Charles Zhang

Disc. 1D
Fri. 10am - 12pm
Song

1)*

- Algorithm: // *n* is the number of points
  - Initialize a global list, S_x, of all the points, sorted by x-components
  - Initialize a global list, S_y, of all the points, sorted by y-components
  - // Function begins here
  - If *n* is less than or equal to 1:
    - Return 0 and a list containing the point
  - If *n* is equal to 2:
    - Return the distance between the the 2 points and a list containing the points
  - Else:
    - Set a variable *left* equal to the return value of recursively calling this function on the left *n* / 2 elements of S_x
    - Set a variable *right* equal to the return value of recursively calling this function on the right *n* / 2 elements of S_x
    - Set a variable *delta* equal to the minimum of *left* and *right*
    - Initialize an empty list *L* and pointers to the beginning of the lists attached to *left* and *right*
    - For all elements of S_y:
      - If the point's x-component is within *delta* of the merge point's x-component:
        - Insert the point into *L*
    - For all elements in *L*:
      - Calculate the distance from the current point to the next 8 elements in the list (or to the end of the list if there are less than 8 elements left)
        - If any distance is smaller than the current *delta*, update *delta*
    - Return *delta* and *L*
- Time Complexity: O(*n* log *n*)
- Time Complexity Proof:
  - The initial sorts at the beginning of the algorithm take O(*n* log *n*) time
    - We have to sort by x-component in order to split the problem by x-component correctly
  - Each divide step will divide the current subproblem in half based on x-component
    - Here, we need to solve 2 subproblems of size *n* / 2, so our time is currently represented by $T(n) = 2T(n / 2) + ?$
  - Within each merge, we need to calculate and maintain a *delta*, which takes constant time

- We then use *delta* to construct a strip containing all candidates for closest point based on the logic that the strip can be at most *delta* wide in either subproblem
    - We can get a list of points in this strip sorted by y-component in O(n) time by making a single pass through our global list of all points sorted by y-component and making a constant time check for each point to see if the point's x-component is within *delta* of the merge point
    - For each point in this strip, we need to compare it to a constant number of other points, determined by the grid-breakdown, allowing us to say that the search for a possible new *delta* takes O(n) time
- Now, our time is represented by $T(n) = 2T(n / 2) + Cn$
- This is the same recurrence relation as merge sort, so we know that this algorithm is O(n log n)

2a)*

- Counterexample:
  - Given N = {2, 1}, S = {1, 2}, M = 100, n = 2
  - This algorithm simply chooses the city that has a lower operating cost in a given month, without regards to the moving cost
  - This algorithm will output [SF, NY], as SF has a lower operating cost in month 1 and NY has a lower operating cost in month 2
    - However, this generates a total cost of 102, as the moving cost is very high relative to the operating costs
  - In reality, the optimal answer would be either [SF, SF] or [NY, NY], which both have total costs of 3

2b)

- Example:
  - Given N = {1, 100, 1, 100}, S = {100, 1, 100, 1}, M = 0, n = 4
  - Here, there is no cost for moving cities
  - As a result, the best option is always to move to the city with the lower operating cost
  - Since there are 4 months, and the city with the lower operating cost changes every month, you must change location every month
  - This results in the optimal solution having 3 location changes

2c)*

- Algorithm:
  - If n is 0 or less:
    - Return 0
  - Initialize an empty array *opt* and variables *inNY* and *inSF* to true
  - If $S_1 < N_1$:
    - Set *inNY* to false
    - Set *opt*[1] to S_1
  - Else if f $N_1 < S_1$:
    - Set *inSF* to false
    - Set *opt*[1] to N_1
  - Else: *// **tie***
    - Set *opt*[1] to S_1
  - Set a variable i to 2
  - While i is less than or equal to n:
    - Initialize variables *nyMove, nyNoMove, sfMove, sfNoMove* to some infinite value
    - If *inNY*:
      - Set *nyMove* equal to *opt*[i - 1] + S_i + M
      - Set *nyNoMove opt*[i - 1] + N_i
    - If *inSF*:
      - Set *sfMove* equal to *opt*[i - 1] + N_i + M

