Week 8 - Monday

COMP 4500

Last time

- What did we talk about last time?
- Finished Master Theorem
- Solved exercises

Questions?

Logical warmup

- An Arab sheikh tells his two sons to race their camels to a distant city to see who will inherit his fortune
- The one whose camel is slower wins
- After wandering aimlessly for days, the brothers ask a wise man for guidance
- Upon receiving the advice, they jump on the camels and race to the city as fast as they can
- What did the wise man say to them?

Review

Interval Scheduling

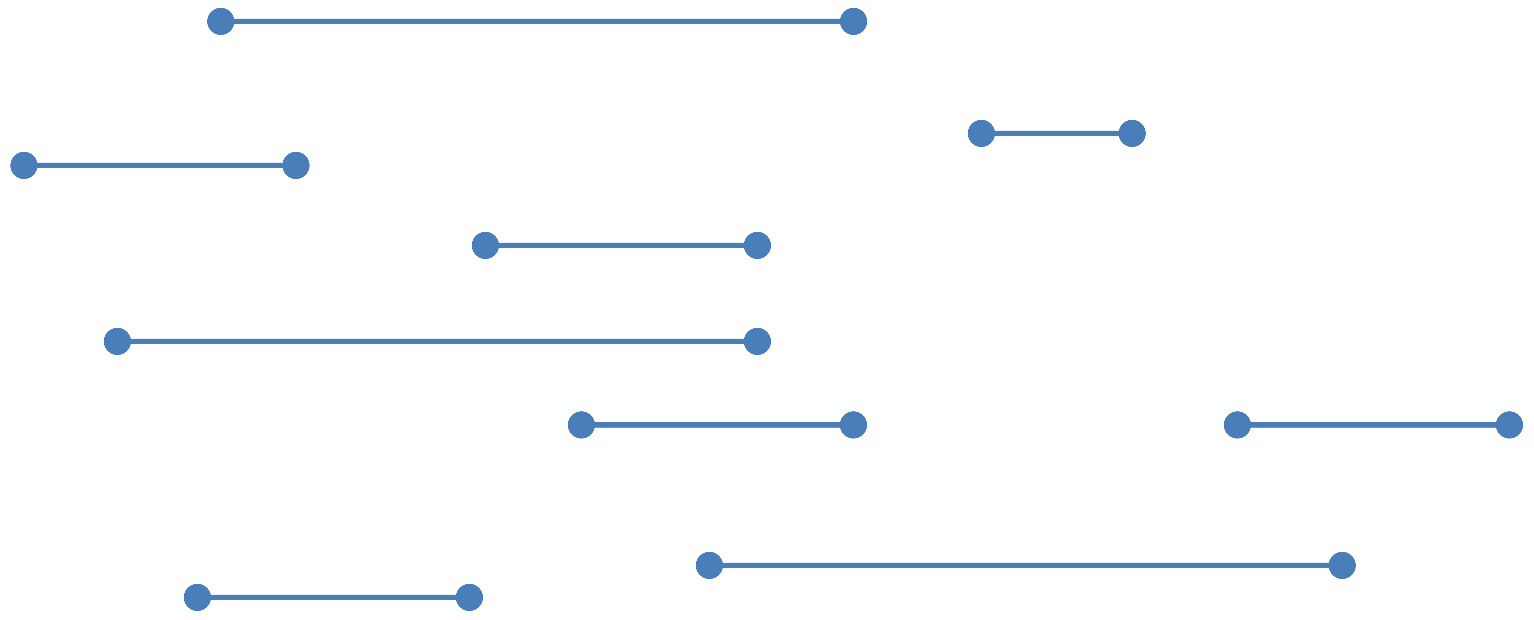
Interval scheduling

- In the interval scheduling problem, some resource (a phone, a motorcycle, a toilet) can only be used by one person at a time
- People make requests to use the resource for a specific time interval [*s*, *f*]
- The goal is to schedule as many uses as possible
- There's no preference based on who or when the resource is used

Interval scheduling algorithm

- Interval scheduling can be done with a greedy algorithm
- While there are still requests that are not in the compatible set
  - Find the request *r* that ends earliest
  - Add it to the compatible set
  - Remove all requests *q* that overlap with *r*
- Return the compatible set

Interval scheduling example



Running time

- First, we sort the ***n*** requests in order of finishing time
  - The best comparison-based sort takes O(***n*** log ***n***)
- We scan through the ***n*** sorted requests again and make an array ***S*** of length ***n*** such that ***S***[***i***] contains the starting value of ***i***, ***s***(***i***)
  - O(***n***) time
- Our algorithm selects the first interval in our list sorted on finishing time. We then move through array ***S*** until we find the first interval ***j*** such that ***s***(***j***) ≥ the finishing time selected. We add it. We continue the process until we have moved through the entire array ***S***.
  - O(***n***) time
- Total time: O(***n*** log ***n***)

Shortest Paths

Shortest path set up

- Directed graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with start node $\boldsymbol{s}$
- Assume that there is a path from $\boldsymbol{s}$ to every other node (although that's not critical)
- Every edge $\boldsymbol{e}$ has a length $\boldsymbol{l_e} \geq 0$
- For a path $\boldsymbol{P}$, length of $\boldsymbol{P}$ $\boldsymbol{l}(\boldsymbol{P})$ is the sum of the lengths of the edges on $\boldsymbol{P}$
- We want to find the shortest path from $\boldsymbol{s}$ to every other node in the graph
- An undirected graph is an easy tweak

