Amy Seo

Dis. 1D
10:00-11:50 AM
Song

1. Closest Pair Problem

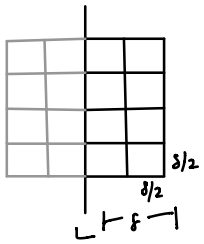
Algorithm: Suppose we are given a
- sort points by increasing x-coordinate into a new array X
- sort points by increasing y-coordinate into a new array Y.
- in both X and Y, each coordinate should have the position of itself in the other array
- return closest-pair helper function (X, Y)

closest-pair helper function
- base case: if size of X or Y ≤ 3, find the closest pair by directly comparing all possible pair distances and return this pair
- partition X into two halves: P and Q
- sort P by increasing x-coordinate into new array P_x
- sort P by increasing y-coordinate into new array P_y
- sort Q by increasing x-coordinate into new array Q_x
- sort Q by increasing y-coordinate into new array Q_y

(we can keep track of sorted coordinates in 2-dimensions w/ 2 separate arrays)

(keep track of each coordinate's position in the other array)

- let (P_0^*, P_1^*) = closest-pair helper function (P_x, P_y)
- let (Q_0^*, Q_1^*) = closest-pair helper function (Q_x, Q_y)
- let δ = min distance $((P_0^*, P_1^*), (Q_0^*, Q_1^*))$
- Let x^* = max x-coordinate in P
- Let L = (x^*, y) vertical line to divide P and Q
- Let S be the set of points within distance δ of L in X found by iterating through Y linearly
  - there can only be 16 max because each point in S must be at least $\delta/2$ apart, or else we would have found a closer pair than the current closest pair w/ distance δ, which is a contradiction.



- For each point in S
  - calculate distance from point to each of the other points in S
  - let (a, b) be the min. of these distances
- return min $((P_0^*, P_1^*), (Q_0^*, Q_1^*), (a, b))$

Proof:
- Base case: when there are are less than or equal to 3 points, we can directly calculate the closest points

- Inductive Hypothesis: Suppose for any two groups P and Q we have the closest pair of points i and j out of n total points.
- Inductive Step: Suppose a new point k is added for a total of n+1 points
  - Case 1: k is part of a closest pair in P
  - Case 2: k is part of a closest pair in Q
  - Case 3: k is not part of any closest pair
  - Case 4: Any of the above and k is in S
  - no matter the case, when combining P and Q, the overall closest pair is either in P, in Q, or one point is in P and the other in Q.
  - we compare the closest points in P and Q to the smallest possible pair in S, so we will always have the closest pair for n+1 points.

Time Complexity:
- sorting the points → $O(n \log n)$
- dividing set of points $2T(\frac{n}{2})$ and performing recurrence relation
  - finding smallest points in S in $O(n)$
- Total = $O(n \log n) + \underbrace{2T(\frac{n}{2}) + cn}$

$$T = O(n \log n) + O(n \log n)$$
$$T = O(n \log n)$$

2. a) Incorrect algorithm:

```
For i=1 to n
    If Ni < Si then
        Output "NY in month i"
    Else
        Output "SF in month i"
End
```

Counterexample:

$n = 3$, $m = 100$

	month 1	month 2	month 3
NY	5	20	10
SF	10	10	50

$5 + (10 + 100) + (10 + 100) = 225$

actual min cost = $5 + 20 + 10 = 35$

b) Example in which every optimal plan must move (change locations) at least 3 times:

$n = 4$, $m = 0$

	month 1	month 2	month 3	month 4
NY	20	30	10	15
SF	50	15	40	5

In this example, every optimal plan must change locations at least 3 times because the moving cost is 0, so the best location to choose will always be the location with the cheaper price. Since the cheaper price alternates between the 2 locations, you'd have to move 3 times for these 4 months.

c) Algorithm: Let n be the number of months, M be the moving cost, and N_i and S_i be the operating cost on month i

- Initialize an empty 2D array OPT of size (2xN) where OPT[NY][0] = OPT[SF][0] = 0

	0	. . . . .	n
NY	0	. . .	
SF	0	. . .	

- For i = 1 to i = n
  - set OPT[NY][i] = N_i + min (OPT[NY][i-1], M + OPT[SF][i-1])
  - set OPT[SF][i] = S_i + min (OPT[SF][i-1], M + OPT[NY][i-1])
- return min (OPT[SF][n], OPT[NY][n])

Time Complexity: we create the table by iterating through all n months, so this algorithm runs in O(n) time where n = number of months

Proof:

- Base Case: The minimum cost for the first month is min(N_1, S_1) as we can stay from any location
- Inductive Hypothesis: the minimum cost of operation at month i is OPT[SF][i] at the SF location, and OPT[NY][i] at the NJ location.
- Inductive Step: At month k=i+1, there are 2 options:
  1) stay at current office
  2) move offices

  and we store the minimum of the 2 for both locations
  OPT[SF][K] = S_K + min (OPT[SF][i], M + OPT[NY][i])
  OPT[NY][K] = N_K + min (OPT[NY][i], M + OPT[SF][i])

  By the inductive hypothesis, we know OPT[SF][i] and OPT[NY][i] are the minimum costs at month i, so the minimum at month k must be the minimum of the 2 options at both locations.

