CSC 373 Week 1 Notes

Divide and Conquer:

- Divide and conquer is an that aims to break down the problem into sub-problems (Divide), solve each problem (conquer) and then combine the solutions (Combine).
- Divide: Break the original subproblems into smaller ones.

  Conquer: Solve the sub-problems. You may need to use recursion, or if the subproblems are simple enough, you can solve it straight forward.

  Combine: Combine the solutions to the subproblems to create a final solution.

- A recurrence is an eqn or inequality that describes a function in terms of its value on smaller inputs.
- There are 3 main methods for finding Θ or O bounds for recurrences:

  1. Substitution Method: We guess a bound and then use induction to prove our guess right.

  2. Recursion Tree Method: We convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

3. Master Method: The master method provides bounds for recurrences in the form of $T(n) = a\,T(\frac{n}{b}) + f(n)$ where

n = size of input

a = number of subproblems in recursion

$\frac{n}{b}$ = size of each subproblem

$f(n)$ = cost of the work done outside the recursive call. This includes the cost of dividing the problem and merging the soln.

Note: $a \geq 1$, $b > 1$ and $f(n)$ is a function.

Let $d = \log_b a$. Then:

a) If $f(n) = O(n^{d-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = O(n^d)$

b) If $f(n) = O(n^d \log^k n)$ for some constant $k \geq 0$, then $T(n) = O(n^d \log^{k+1} n)$.

c) If $f(n) = O(n^{d+\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = O(f(n))$

Examples:

- Some problems that use divide and conquer include:
  1. Quicksort
  2. Mergesort
  3. Counting Inversions
  4. Closest pair in R^2
  5. Karatsuba's Algorithm
  6. Strassen's Algorithm

- Quicksort:
  - With quicksort, you choose a pivot (I'll be using the last element, but you can choose another), put all elements that are less than the pivot to the left and all other elements to the right. Then, you recursively repeat these steps for the left and right subarraies.

  - E.g. [1, 30, 27, 2, 15, 8, 16]

    Left (1), Right (8), Pivot (16)

    P stands for pivot. I'll also be using 2 pointers, L (Left) and R (Right).

    Step 1: Since the Left value (1) is less than the pivot value (16), we move it 1 to the right.

    [1, 30, 27, 2, 15, 8, 16]

    L (30), R (8), P (16)

Step 2: Since the Left value (30) is greater than the pivot value (16) and the Right value (8) is less than the pivot value, swap Left and Right value, then, move Left one to the right and Right one to the left.

[1, 30, 27, 2, 15, 8, 16]

Swap

↓

[1, 8, 27, 2, 15, 30, 16]

L R P

Step 3: Since the Left value (27) is greater than the pivot value (16) and the Right value (15) is less than the pivot value, swap the Left and Right values and then move Left one to the right and Right one to the left.

[1, 8, 27, 2, 15, 30, 16]

Swap

↓

[1, 8, 15, 2, 27, 30, 16]

L P

R

Step 4: If Left ≥ Right, the point that they met is where the left subarray ends and the right subarray begins. You apply the same steps to each subarray recursively.

[$\underbrace{1, 8, 15, 2}_{\text{Left Subarray}}$, $\underbrace{27, 30}_{\text{Right Subarray}}$, 16]

- Pseudo-Code:

```
func Quicksort (array):
    left = 0
    right = array.length - 1

    if (right - left == 0):  # Only 1 element
        return array            in array

    pivot = array [right]
    partition = partitionFunc (left, right-1, pivot)
    Quicksort (left, partition -1)
    Quicksort (partition +1, right)
```

- Merge Sort:
  - With merge sort, you divide up the array into halves and you recursively do merge sort on each half. Then you merge the sorted halves.

  - Pseudo-Code:

```
func MergeSort(arr):
    left = 0
    right = arr.length - 1

    if r > l:
        m = (l + r) / 2   (Find the index to split arr)

        Mergesort(arr[:m+1])  (Call mergesort on
                               the left subarray)
        Mergesort(arr[m+1:])  (Call mergesort on
                               the right subarray)
        merge()   (Merge the 2 subarraies)
```

- Counting Inversions:
  - Problem: Given an array of length n, count the number of pairs (i, j) s.t. $i<j$ but $a[i] > a[j]$.

  - Brute Force Solution:
    - Use a nested for loop.
    - Pseudo-Code:

```
func count_inversions(a):
    result = 0

    for i in range(n):
        val = a[i]
        for j in range(i+1, n):
            new_val = a[j]
            if(new_val > val):
                result += 1

    return result
```

    - Time Complexity: $\Theta(n^2)$

  - Divide and Conquer Solution:
    - Divide: Break the array into 2 halves of equal length, x and y.

    - Conquer: Count the number of inversions in each subarray recursively.