                - Set *sfNoMove opt*[*i* - 1] + S_i
            - Get minimum value of *nyMove, nyNoMove, sfMove, sfNoMove* and place it in *opt[i]*
            - Set *inSF* and *inNY* to false
            - If *nyMove* or *sfNoMove* were the minimum:
                - Set *inSF* to true
            - If *sfMove* or *nyNoMove* were the minimum:
                - Set *inNY* to true
            - Increment *i*
    - Return *opt*[*n* - 1]
- Proof:
    - Base case:
        - *n* = 1
        - The only 2 operation costs are compared
        - The lesser one is placed into *opt*
        - *opt* is returned
        - Since no moving can possibly happen, the lesser operation cost is the correct option
        - Base case passed
    - Inductive step:
        - Assume that we have the optimal solution for months 1 through *i* - 1
        - Prove we can get the optimal solution for month *i*
        - Given the prior optimal solution, we have 2 options for month *i*:
            - Move:
                - If we move, we incur the cost of the moving cost and the other location's operating costs
                - This is calculated correctly within the algorithm with a simple calculation
            - Don't move:
                - If we don't move, we incur the cost of the current location's operating costs
                - This is calculated correctly within the algorithm with a simple calculation
            - Once our options are calculated, we take the minimum of both options and we provide the correct location information for the next iteration (simple logic tells us the location information is correct)
            - By definition, the minimum total cost after incurring the costs of a month is the optimal solution for that month
            - This logic can then be repeated until we have analyzed all *n* months
        - Inductive step complete
    - Proof by induction complete
- Time Complexity: O(*n*)

- Time Complexity Proof:
    - Each element of N and S will be accessed once
    - Each of these accesses comes with a constant number of computations and comparisons, so each access takes O(1) time
    - Since both N and S are of size n, the algorithm takes O(n) time overall

3)*

- Algorithm:
    - For all *n* words:
        - If the current word has more characters than *L*:
            - Return some failure condition
    - Initialize a 2-D array *costs* with size (*n* + 1) x (*n* + 1)
    - Initialize iterators *i* and *j* to 1
    - For all words, using *i* as an iterator: *//* **After loop, *costs* holds slack for words i → j**
        - Set *costs*[*i*][*i*] to *L* minus c_i
        - Set *j* to *i* + 1
        - While *j* <= *n*: *//* **Get costs for remainder of words**
            - Set *costs*[*i*][*j*] to *costs*[*i*][*j - 1*] minus c_j
            - If *j* < *n*: *//* **Add space**
                - Decrement *costs*[*i*][*j*] by 1
    - Reset *i* to 1
    - For all words, using *i* as an iterator: *//* **Square/invalidate entries**
        - Set *j* to *i*
        - While *j* <= *n*:
            - If *costs*[*i*][*j*] is negative:
                - Set *costs*[*i*][*j*] to some infinite value
            - Else:
                - Set *costs*[*i*][*j*] to the square of itself
        - Increment *i*
    - Initialize array *opt* and *partition* of length *n* + 1
    - Set *opt*[0] to 0
    - Reset *i* to 1
    - For all words, using *i* as an iterator:
        - Set *opt*[*i*] to some infinite value
        - Set *j* to 1
        - While *j* is less than or equal to *i*:
            - If *costs*[*j*][*i*] is not infinity and *opt*[*i* - 1] + *costs*[*j*][*i*] < *opt*[*i*]:
                - Set *opt*[*i*] to *opt*[*j* - 1] + *costs*[*j*][*i*]
                - Set *partition*[*i*] to *j*
    - Initialize an array *res*
    - Set iterator *i* to *n*
    - While *i* > 0: *//* **Write actual words**
        - Initialize an array *line*
        - Set a variable *x* to *partition*[*i*]
        - Insert words w_x to w_i into *line*
        - Insert *line* into *res*
        - Set *i* to *x* - 1
    - Reverse *res*
    - Return *res*

