Donovan Guelde
PS1
CSCI 5454
Collaborators: None

1. To show $(n+a)^b = \Theta(n^b)$, we must show both $(n+a)^b = O(n^b)$ and $(n+a)^b = \Omega(n^b)$

$(n+a)^b = O(n^b)$:
There exists some constant c s.t. $(n+a)^b \leq c(n^b)$
if $a < 0$ then $(n+a) < n$, so $(n+a)^b < c(n^b)$ where $c \geq 1$
if $a = 0$ then $(n+a) = n$, so $(n+a)^b = c(n^b)$ where $c \geq 1$
if $a > 0$ then $(n+a) > n$, so we must find constant c s.t. $(n+a)^b \leq c(n^b)$:
as n approaches infinity, $(n+a) \leq 2n$ because a is a constant, and regardless of a's value, 2n will eventually be larger than $(n+a)$. So:
$(n+a)^b \leq (2n)^b$
Therefore, $(n+a)^b \leq c(n^b)$ where $c = 2^b$
Therefore, $(n+a)^b \leq c(n^b)$ for all constants $c \geq 2$. $(n+a)^b = O(n^b)$

$(n+a)^b = \Omega(n^b)$:
There exists some constant c s.t. $(n+a)^b \geq c(n^b)$
if $a = 0$ then $(n+a) = n$, so $(n+a)^b = c(n^b)$ where $c \geq 1$
if $a > 0$ then $(n+a) > n$, so $(n+a)^b > c(n^b)$ where $c \geq 1$
if $a < 0$ then $(n+a) < n$, so we must find constant c s.t. $(n+a)^b \geq c(n^b)$:
as n approaches infinity, $(n+a) \geq \frac{n}{2}$ because a is a constant, and regardless of a's value, $(n+a)$ will eventually be larger than $\frac{n}{2}$. So:
$(n+a)^b \geq c(n)^b$ for all c s.t. $c \leq (\frac{1}{2})^b$

As $(n+a)^b = \Omega(n^b)$ and $(n+a)^b = O(n^b)$, it follows that $(n+a)^b = \Theta(n^b)$

2. $1 = (n^{\frac{1}{n^{logn}}}) \leq (2^{log*n}) \leq n \leq nlog(n) \leq n^2 \leq (\sqrt{2})^{logn} \leq (\frac{3}{2})^n$
notes:
1 is, of course, linear
$(n^{\frac{1}{n^{logn}}})$ approaches 2 as n approaches infinity, so this expression is also linear
(2^{log*n}) grows extremely slowly, but does grow faster than linearly. When $n = 10^6$, $2^{log*n} = 16$
n, $nlog(n)$, and n^2 are all textbook examples of big O ordering.
$(\sqrt{2})^{logn}$ and $(\frac{3}{2})^n$ both exhibit exponential growth, but since $n > logn$ for all $n > 1$, $(\frac{3}{2})^n$ grows faster than $(\sqrt{2})^{logn}$

3.a. $T(n) = T(n-1) + n, T(1) = 1$
Expanding the recursion results in:

$T(n) = T(n-1) + n$
$T(n-1) = T(n-2) + (n-1)$
$T(n-2) = T(n-3) + (n-2)$
$T(n-3) = T(n-4) + (n-3)$
...
$T(2) = T(1) + 2$
$T(1) = 1$

It quickly becomes apparent that $T(n) = T(n-1) + n$ expands to:
$\sum_i^n (i) = n^2$

3.b. $T(n) = 2T(\frac{n}{2}) + n^3; T(1) = 1$
Using the Master Theorem, a = 2, b = 2, d = 3
$log_b a = log_2 2 = 1 < d$, so the recurrence grows at $O(n^3)$

4.a. Given an unsorted array U where all $U[i] > 0$ and $max(U) = k$:

```
S = [0]*k #initialize an array of zeroes of size k
for all i in U:
    S[i]+=1 # for every number in the unsorted array, increment the array S at
            index [number] by 1
for all j in S:
    while $S[j] > 0$,
    print j
    S[j] -=1
```

4.b. By starting with the assumptions that all items to be sorted are between(or equal to) 1 and k, we can instantly ignore the possibility of numbers falling outside this range. This allows us to set up a data structure of size k, rather than worrying about values of unknown size. By doing so, we are no longer forced to make comparisons between values, so the $nlog(n)$ limit no longer applies, as we are not recursing or building a tree.