∴ our algorithm always finds the minimum operating cost.

3. Pretty-Printing

Algorithm:

- Calculate all possible slack values, let slack(i,j) be the slack value for a line with words [W_i, ..., W_j]
  - For each word i = 1 to i = n
    - let l_i = C_i of W_i
    - if l_i <= L
      - set slack(i,i) = L - l_i
    - else
      - set slack(i,i) = ∞
    - l_i = l_i + 1 (increment for space)
    - for j = i+1 to j = n
      - set l_i = l_i + C_j
      - if l_i <= L
        - set slack(i,j) = L - l_i
        - l_i = l_i + 1 (increment for space)

- else
  - slack(i,j) = ∞
  - exit loop

- let OPT(x) be the minimum slack for the first x words and start(i) be the first word in the current line [Wi, ..., Wj]
- initialize OPT(0) = 0
- For x=1 to x=n:
  - OPT(x) = min (slack(y, x)^2 + OPT(y-1)) for y=1 to y=x
  - set start(x) = y min
- Trace backwards from start(n) to recover each line
- return OPT(n)

Proof:

- Base case: when there is one word (assuming its length <=L), the minimum slack is $(L - c_i)^2$.
- Inductive Hypothesis: Suppose there are n=k words split up into lines with the minimum slack possible.
- Inductive Step: suppose there are n=k+1 words. There are 2 cases:
  - calculate the new slack = min(slack(y, W_{k+1})^2 + OPT(y-1)) for all y=1 to W_{k+1}
  - Case 1: add this word W_{k+1} to the last line
  - Case 2: add this word to a new line
  - the minimum possible slack is chosen out of the 2 options, so the minimum slack is maintained for n=k+1 words.

∴ Our algorithm always chooses words to go on the lines that will result in the minimum slack.

Time Complexity:

- Calculating slack values = $O(n^2)$
- calculating minimum slack w/ recurrence relation = $O(n^2)$
- total = $O(n^2 + n^2)$
  $= O(n^2)$

4. a)

	Day 1	Day 2	Day 3	Day 4	Day 5
x	10	4	10	6	10
s	9	2	3	4	1

↑ ↑

$x_i > s_i$

optimal solution: reboot on Day 2 + Day 4

b) Algorithm: Let n = number of days

- Let OPT(n,j) denote max amount of TB that can be processed starting from day n where the last day the software was rebooted is denoted by j
- For j=1 to j=n:
  - set OPT(n,j) = min(X_n, S_j) } on last day, never reboot!
- For i = n-1 to i=1:

- For j=1 to j=i:
  - set OPT(i,j) = max(OPT(i+1,1), min(Xi, Sj) + OPT(i+1, j+1))
    - OPT(i+1,1): spend current day rebooting
    - OR
    - min(Xi, Sj) + OPT(i+1, j+1): process as much data on current date as possible while possibly possibly rebooting next day (later day)
- return OPT(1,1)

Time Complexity:
- setting all OPT(n,j) to make the last day never reboot → O(n) time
- setting the recurrence relations → $O(n^2)$ because we compare in constant time within the 2 nested loops.
- $O(n+n^2) = O(n^2)$ time

Proof:
- Base case: n=1
  - choose min(X_1, S_1) since there is only one day
- Inductive hypothesis:
  - Suppose for some n=k, we have the most TB that can be processed by our software by day k.
- Inductive step:
  - Suppose n = k-1, then we have 2 options
    1) spend the day rebooting our computer
       ↳ OPT(i+1,1)
    2) process as much data as possible today and possibly reboot tomorrow or some future day
       ↳ min (Xi, Sj) + OPT(i+1, j+1)
  - we take the max of these options, and thus, we have the maximum amount of TB that can be processed on day k-1

∴ By induction, our algorithm always returns the maximum amount of TB that can be processed.