- Proof:
    - The initialization steps are more or less directly from the problem definition
        - We use some dynamic programming to calculate the costs, but it is pretty much a brute force approach
    - Base Case:
        - $n = 1$
        - The initialization will simply calculate the square of the slack from the lone word to the word limit L
        - The optimal solution for n will be set to the only existing value resulting from the initialization
        - The partition for n will then be set to the same index
        - When the solution is put together, the single word will be placed into the result and the loop will end
        - The correct result is returned
        - Base case passed
    - Inductive Step:
        - Assume: Initialization has completed properly and we have the optimal solutions for all values from 1 to i - 1
        - Prove: We can find the optimal solution for the ith value
            - Our initialization ensures we can't process a word count that would exceed the maximum line size
                - This is because all word sequences that do this are set to some infinite value, and this is checked for
            - By definition, the optimal solution has the lowest possible cost
            - We find this by assigning the minimum of our current solution and the next possible solution to our current solution
                - When this occurs, we also save the word number we should partition at
            - This process guarantees we enter the last loop with the optimal partition point associated with the ith value
            - We can repeat this logic to say that we can get the same success for the nth value
            - We then simply trace the partition points back through the array for our result
        - Inductive step passed
    - Proof by induction complete
- Time Complexity: $O(n^2)$
- Time Complexity Proof:
    - The first check of the input simply loops through all n values once, resulting in an $O(n)$ runtime
    - The next loop accesses the upper diagonal of an n x n (ish) matrix, resulting in an $O(n^2)$ runtime
    - The following loop runs through the elements accessed by the last loop and squares their values, resulting in an $O(n^2)$ runtime

- The third loop also runs through the same grouping of elements, resulting in an $O(n^2)$ runtime
- The final loop runs through all elements in an array of size n, and performs, at maximum, L operations, resulting in an $O(Ln)$ runtime
    - However, if $L > n$, then there will only be n operations performed, as there are only n words to loop through
    - Therefore, this loop can also be written as an $O(n^2)$ runtime
- Due to the difference in terms, we can only simplify this runtime to $O(Ln + n^2)$

4a)*

- Example:
  - Given s = {100, 4, 3, 2, 1}, x = {101, 101, 101, 101, 101}, n = 5
    - There is a surplus of data for all x_i
  - The optimal solution is to reboot on days 2 and 4
    - This results in a total of 300 terabytes processed
    - This is because the number of terabytes that can be processed drops off so fast after the first day, rebooting to regain optimal performance is more than worth the missed day of data

4b)

- Algorithm:
  - Initialize a 2-D array *opt* of size $n + 1$ by $n + 1$
  - For all values v from 1 to n:
    - Set *opt*[n][v] equal to the minimum of x_n and s_v
  - Initialize iterator variables i to $n - 1$ and j to n
  - While $i >= 1$:
    - Set a variable *reboot* to the value at *opt*[$i + 1$][1]
    - Set a variable *noReboot* to the value at *opt*[$i + 1$][$j + 1$] plus the minimum of s_j and x_i
    - Set *opt*[i][j] to the maximum of *reboot* and *noReboot*
    - If $j == 1$:
      - Set j to n
      - Decrement i by 1
    - Else:
      - Decrement j by 1
  - Return *opt*[1][1]
- Proof:
  - Base Case:
    - It's easy to see that there is no reason to reboot on the last day, as there is no more data to process if you do, you're just missing out on a day of data
    - As a result, we can easily fill out the optimal amount of data to process on the last day, assuming j days since the last reboot
    - This allows us to fill out the entire *opt*[n] array
    - Based on our logic above, processing as much data as possible given our limitations is the optimal behavior
    - This is what we do in our initialization step
    - Base case passed
  - Inductive Step:
    - Assume: We know the optimal amount of data that can be processed from day $i + 1$ to n and the optimal amount of data that can be processed from *day* i to n, where the last reboot occurred $j + 1$ days ago
    - Prove: We can find the optimal amount of data processed from day i to n,

where the last reboot occurred *j* days ago

- For any given entry, one of 2 possible options provides the optimal behavior:
    - You reboot:
        - You miss a day of data, but you reset the performance of your machine back to optimal levels
        - As a result, the amount of data you process is equal to the amount of data you process on day *i* + 1 when the last reboot was exactly 1 day ago
        - This is calculated correctly in the variable *reboot*
    - You don't reboot:
        - You no longer miss a day of data, but you incur the cost of the declining performance of your machine
        - As a result, the amount of data you process is equal to the amount of data you process from day *i* + *1* with *j* + 1 days since the last reboot, plus the amount of data you process today
            - This amount is given by the minimum of your processing limit and the amount of data you're given (min(x_i, s_j))
        - This is calculated correctly in the variable *noReboot*
- By definition, the optimal amount of data you can have on this day is the maximum of these 2 options
- We calculate this by calculating the maximum result of our options and assigning it to the current entry
- This provides the correct value for the current entry
- We can repeat this logic until we've arrived at *opt*[1][1], which represents the optimal amount of data we can process from day 1 to *n* with a fresh machine
- From here, we simply need to ensure we index our memoized array correctly
    - The if-else checks if we've reached the beginning of a given row in the array and increments one row up if that is the case
    - This follows with our assumption, as we traverse the array from right to left and bottom to top
    - Our indexing is correct