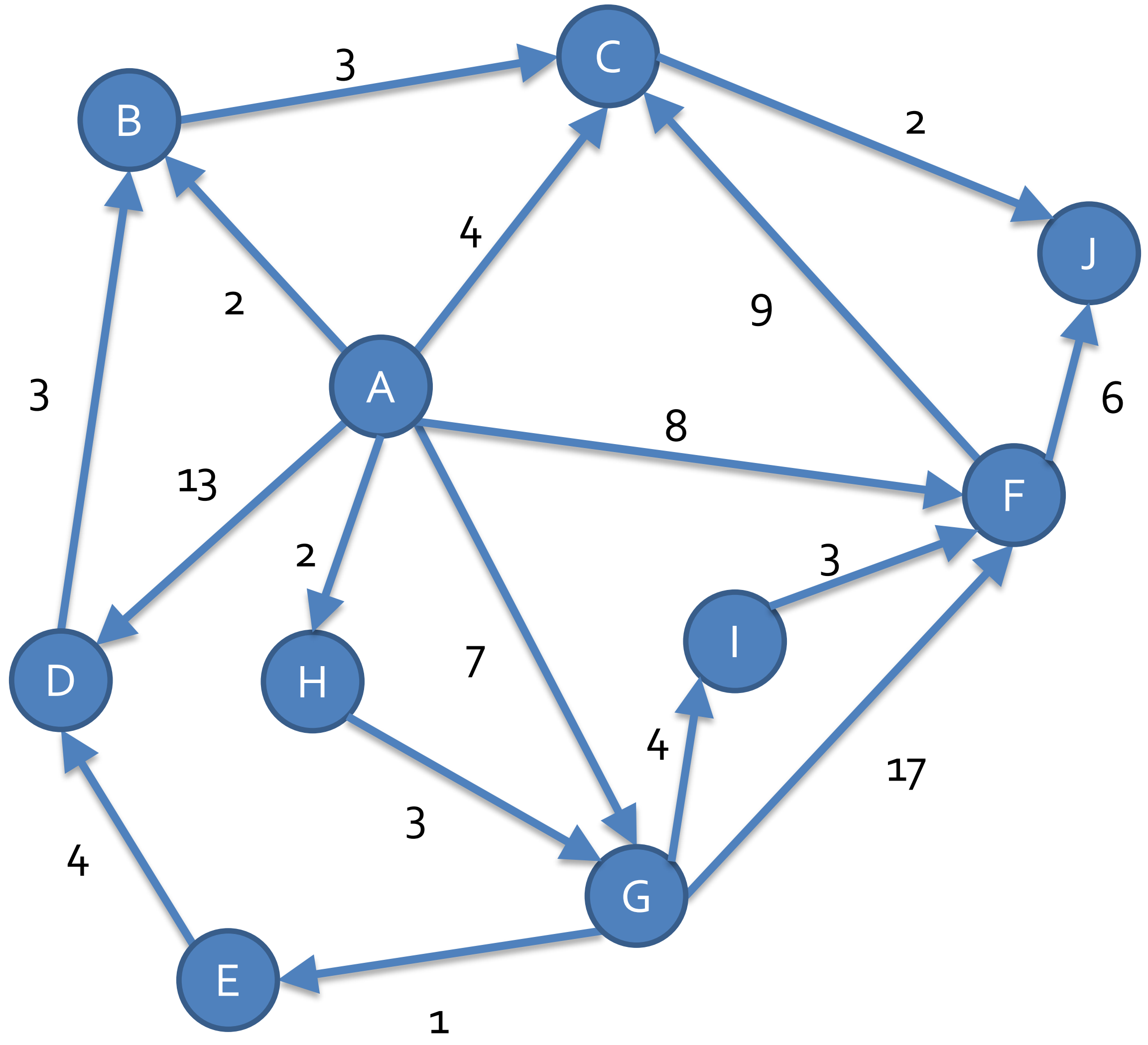
Designing the algorithm

- Let's first look at the **length** of the paths, not the actual paths
- We keep set ***S*** of vertices to which we have determined the true shortest-path distance
  - ***S*** is the explored part of the graph
- Then, we try to find the shortest new path by traveling from any node in the explored part ***S*** to any node ***v*** outside
- We update the distance to ***v*** and add ***v*** to ***S***
- Then, continue

Dijkstra's Algorithm

- Let $\boldsymbol{S}$ be the set of explored nodes
  - For each $\boldsymbol{u} \in \boldsymbol{S}$, we store a distance $\boldsymbol{d}(\boldsymbol{u})$
- Initially $\boldsymbol{S} = \{\boldsymbol{s}\}$ and $\boldsymbol{d}(\boldsymbol{s}) = 0$
- While $\boldsymbol{S} \neq \boldsymbol{V}$
  - Select a node $\boldsymbol{v} \notin \boldsymbol{S}$ with at least one edge from $\boldsymbol{S}$ for which $\boldsymbol{d'}(\boldsymbol{v}) = \min_{e=(u,v):u\in S} \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{l_e}$ is as small as possible
  - Add $\boldsymbol{v}$ to $\boldsymbol{S}$ and define $\boldsymbol{d}(\boldsymbol{v}) = \boldsymbol{d'}(\boldsymbol{v})$

Dijkstra's Algorithm Example



Reflections on Dijkstra's algorithm

- You can think of Breadth-First Search as a pulse expanding, layer by layer, through a graph from some starting node
- Dijkstra's algorithm is the same, except that the time it takes for the pulse to arrive is based not on the number of edges, but the lengths of the edges it has to pass through
- Because Dijkstra's algorithm expands from the starting point to whatever is closer, it grows like a blob
- There are algorithms that, under certain situations, can cleverly grow in the direction of the destination and will often take less time to find the path there