5. Observations:

Ex] n = 2, m=6, X = 10
- whatever value the first die is, the problem breaks up into subproblems:
- In this example: diceThrows(2,6,10) where diceThrows(n, m, x)
  - n: # of dice
  - m: # of sides
  - x: target sum

diceThrows(2,6,10) =
diceThrows(1,6,9) + diceThrows(1,6,8)
+ diceThrows(1,6,7) + diceThrows(1,6,6)
+ diceThrows(1,6,5) + diceThrows(1,6,4)

- case 1: First die = 1 → sum = 10-1=9
- case 2: First die = 2 → sum = 10-2=8
- case 3: First die = 3 → sum = 10-3=7
- case 4: First die = 4 → sum = 10-4=6
- case 5: First die = 5 → sum = 10-5=5
- case 6: First die = 6 → sum = 10-6=4

* extend to multiple dice? → table of n dice by target sum

Algorithm: Let n be the number of dice, m be the number of faces, and X be the target sum

- initialize empty set OPT
- define recurrence_step(n,x) helper function as:
  - if $X > (m \times n)$:
    OPT(n,x) = 0
    return 0
    } no possible way to sum up to X
  - if n is 0:
    - if target >= 0
      return 1
    - else
      return 0
  - if (n,X) in OPT
    - return OPT(n,x)
  - set combos = 0
  - For i = 1 to i = m:
    - combos += recurrence_step(n-1, X - i)
  - OPT(n,X) = combos
  - return combos
- return recurrence_step(n,x)

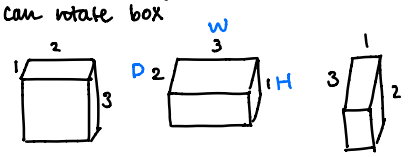
Proof:

- Base Case: $n = m_b = X = 1$ One die w/ one face w/ target sum = 1
  ↳ only 1 possibility: 1 die
- Inductive Hypothesis: Suppose we have n dice with m faces and a target sum of X, and we know the number of combos
- Inductive Step:
  - Case 1: Suppose we increase $m = m+1$
    - Then we increase the number of iterations of the for loop by 1 to account for the additional possibility of a dice to have this extra value for each n dice.
  - Case 2: Suppose we increase $n = n+1$
    - Then another possible dice is accounted for in the recurrence step, creating another set of for loop iterations for all possible face values!

  In both cases, our algorithm handles the additional possibilities of combinations, thus maintaining the total number of combinations.

Time complexity: $O(mnx)$ because we run the for loop m times to calculate all the possible states of each n dice for every possible sum below x. This is pseudopolynomial as our input size is 3, but the values of m, n, and/or x could cause the runtime to be exponential.

6. observations:
   - 2D base of lower box > 2D base of higher box → $w_1 > w_2$ + $l_1 > l_2$
   - can use any # of same box
   - can rotate box



1 x 2 x 3 | 2 x 3 x 1 | 3 x 1 x 2
2 x 1 x 3 | 3 x 2 x 1 | 1 x 3 x 2

Algorithm:
- for each box with height h(i), width w(i), and depth d(i)
  - add 2 other possibilities of the box (rotated once & rotated twice) to the array of boxes
- sort all 3n boxes by decreasing base area
- let OPT(i) = maximum possible stack height with current box i at the top of the stack
- for each box i = 0 to i = 3n-1
  - set OPT(i) = height of box i
- for j = 1 to j = 3n-1
  - for k = 0 to k = j-1
    - if width of box$_j$ < width of box$_k$ AND depth of box$_j$ < depth of box$_k$
      OPT(j) = max (OPT(j), OPT(k) + height of box$_j$)
- return max value in OPT

Proof:
- Base case: For n=1 box, the max height is the height of all the 3 possible rotations of the box stacked from biggest base area at the bottom to smallest base area at the top, only including each box if $w_i < w_{i-1}$ and $d_i < d_{i-1}$.
- Inductive Hypothesis: For n=k boxes, suppose we know the max height of the stack.
- Inductive Step: Suppose there are n=k+1 boxes. There are 2 cases:
  - Case 1: add the additional box to the stack
    max (OPT(j), OPT(k) + height of box$_j$) = OPT(k) + height of box$_j$
    - since our boxes are sorted by decreasing base area, we know to add the box if each dimension of the 2D base is strictly less than that of the greater box, and if adding this box's height creates a larger stack height than the current one
  - Case 2: don't add the box to the stack
    max (OPT(j), OPT(k) + height of box$_j$) = OPT(j)
    - adding this box would prevent a different box with similar/slightly larger dimensions from contributing its larger height to the stack, so we don't add it
- Since the box is either added or not added, our algorithm always returns the max height for any n number of boxes.

Time complexity:

- adding all possible rotations for each box = $O(n)$
- sorting boxes = $O(3n \log 3n) = O(n \log n)$
- initialize opt = $O(3n) = O(n)$
- recurrence relation = $O((3n)^2) = O(n^2)$

$O(n^2)$ where n = number of boxes