Sorting *n* sublists of *k* elements will then take

$$\frac{n}{k}(ak^2+bk+c) =$$
$$\frac{ak^2n+bkn+cn}{k} =$$
$$\frac{k(akn+bn)+cn}{k} =$$
$$akn+bn+\frac{cn}{k} = \Theta(kn)$$

time to complete.

6b) Using a divide and conquer approach, we can recursively divide the number of arrays in half with each call doing *kn* work where *k* is the number of elements and *n* is the number of arrays being merged. The recurrence relation for such an algorithm would then be, T(*n*) = 2T(*n*/2) + *kn*. Using the master theorem,

$$n^{\log_2 2} = n$$
$$n = kn \ (they\ are\ polynomial\ equivalents)$$
$$T(n) = \Theta(kn \ \log_2 n)$$

Thus, if we substitute *n* subarrays with *n*/*k* subarrays, we get

$$T\left(\frac{n}{k}\right) = \Theta\left(k\frac{n}{k} \log_2 \frac{n}{k}\right)$$
$$T\left(\frac{n}{k}\right) = \Theta\left(n \log_2 \frac{n}{k}\right)$$

6c) By examining the leading term of the modified merge sort at Θ(*nk* + *n* lg(*n*/*k*)) and the classic merge sort run time at Θ(*n* lg *n*), we can see that if Θ(*nk* + *n* lg(*n*/*k*)) ≤ Θ(*n* lg *n*) then *k* cannot be larger than lg *n*, otherwise this would invalidate the truth of the inequality. Let *k* = lg *n* so that,

$$\Theta\left(nk+n\log_2\frac{n}{k}\right)=$$
$$\Theta\left(n\log_2 n+n\log_2\left(\frac{n}{\log_2 n}\right)\right)=$$
$$\Theta\left(n\log_2 n+n\left(\log_2 n-\log_2\log_2 n\right)\right)=$$
$$\Theta\left(2n\log_2 n-n\log_2(\log_2 n)\right)=\Theta(n\log_2 n)$$

In this way, the largest value of *k* that will allow for the same running time as merge sort is $k = \lg n$.

6d) We know asymptotically that the modified merge sort runs faster than the classic implementation when $k < \lg n$. However, this kind of analysis ignores the influence of constant factors. Considering such conditions in a practical setting will inevitably inform which value of *k* is likely to produce a run time that is at least equivalent to the classic merge sort implementation. At most, *k* should never be larger than is acceptable for allowing insertion sort to run faster than a classic merge sort algorithm.

7) Pseudocode for divide and conquer min_max algorithm:

```
min_and_max(list, start, end)
    if(list.length == 1)
        return (list[0], list[0])
    else if(list.length == 2)
        if(list[0] < list[1])
            return (list[0], list[1])
        else
            return (list[1], list[0])

    tupleA = recMinMax(list, 0, list.length/2-1)
    tupleB = recMinMax(list,list.length/2,a.length)
    return compare(tupleA,tupleB)

compare(tupleA,tupleB)
    min = tupleA[0]
    max = tupleA[1]

    if(tupleA[0] > tupleB[0])
        min = tupleB[0]

    if(tupleA[1] < tupleB[1])
        max = tupleB[1]

    return (min,max)
```

This algorithm makes two recursive calls that divide the given input size in half with each call. At worst, 5 constant time comparisons will me made, thus all work done by each call takes some constant amount of time. The recurrence is therefore $T(n) = 2T(n/2) + c$. We can use the master method to decipher the asymptotic complexity of this algorithm.

$$n^{\log_2 2} = n$$
$$n^{2-\varepsilon} \geq cn^0 \ \ where \ \ \varepsilon = 1$$
$$af\left(\frac{n}{b}\right) \leq cf(n)$$
$$2\left(\frac{cn^0}{2}\right) \leq cn^0$$
$$c \leq c \ \ for \ \ all \ \ c \geq 0$$

Thus, T(*n*) = Θ(*n*). This is the same as the iterative solution which must run through *n* elements comparing each one to the currently established minimum and maximum values. The number of comparisons that must be made is where these algorithms differ, insofar as the iterative algorithm will make 2*n* comparisons, while the recursive algorithm will make at most 5.