Minimum Spanning Trees

Minimum spanning tree

- We have a weighted, connected graph and we want to remove as many edges as possible such that:
  - The graph remains connected
  - The edges we keep have the smallest total weight
- This is the **minimum spanning tree** (MST) problem
- We can imagine pruning down a communication network so that it's still connected but only with the cheapest amount of wire total
- MST algorithms are also used as subroutines in other graph problems

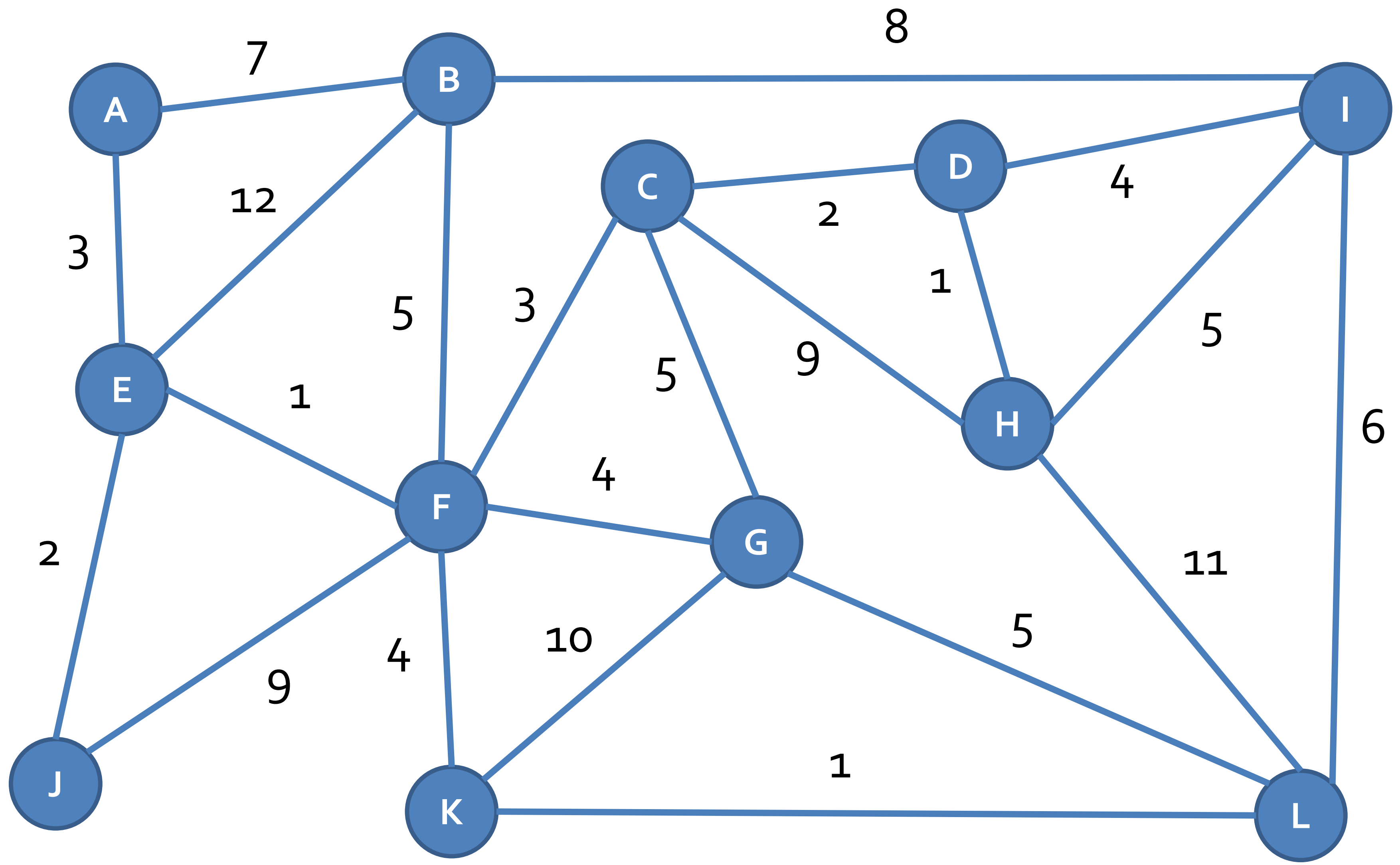
MST observations

- Assuming positive edge weights, the resulting graph is obviously a tree
  - If the graph wasn't connected, it wouldn't be a solution to our problem
  - If there was a cycle, we could remove an edge, make it cheaper, and still have connectivity

Approaches

- **Kruskal's algorithm:** Add edges to the MST in order of increasing cost unless it causes a cycle
- **Prim's algorithm:** Grow outward from a node, always adding the cheapest edge to a node that is not yet in the MST
- **Backwards Kruskal's algorithm:** Remove edges from the original graph in order of decreasing cost unless it disconnects the graph
- All three algorithms work!