    - Combine: Count the number of inversions across x and y. (I.e. One element is in x and the other is in y.) Then, combine the 3 results.

- Pseudo-Code:

```
func count-inversions(L):
    if(len(L)==0):
        return (0, L)

    # Divide L into 2 halves, A and B
    (rA, A) = count-inversions(A):
    (rB, B) = count-inversions(B):
    (rAB, L') = merge-and-count(A,B)
    ↑ Here, I'm counting the inversions where
      one element is in A and the other is B.
    return (rA + rB + rAB, L')
```

Note: We don't have to sort the 2 halves, but it becomes a lot easier if we do.

- E.g.

[1, 5, 4, 8, 10, 2, 6, 9, 3, 7]

Step 1: Split the array into 2 halves.
A = [1, 5, 4, 8, 10]
B = [2, 6, 9, 3, 7]

Step 2: Find the number of inversions in A.
There is only 1: 5 and 4.

Step 3: Find the num of inversions in B.
There are 3: 6 and 3
9 and 3
9 and 7

Step 4: Find the num of inversions across A and B.
There are 13: 4 and 2, 4 and 3,
5 and 2, 5 and 3,
8 and 2, 8 and 6,
8 and 3, 8 and 7,
10 and 2, 6, 9, 3, 7

Hence, the total number of inversions is 17.
(1 + 3 + 13 = 17.)

- What happens if we sort the 2 halves:
  - Suppose that the 2 halves, A and B are sorted.

    I.e. $A = [1, 3, 7, a_i, 10]$
    $B = [2, 4, 9, b_j, 13]$

  - Then, if we compare any element in A, say a_i, with any element in B, say b_j, we'll know these info:
    1. If $a_i < b_j$, then $a_i < b_{j+n}$.
    2. If $a_i > b_j$, then $a_{i+n} > b_j$.

  - Because of the 2 points above, we don't need to compare every point/element in A with every element in B. It can save us a lot of time.

- Run Time Analysis:
  - Suppose $T(n)$ is the worst-case running time for inputs of size n.
  - Our algo satisfies $T(n) \le 2T(\frac{n}{2}) + O(n)$ if the 2 halves are sorted. This is because if the 2 arrays are sorted, then we will only traverse each array once.

$$T(n) \le \underbrace{2T(\tfrac{n}{2})}_{\text{This comes from dividing the array into halves and finding the num of inversions in each half.}} + \underbrace{O(n)}_{\text{This comes from traversing each array exactly once.}}$$

Note: Master's Theorem says $T(n) = O(n \lg_2 n)$

However, if the 2 halves aren't sorted, then the algo satisfies

$$T(n) \le 2T(\tfrac{n}{2}) + \underbrace{O(n^2)}_{\text{Now, we have to compare each element in A with each element in B. Hence, we'll need a nested loop.}}$$

- Proof of Run Time Complexity:
  1. **Using Substitution Method:**
     - We'll "guess" that the run time complexity is $O(n\lg_2 n)$ and then use induction to prove it.

       I.e. "Guess" $T(n) = O(n\lg_2 n) \leq c \cdot n\lg_2 n$ and use induction to prove it.

     - Base Case $(n=1)$:
       Let $n=1$
       LHS: $T(1) = O(1 \lg 1) \leftarrow \log 1 = 0$
       $= O(0)$

       RHS: $c \cdot (1\lg 1) = c \cdot 0$
       $= 0$

       $\therefore$ Any c works for the base case.
       $\therefore$ The base case holds.

     - Hypothesis Step:
       Assume that $T(n)$ holds for some $n=k$, $k \geq 1$, $k \in \mathbb{N}$.

     - Induction Step:
       We know that $T(n) \leq 2\,T(\frac{n}{2}) + O(n)$

Here, I'm subbing $2T(\frac{n}{2}) + O(n)$ for $T(n)$ $\longrightarrow$ $\leq 2(c \cdot \frac{n}{2} \cdot \lg \frac{n}{2}) + m \cdot n$

$\leq \frac{2cn}{2} \cdot \lg(\frac{n}{2}) + mn$

$\leq cn \cdot (\lg n - \lg 2) + mn$

Recall that $\lg_2 2 = 1$ $\longrightarrow$ $\leq cn(\lg n - 1) + mn$

$\leq cn \cdot \lg n - cn + mn$

$\leq cn\lg n + (m-c)n$

$\leq cn\lg n$

2. Using Master's Theorem:
   - In our case, $a = 2$ and $b = 2$.
     Hence, $d = \log_b a$
     $= \log_2 2$
     $= 1$
   - Our $f(n) = O(n^d \cdot \log^k n)$ for some $k \geq 0$
     This is because $d=1$, so we get $f(n) = O(n \cdot \log^k n)$. If $k = 0$, then we get $f(n) = O(n)$, which is what we have originally.
   - Hence, $T(n) = O(n^d \log^{k+1} n)$
     $= O(n \log n)$

- Closest Pair in R^2:
  - Problem: Given n points of the form (x_i, y_i), find the closest pair of points.