        - Inductive step passed
    - Proof by induction complete
- Time Complexity: O(n^2)
- Time Complexity Proof:
    - At the beginning of the algorithm, we create an *n* x *n* (ish) 2-D array
    - For each element in this array, we do multiple constant time operations like array accesses and additions

- This means our algorithm makes n^2 constant time operations
- Our algorithm has a runtime time of $O(n^2)$

5)

- Algorithm: // **Assuming dice are distinct**
  - Initialize a 2-D array *dp* of size [*n* + 1][*X* + 1]
  - For all values from 1 to *X*:
    - If the value is less than or equal to *m*:
      - Set *dp*[1][value] to 1
    - Else:
      - Set *dp*[1][value] to 0
  - Initialize iterator variables *i* to 2 and *j* to 1
  - While *i* <= *n* and *j* <= *X*:
    - Initialize a variable *sum* to 0
    - Initialize a counter *f* to 1
    - While *f* is <= *m* and *j* - *f* > 0:
      - Increment *sum* by *dp*[*i* - 1][*j* - *f*]
    - Set *dp*[*i*][*j*] to *sum*
    - If *j* == *X*:
      - Increment *i* by 1
      - Set *j* to 1
    - Else:
      - Increment *j* by 1
  - Return *dp*[*n*][*X*]
- Proof:
  - Base Case:
    - Using basic reasoning, we can fill out all results when we have 1 die and need to get a sum of *X*
      - If the sum is less than or equal to the number of faces on the die, we can achieve that sum by rolling that number on the die → 1 way
      - If the sum is greater than the number of faces on the die, we cannot possibly achieve that sum with 1 die → 0 ways
    - Our algorithm performs these exact checks, resulting in correct behavior for 1 die
    - Base case passed
  - Inductive Step:
    - Assume: We have all number of ways to roll sums 1 through *X* with *i* - 1 dice and sums 1 through *x* - 1 with *i* dice
    - Prove: We can get the number of ways to roll a sum of *x* with *i* dice
      - By adding an *i*th die, we add a possibility to roll an extra 1, 2, …, *m* to add to the sum
      - It then follows that, in order to arrive at a sum *x* using *i* dice, we can simply find the number of ways to arrive at a sum *x* - *k* using *i* - 1 dice, where *k* is equal to 1, 2, …, *m*
      - Using our memoized array, we can access these values based on our assumption

                - Therefore, we can get any value we need to calculate
                - From here, we simply need to ensure we index our memoized array correctly
                    - The if-else checks if we've reached the end of a given row in the array and increments one row down if that is the case
                    - This follows with our assumption, as we traverse the array from left to right and top to bottom
                    - By allocating n + 1 rows and X + 1 columns, we can directly use the number of dice and sum as indices
                    - Our indexing is correct
            - Inductive step passed
        - Proof by induction complete
- Time Complexity: O(nmX)
- Time Complexity Proof:
    - At the beginning of the algorithm, we create an n x X 2-D array
    - For each element in this array, we do m array accesses and some other constant time operations
    - This means our algorithm makes nmX-ish constant time operations
    - Our algorithm has a pseudo-polynomial runtime of O(nmX)

6)