MST example



Clustering

Clustering

- Imagine you have a set of objects
  - Photographs
  - Documents
  - Microorganisms
- You want to classify them into related groups
- Usually, you have some **distance function** that says how far away any two objects are
- You want to group together objects so that all the objects in a group are close

Notes about distance

- The distance function is usually defined between all points
  - If the points are in the plane or another Euclidean space, the distance could simply be the distance between them
  - A more flexible way to define distance is as weights on graph edges in a complete graph
- The distance between a point and itself is 0
- The distance between any two distinct points is greater than 0
- The distance between two points is symmetrical

Clustering by maximum spacing

- What if we want to divide our objects into *k* non-empty sets:
  - $C_1, C_2, \ldots, C_k$
- The **spacing** of this ***k***-clustering is the minimum distance between any pair of points in different clusters
- We want to find clusters with maximum spacing
  - There are other metrics to optimize your clusters on

Algorithm

- We don't want to group together objects that are far apart
- We sort all of the edges by weight and begin adding them back to our graph in order
- If an edge connects nodes that are already in the same cluster, we skip it
  - Thus, we don't make cycles
- We stop when we have ***k*** connected components

MST saves the day

- This algorithm is exactly Kruskal's algorithm
  - Add edges by increasing size, skipping ones that make a cycle
- We simply stop when we have ***k*** connected components instead of connecting everything
  - Alternatively, you can make the MST and delete the $\boldsymbol{k} - 1$ most expensive edges

Huffman Codes

Prefix codes

- We want to make an encoding such that the encoding of one letter is not a prefix of the coding of another letter
  - Such an encoding is called a prefix code
- If you have a prefix code, you can scan bits from left to right and output a letter as soon as it matches
- Example prefix code:
  - $a \to 11$
  - $b \to 01$
  - $c \to 001$
  - $d \to 10$
  - $e \to 000$

Optimal prefix codes

- If each letter ***x*** has a frequency f_x, with ***n*** letters total, nf_x gives the number of occurrences of ***x*** in a document
- Let ***code***(***x***) be the encoding of a letter ***x*** and ***S*** is the alphabet
- Total length of an encoding is:

$$\sum_{x \in S} nf_x|code(x)| = n\sum_{x \in S} f_x|code(x)|$$

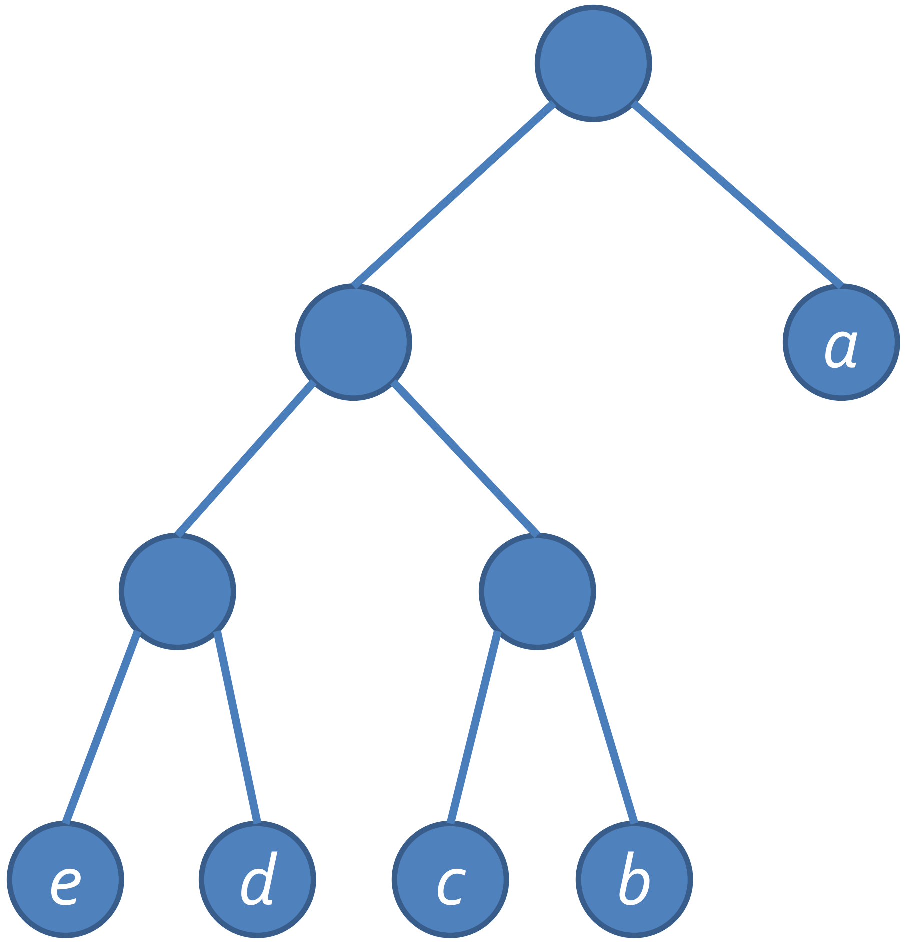
- An **optimal prefix code** minimizes average encoding length:

$$\sum_{x \in S} f_x|code(x)|$$

Algorithm design

- A key idea is that we can represent letters as leaves in a binary tree
  - Each left turn is a 0
  - Each right turn is a 1
- No letter will be the prefix of another
- Why?
- If a letter was the prefix of another, it would be on the path to the other letter, but every letter is a leaf

Prefix code tree example



$a \rightarrow 1$

$b \rightarrow 011$

$c \rightarrow 010$

$d \rightarrow 001$

$e \rightarrow 000$

Full binary trees

- Recall that a binary tree is a rooted tree in which each node has 0, 1, or 2 children
- A **full binary tree** is one in which every node that isn't a leaf has two children

How can we figure out the tree structure?

- We know that the binary tree will be full, but there are many full binary trees with ***n*** leaves
- Imagine that we had a full binary tree ***T**** that was an optimal prefix tree
- We know that the low frequency letters should appear at the deepest levels of the tree
- For letters ***y*** and ***z***, and corresponding nodes *node*(***y***) and *node*(***z***), if *depth*(*node*(***y***)) < *depth*(*node*(***z***)) then $f_y \geq f_z$.