5.a. To differentiate good minions from bad, choose a minion, say M_1, and ask the other minions if M_1 is good or bad, while disregarding M_1's opinion on the other minion. This will result in $n-1$ opinions about the given minion. Since good minions always tell the truth, and bad minions are unreliable (may lie, but not necessarily), if M_1 receives at least $\frac{n}{2}$ good votes, then M_1 must be good. If M_1 does not receive $\frac{n}{2}$ good votes, repeat the process for M_2, M_3, etc. until a good minion is found. Once a good minion is found, his opinion can be used to judge all others.

However, if $\frac{n}{2}$ or more minions are bad, it will be impossible to reliably identify a good minion, as long as the bad minions conspire to trick Gru. If exactly $\frac{n}{2}$ minions are bad, they can conspire to ensure than no good minion is identified, since a good minion can only be voted as good by a maximum of $\frac{n}{2} - 1$ other good minions. In fact, they could conspire to vote 'good' for one of their own, giving Gru the false impression that he has identified a good minion. This holds if more than $\frac{n}{2}$ minions are bad. However, if not all of the bad minions conspire to fool GRU and truthfully report a good minion, Gru may be able to make the determination, but it is impossible to guarantee, as long as there are at least $\frac{n}{2}$ bad minions.

5.b. Given that at least $\frac{n}{2}$ minions are good and will always tell the truth:

Choose $\lfloor \frac{n}{2} \rfloor$ minions at random and conduct pairwise comparisons with all other minions. The results will fall into one of two categories:

(1) number of "Both Good" results $\geq \lfloor \frac{n}{2} \rfloor$. Since at least $\frac{n}{2}$ minions are good, the only way to have $\geq \lfloor \frac{n}{2} \rfloor$ "Both Good" results is if the minions chosen for the pairwise comparison testing are good themselves.

(2) number of "Both Good" results $< \lfloor \frac{n}{2} \rfloor$. Since good minions outnumber bad, any result in this category indicates that the chosen minion is bad, even if the other bad minions lie and indicate the chosen minion is good.

5.c. Assuming that at least $\frac{n}{2}$ minions are good and will always tell the truth:

1. Choose any minion, M' at random, and conduct pairwise testing against all other minions.

2. If the results of step 1 are $\geq \lfloor \frac{n}{2} \rfloor$ 'both good' results, then M' is a good minion. The run-time here is $O(n)$.

3. If M' did not receive $\geq \lfloor \frac{n}{2} \rfloor$ 'both good' results, we know that any minion that voted M' as bad is potentially good, and any minion who voted M' as good is bad, and that M' itself is bad.

Step 3 is also $O(n)$. The maximum number of times this process can repeat is $\lfloor \frac{n}{2} \rfloor$, since good minions outnumber bad. Therefore, this process is $\Omega(n)$.

6. ?? :(

7. def Dijkstra(V,E): #from Dasgupta, Papadimitriou, and Vazirani p.115

```
for all $u \in$ V:
dist(u) = $\infty$
parent(u) = nil
root = $v_0$
dist(root) = 0
Q = queue()
while Q not empty:
    u = deletemin(Q)
    for all edges (u,v) $\in$ E:
        if v.dist > dist(u) + l(uv):
                dist(v) = dist(u) + l(u,v)
                parent(v)=u
                decreaseKey(Q,v)

$G'$ = reverse(G) # make a copy of graph G with edges reversed
Dijkstra(G.V, G.E)
Dijkstra(G'.V,G'.E)
index1=0
index2=0
Array[sizeG][sizeG] # a 2D array to store distance between vertex pairs
for v in G:
    for v' in G':
        Array[index][index2] = v.distance + v'.distance
        index2 +=1
    index1 +=1
```

8. def modifiedDijkstra(V,E,startNode): #modified from Dasgupta, Papadimitriou, and Vazirani

```
    for all $u \in$ V:
    totalCost(u) = $\infty$
    parent(u) = nil
    root = $startNode$
    totalCost(root) = 0
    Q = queue()
    while Q not empty:
        u = deletemin(Q)
        for all edges (u,v) $\in$ E:
            if v.totalCost > totalCost(u) + l(uv) + cost(v):
                totalCost(v) = totalCost(u) + l(u,v) + cost(v)
                u.CostArray(v) = totalCost(v) # record the cost from start node to
                        just-closed node, as an attribute of start node
                parent(v)=u
                decreaseKey(Q,v)

    for all v in G:
        modifiedDijkstra(G.V,G.E,v) # calculate total cost from u to v for all(u,v) $\in$ G
```