  - Brute Force Solution:
    - Get the distance between every pair of points.
    - Time Complexity: $\Theta(n^2)$

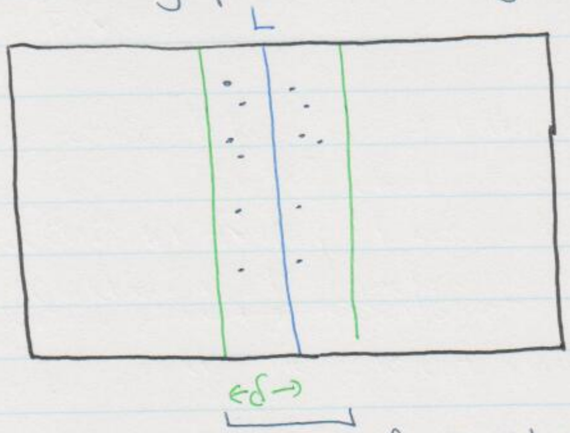
  - Divide and Conquer Solution:
    - Divide: Divide the points in half (equal halves) by drawing a vertical line L.

    - Conquer: Solve each half recursively.

    - Combine: Find the closest pair with one point on each side of L. Then, return the best of the 3 solutions.

- One thing we can do to reduce the amount of work we do is to find the min distance on both sides of L and then restrict the straddling points based on that distance.

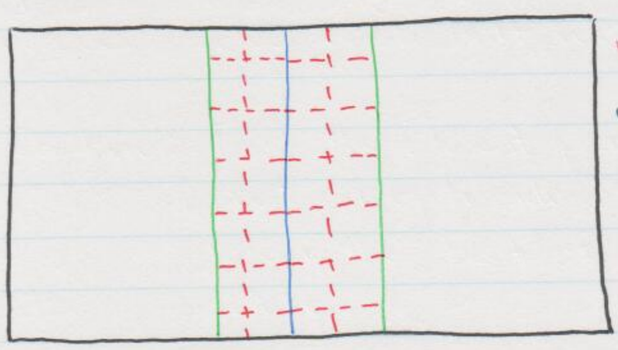
  E.g. Suppose δ is the min dist between 2 points s.t. both points are on the same side. Then, we can restrict the straddling points s.t. they are within δ of L.



We will only consider points within this area.

- After doing that, we sort the points with δ horizontal distance of L by their y coordinate.

- Now, consider this: We divide up the strip into $\delta/2$ by $\delta/2$ squares.

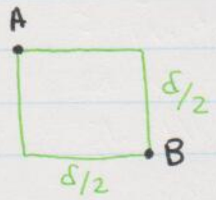


Note: Assume each "red" square has side length of $\delta/2$.

We know that there can't be 2 or more points in 1 $\delta/2 \cdot \delta/2$ square.

Proof:

I will prove this with a proof by contradiction. Assume that there are 2 points in 1 $\delta/2 \cdot \delta/2$ square.



$A = (x_1, y_1)$
$B = (x_2, y_2)$

The dist between A and B is $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$

$(x_1 - x_2) = \delta/2$

$(y_1 - y_2) = \delta/2$

So now, we have $\sqrt{\delta^2/4 + \delta^2/4}$

$\sqrt{\delta^2/2}$

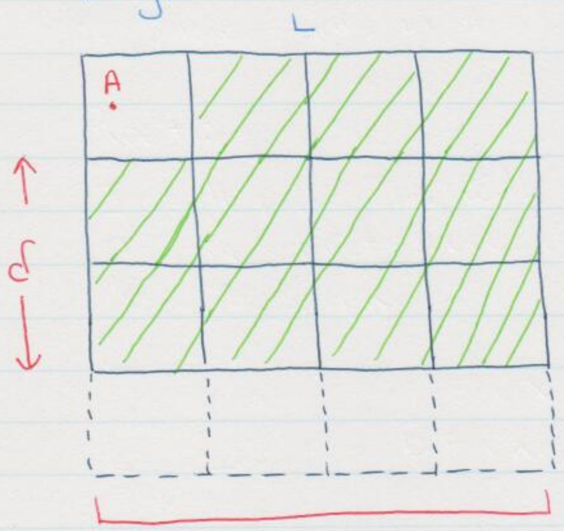
$\delta/\sqrt{2}$

$\delta/\sqrt{2} < \delta$ and since we assumed earlier that the smallest dist between 2 points on the same side of L is δ, this is a contradiction.

$\therefore$ Each $\delta/2 \cdot \delta/2$ square can't have more than 1 point.