We don't have the structure of *T**

- If we did, we could label it by putting the highest frequency letters in the highest levels of the tree and then going down, level by level
- Instead, we work backwards
- The lowest frequency letter must be at the deepest leaf in the tree, call it ***v***
- Since this is a full binary tree, ***v*** must have a sibling ***w***

Algorithm description

- Take the two lowest frequency letters ***y*** and ***z***.
- Since they are neighbors in a full tree, we can stick them together and treat them like a meta-letter ***yz*** with the sum of their frequencies.
- Recursively repeat until everything is merged together.

Algorithm

- If ***S*** has two letters then
  - Encode one with 0 and the other with 1
- Else
  - Let ***y*** and ***z*** be the two lowest-frequency letters
  - Form a new alphabet ***S'*** by deleting ***y*** and ***z*** and replacing them with a new letter ***w*** of frequency $f_y + f_z$
  - Recursively construct a prefix code for ***S'*** with tree ***T'***
  - Define a prefix code for ***S*** as follows:
    - Start with ***T'***
    - Take the leaf labeled ***w*** and add two children below it labeled ***y*** and ***z***

Recurrence Relations

Divide and conquer

- **Divide and conquer** algorithms are ones in which we divide a problem into parts and recursively solve each part
- Then, we do some work to combine the solutions to each part into a final solution
- Divide and conquer algorithms are often simple
- However, their running time can be challenging to compute because recursion is involved

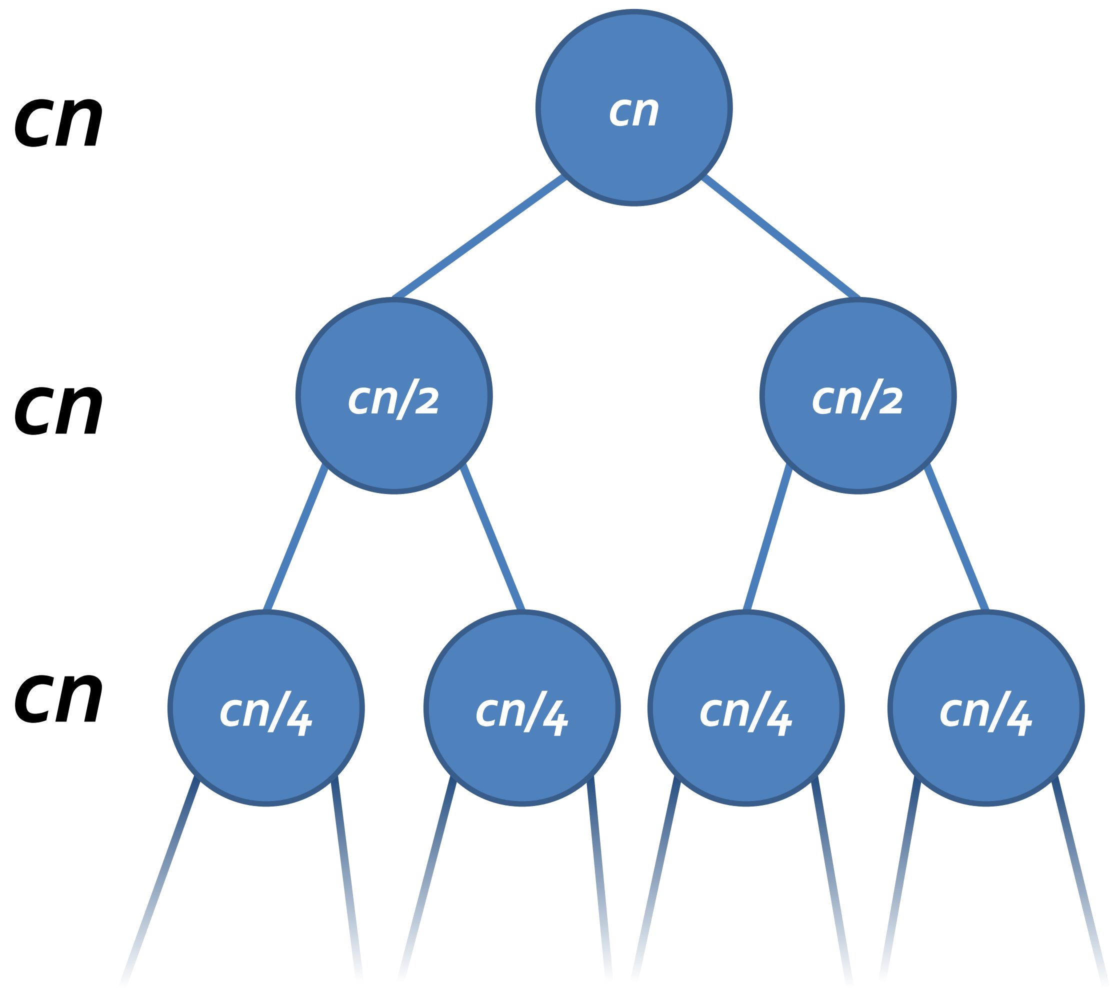
Mergesort algorithm

- If there are two elements in the array or fewer then
  - Make sure they're in order
- Else
  - Divide list into two halves
  - Recursively merge sort the two halves
  - Merge the two sorted halves together into the final list

Time for mergesort

- The algorithm is simple, but recursive
- We'll use ***T*(*n*)** to describe the total running time recursively
  - $T(n) \leq c, \qquad n \leq 2$
  - $T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \qquad n > 2$
- Is it really the same constant ***c*** for both?
  - No, but it's an inequality, so we just take the bigger one