- Algorithm:
    - Initialize an array *allBoxes* of size 3 * *n*
    - Create some struct *Box* that can hold height, depth, and width in the same object
    - Initialize iterator variables *i* and *j* to 0 // assuming given lists start from 0
    - For all *n* boxes:
        - Set a variable *x* to *h*(*i*)
        - Set a variable *y* to the minimum of *w*(*i*) and *d*(*i*)
        - Set a variable *z* to the maximum of *w*(*i*) and *d*(*i*)
        - Put a *Box* with height *x*, width *y*, and depth *z* into *allBoxes*[*j*]
        - Increment *j*
        - Set *x* to *d*(*i*)
        - Set a variable *y* to the minimum of *w*(*i*) and *h*(*i*)
        - Set a variable *z* to the maximum of *w*(*i*) and *h*(*i*)
        - Put a *Box* with height *x*, width *y*, and depth *z* into *allBoxes*[*j*]
        - Increment *j*
        - Set a variable *x* to *w*(*i*)
        - Set a variable *y* to the minimum of *h*(*i*) and *d*(*i*)
        - Set a variable *z* to the maximum of *h*(*i*) and *d*(*i*)
        - Put a *Box* with height *x*, width *y*, and depth *z* into *allBoxes*[*j*]
        - Increment *i* and *j*
    - Sort *allBoxes* by decreasing width times depth (base area)
    - Initialize an array *maxes* of size 3 * *n*
    - Set *i* to 0
    - For all 3 * *n* entries of *allBoxes*, using *i* as an iterator:
        - Set *j* to 0
        - For all values of *j* from 0 to *i*:
            - If *allBoxes*[*i*]'s width and depth are both less than *allBoxes*[*j*]'s width and depth, set *maxes*[*i*] to the maximum of *maxes*[*i*] and *maxes*[*j*]
        - Increment *maxes*[*i*] by the height of *allBoxes*[*i*]
    - Initialize a variable *max* to -1
    - For all values of *maxes*:
        - Set *max* to the maximum of *maxes*[*i*] and *max*
    - Return *max*
- Proof:
    - Base Case:
        - The maximum height stack has 1 box in it
            - This maximum height is therefore the height of that 1 box
        - This tells us that no stacking occurs at all
            - Due to this, the initialization is mostly irrelevant
        - In order for this to be true, it must also be true that there is no situation where one box's width and depth are less than that of another's
        - This condition is checked within the nest for loop

        - The end of our for loop increments the possible maximums by the height of the box
        - The final loop brute forces the checking of a maximum
        - The correct result is returned
        - Base case passed
    - Inductive Step:
        - Assume: We have the optimal stack height for all boxes with smaller base area than the current one
        - Prove: We can find the optimal stack height for the current box
            - This box's 3 possible rotations will have been stored into the *allBoxes* array
                - This storing forces the condition that width <= depth
            - Each element of *maxes* represents the corresponding box's maximum stack height if that box were at the very top
            - It's a simple mathematical fact that, by the problem definition, it is impossible to stack a box with a greater base area on top of a box with a smaller base area
                - This is because either or both of the larger area's width and depth will be larger than the smaller area's
                - By the problem definition, this is not allowed
            - Since through the boxes from largest base to smallest base, we therefore know we aren't skipping any possible stackings by not iterating through the boxes with smaller bases
            - When we iterate through a old box, we have 2 options:
                - We want it under our current box:
                    - This means we will stack our box on top of the old box
                    - In order for this to occur, we must satisfy 2 criteria:
                        - The stack formed by this operation is the largest so far:
                            - This is taken care of by checking for the maximum between the old box's stack and the current box's stack
                        - The current box fits within the dimensions of the old box:
                            - This is taken care of by the conditional in the nested for loop
                        - Both criteria are checked
                    - Once this is done, the new height of the stack will be equal to the height of the old stack plus the height of the current box
                    - This is taken care of once the optimal height of the old stack is discovered
                - We don't want it under our current box:

            - There are 2 possible reasons for this:
                - The bigger box's dimensions don't allow our current box to be stacked on it
                    - This is taken care of by the conditional within the nested for loop
                - We have already found a potential stack that is taller than the stack formed by this bigger box
                    - This is taken care of by checking for the maximum between the old box's stack and the current box's stack
            - Both possible reasons are taken care of
        - We can repeat this logic until we have found the maximum stack height for each box
        - The final loop brute forces the checking of a maximum
        - The correct result is returned
    - Inductive step passed
- Proof by induction complete
- Time Complexity: $O(n^2)$
- Time Complexity Proof:
    - The initialization of our list requires $3n$ sets of constant operations, which translates to $O(n)$ time
    - The sorting of our list takes $3n \log(3n)$ ish operations, which translates to $O(n \log n)$ time
    - The next for loop iterates through all $3n$ boxes, iterating through and performing constant time operations for each box that preceded the current element in the array
        - This is at worst a $O(n^2)$ runtime
    - Searching for a max requires searching through $3n$ elements, resulting in an $O(n)$ runtime
    - The overall algorithm's runtime can therefore be represented by the $O(n^2)$ runtime