After sorting the points by their y coordinate, each point only has to be compared with the next 11 points in the list. This is because any point more than 11 positions away will have a vertical distance of at least d.

E.g.



A has to check with each of the shaded squares to see if there's a point in there s.t. the dist btwn them is less than δ.

Any points in these dotted boxes have a vertical distance of > δ.

Note: We only care about not going over δ for the vertical dist btwn 2 points.
We may go over δ for the horizontal dist btwn 2 points even if the second point is in 1 of the shaded squares.

- Run Time Analysis:
  - Finding points on the strip: $O(n)$
  - Sorting points on the strip by their y-coordinate is $O(n \log n)$.
  - Testing each point against 11 points: $O(n \cdot 11)$ or $O(n)$
  - Total running time: $T(n) \le 2T(\frac{n}{2}) + O(n \lg n)$
  - By Master's Thm, we get: $T(n) = O(n \lg^2 n)$

- Karatsuba's Algorithm:
  - It is a fast way to multiply 2 n digit numbers x and y.

  - Brute Force Soln:
    - Doing it the way we learned at school.
    - Time Complexity: $O(n^2)$

  - Divide and Conquer Soln:
    - Divide each digit into 2 parts:

      $x = x_1 \cdot 10^{n/2} + x_2$

      $y = y_1 \cdot 10^{n/2} + y_2$

    - Now, $x \cdot y = (x_1 \cdot 10^{n/2} + x_2)(y_1 \cdot 10^{n/2} + y_2)$

      $= (x_1 y_1) \cdot 10^n + (x_1 y_2 + x_2 y_1) \cdot 10^{n/2} + x_2 y_2$

      $= \underline{(x_1 y_1)} \cdot 10^n + \underline{x_2 y_2} +$

      $((x_1 + x_2)(y_1 + y_2) - \underline{x_1 y_1} - \underline{x_2 y_2}) \cdot 10^{n/2}$

      This term came from $(x_1 y_2 + x_2 y_1)$

    - Now, we just have 3 multiplications:
      1. $x_1 y_1$
      2. $x_2 y_2$
      3. $(x_1 + x_2)(y_1 + y_2)$

    - Time Complexity:
      - $T(n) \leq 3T(\frac{n}{2}) + O(n)$
      - By Master's Thm, $T(n) = O(n^{\lg_2 3})$

        $\approx O(n^{1.58})$

- E.g. Let's multiply 1234 with 4321.

$x = 12 \cdot 10^2 + 34$
$y = 43 \cdot 10^2 + 21$

$a_1 = 12 \cdot 43$
$\rightarrow x_1 = 1 \cdot 10 + 2$
$\rightarrow y_1 = 4 \cdot 10 + 3$
$\rightarrow a_2 = 1 \cdot 4 = 4$
$\rightarrow d_2 = 2 \cdot 3 = 6$
$\rightarrow e_2 = (1+2)(4+3) - a_2 - d_2$
$= (1+2)(4+3) - 4 - 6$
$= (3)(7) - 10$
$= 11$
$= 4 \cdot 10^2 + 11 \cdot 10 + 6$
$= 516$

$d_1 = 34 \cdot 21$
$\rightarrow x_1 = 3 \cdot 10 + 4$
$\rightarrow y_1 = 2 \cdot 10 + 1$
$\rightarrow a_2 = 2 \cdot 3 = 6$
$\rightarrow d_2 = 1 \cdot 4 = 4$
$\rightarrow e_2 = (3+4)(2+1) - a_2 - d_2$
$= (7)(3) - 6 - 4$
$= 11$
$= 6 \cdot 10^2 + 11 \cdot 10 + 4$
$= 714$

$e_1 = (12+34) \cdot (43+21) - a_1 - d_1$
$= \underline{46 \cdot 64} - 516 - 714$
$\rightarrow x_1 = 4 \cdot 10 + 6$
$\rightarrow y_1 = 6 \cdot 10 + 4$
$\rightarrow a_2 = 4 \cdot 6 = 24$
$\rightarrow d_2 = 6 \cdot 4 = 24$
$\rightarrow e_2 = (4+6)(6+4) - 24 - 24$
$= 52$

$\rightarrow e_1 = 24 \cdot 10^2 + 52 \cdot 10 + 24 - 516 - 714$
$= 1714$

Combining everything, we get:
$1234 \cdot 4321$
$= 516 \cdot 10^4 + 1714 \cdot 10^2 + 714$
$= 5,332,114$

- Strassen's Algorithm:
  - Generalizes Karatsuba's insight to design a fast algo for multiplying 2 n by n matrices.

  - Time Complexity:
    - $T(n) \leq 7T(\frac{n}{2}) + O(n^2) \Rightarrow T(n) = O(n^{\lg_2 7})$

Note:
$\lg_2 8 = 3$, so $O(n^{\lg_2 7})$ is a bit better than $O(n^3)$