Intuition about mergesort recursion



- Each time, the recursion cuts the work in half while doubling the number of problems
  - The total work at each level is thus always ***cn***
- To go from ***n*** to 2, we have to cut the size in half ($\log_2$ ***n***) – 1 times

Recursively defined sequences

- Defining a sequence recursively as with Mergesort is called a **recurrence relation**
- The **initial conditions** give the starting point
- Example:
  - Initial conditions
    - $T(0) = 1$
    - $T(1) = 2$
  - Recurrence relation
    - $T(k) = T(k-1) + kT(k-2) + 1$, for all integers $k \geq 2$
  - Find $T(2)$, $T(3)$, and $T(4)$

Finding explicit formulas by iteration

- We want to be able to turn recurrence relations into explicit formulas whenever possible
- Often, the simplest way is to find these formulas by **iteration**
- The technique of iteration relies on writing out many expansions of the recursive sequence and looking for patterns
- That's it

Employing outside formulas

- Intelligent pattern matching gets you a long way
- However, it is sometimes necessary to substitute in some known formula to simplify a series of terms
- Recall
  - Geometric series: $1 + \boldsymbol{r} + \boldsymbol{r}^2 + \ldots + \boldsymbol{r}^{\boldsymbol{n}} = (\boldsymbol{r}^{\boldsymbol{n}+1} - 1)/(\boldsymbol{r} - 1)$
  - Arithmetic series: $1 + 2 + 3 + \ldots + \boldsymbol{n} = \boldsymbol{n}(\boldsymbol{n} + 1)/2$

Further recurrence relations

- We have seen that recurrence relations of the form $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ are bounded by O(***n*** log ***n***)
- What about $T(n) \leq qT\left(\frac{n}{2}\right) + cn$ where ***q*** is bigger than 2 (more than two sub-problems)?
- There will still be $\log_2$***n*** levels of recursion
- However, there will not be a consistent ***cn*** amount of work at each level

Converting to summation

- In general, it's

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j$$

- This is a geometric series, where $r = \frac{q}{2}$

$$T(n) \leq cn\left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r - 1}\right)$$

Final bound

$$T(n) \leq cn\left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r - 1}\right)$$

- Since $r - 1$ is a constant, we can pull it out
- $T(n) \leq \left(\frac{c}{r-1}\right) n r^{\log_2 n}$
- For $a > 1$ and $b > 1$, $a^{\log b} = b^{\log a}$, thus $r^{\log_2 n} = n^{\log_2 r} = n^{\log_2(q/2)} = n^{(\log_2 q)-1}$
- $T(n) \leq \left(\frac{c}{r-1}\right) n \cdot n^{(\log_2 q)-1} \leq \left(\frac{c}{r-1}\right) n^{\log_2 q}$ which is $O\left(n^{\log_2 q}\right)$

Counting Inversions

Ranking similarity

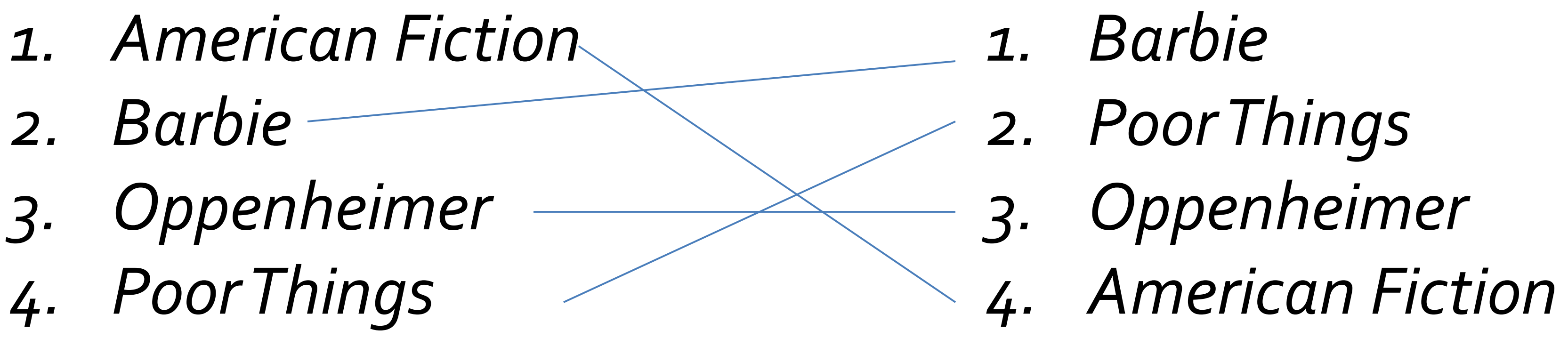
- What if we wanted to measure the similarity of one ranking to another ranking?
- **Inversions** are pairs of elements that are out of order in one ranking with respect to the other
- Formally, for indices $\boldsymbol{i} < \boldsymbol{j}$, there's an inversion if ranking $\boldsymbol{r}_i > \boldsymbol{r}_j$

Minimum and maximum inversions

- If two rankings are the same, they would have no inversions
- If two rankings were sorted in opposite directions, they would have $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$ inversions

Visualization of inversions

- You can visualize inversions as the number of line segments crossings if you match up items in one list with the other



- A total of 4 inversions

Can we do better than O(n^2)?

- Of course!
- We can borrow from the Mergesort algorithm
- Divide the problem in half
- Then, we will get the number of inversions in the first half and in the second half
- Are we done?
  - No, we also have to count the inversions between the first half and the second half
  - Those are exactly those elements in the first half that are bigger than elements from the second half
  - We can find those during the merge process

Merge-and-Count(*A*, *B*)

- Maintain a ***Current*** pointer into each list, initialized to point to the front elements
- Set ***Count*** = 0
- While both lists have elements
  - Let $\boldsymbol{a_i}$ and $\boldsymbol{b_j}$ be the elements pointed to by the ***Current*** pointer
  - Append the smaller one to the output list
  - If $\boldsymbol{b_j}$ is smaller then
    - Increment ***Count*** by the number of elements left in ***A***
  - Advance the ***Current*** pointer in the list that had the smaller element

Sort-and-Count(*L*)

- If the list has one element then
  - Return 0 inversions and the list ***L***
- Else
  - Divide the list into two halves:
    - ***A*** has the first $\left\lceil \frac{n}{2} \right\rceil$ elements
    - ***B*** has the remaining $\left\lfloor \frac{n}{2} \right\rfloor$ elements
  - (***inversions**$_A$*, ***A***) = Sort-and-Count(***A***)
  - (***inversions**$_B$*, ***B***) = Sort-and-Count(***B***)
  - (***inversions***, ***L***) = Merge-and-Count(***A***, ***B***)
  - Return ***inversions*** + ***inversions**$_A$* + ***inversions**$_B$* and sorted list ***L***

Running time

- Since Merge-and-Count is bounded by O(***n***), the running time for Sort-and-Count is clearly:
  - $T(1) \leq c$
  - $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$, for $n \geq 2$
- By the same analysis as for Mergesort, ***T***(***n***) is O(***n*** log ***n***)

Closest Pair of Points

Closest pair of points

- Imagine you have a set of points in a 2D plane
- How do you find the pair of points that's closest?
- This is a fundamental problem in the area of **computational geometry**
- As usual, you could look at all pairs of points

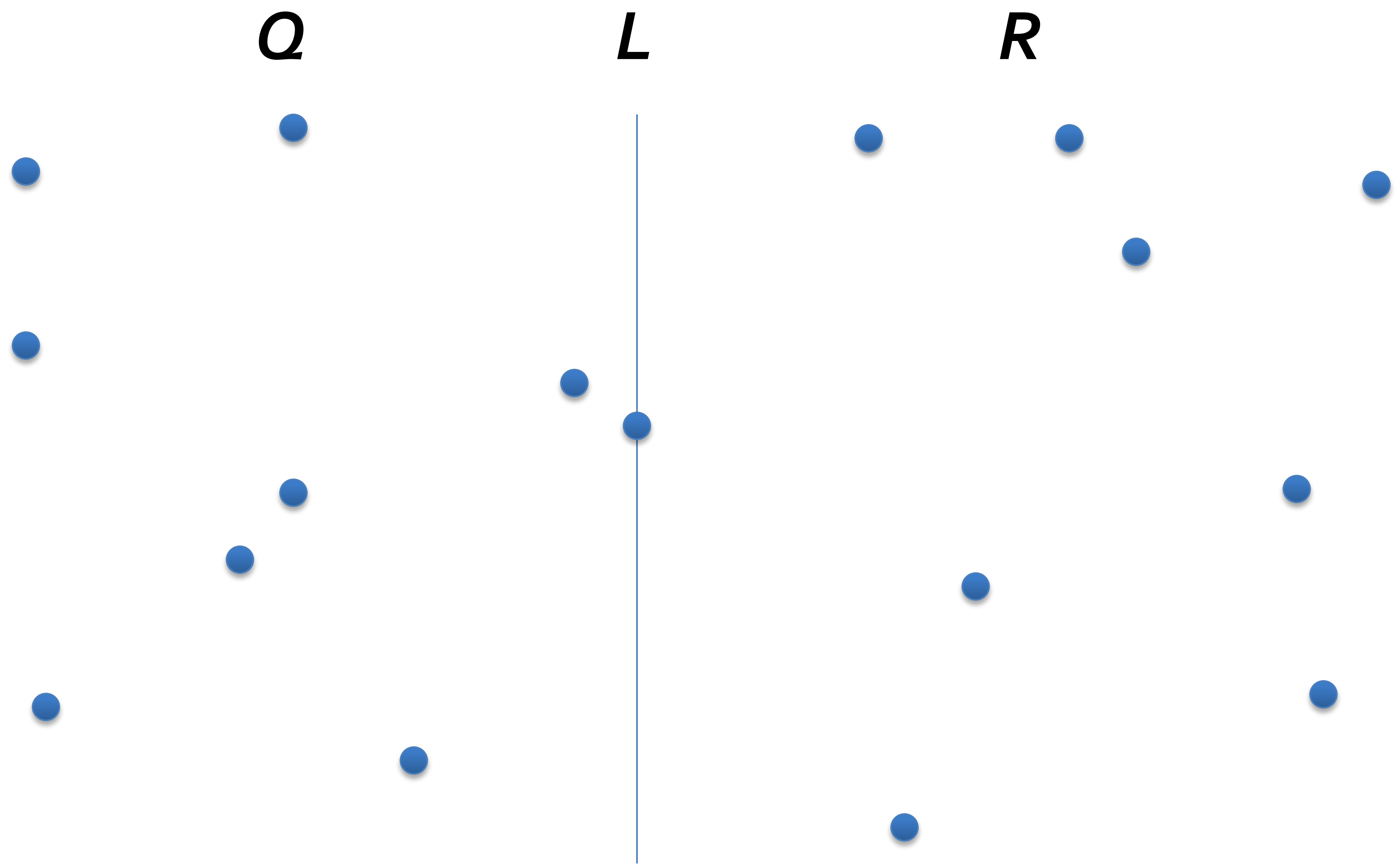
Designing the algorithm

- To make things simpler, we assume that no two points have the same ***x***-coordinate or ***y***-coordinate
- Think about a one-dimensional approach:
  - Sort the list by ***x***-value
  - The two closest points must be next to each other in the list

Divide

- Since the name of the chapter is divide and conquer, that's what we do
- First, sort all of the points by increasing ***x***-values, calling this list P_x
- Then, sort all of the points by increasing ***y***-values, calling this list P_y
- Find the median point in P_x and drop a line through it, dividing the points into those with smaller ***x*** (set ***Q***) and larger ***x*** (set ***R***)
- Recursively find the closest pair of points on the left side and the closest pair of points on the right side

Divide points



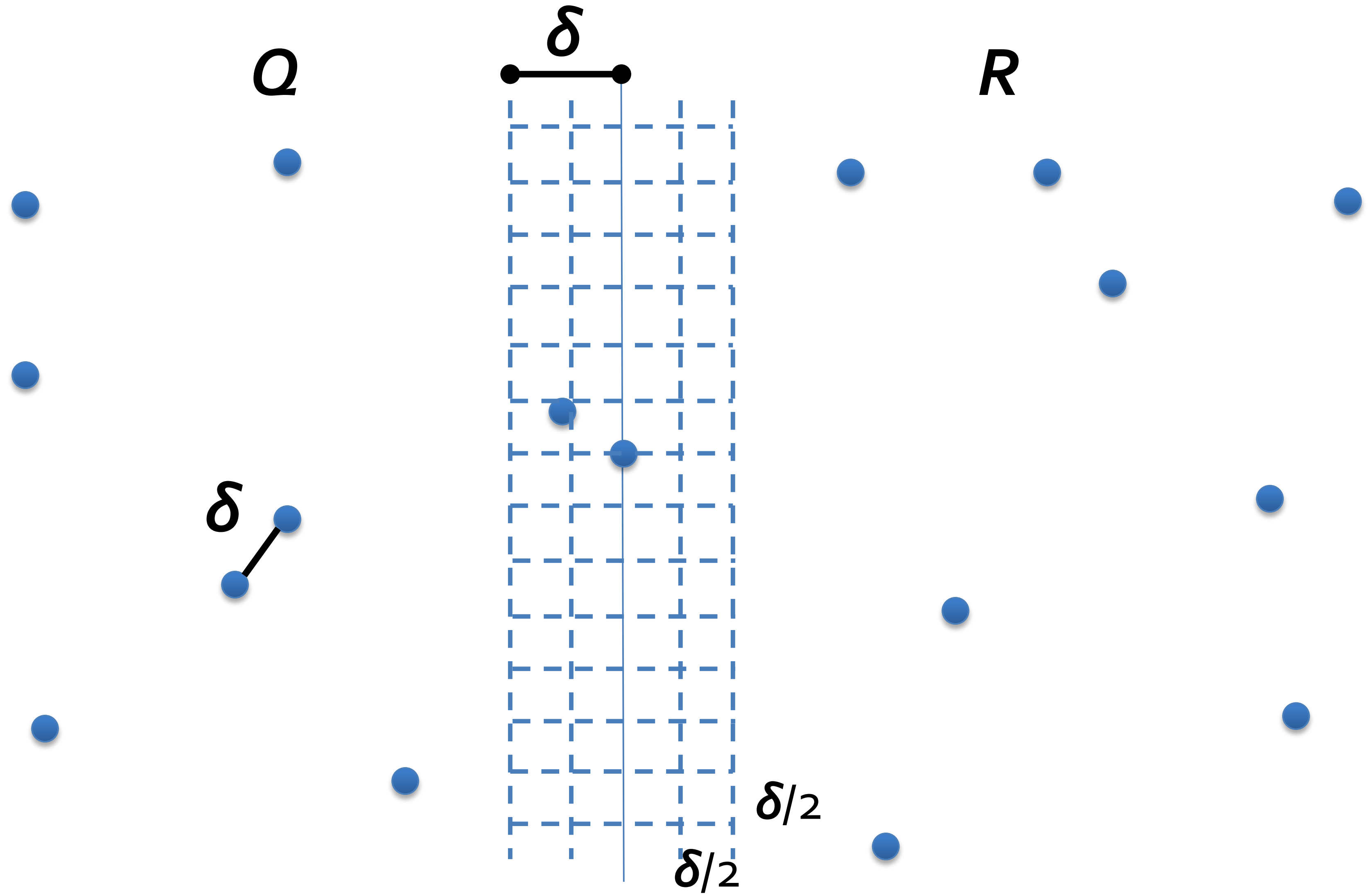
...and...

- We have ~~magically~~ recursively found the closest pair of points in ***Q*** and the closest pair in ***R***
  - Between those two pairs, let's say the closest has distance $\boldsymbol{\delta}$
- But what if the closest pair straddles ***L***, with one point in ***Q*** and the other in ***R***?
- We do a linear scan of $\boldsymbol{P_y}$, the list of points sorted by ***y*** values, making a new ***y***-sorted list of points $\boldsymbol{S_y}$ whose ***x***-coordinate is within $\boldsymbol{\delta}$ of ***L***

…conquer!

- We scan through the list $\boldsymbol{S_y}$
- For each element, we compute the distance between it and the next 15 elements
- We find the closest distance
- If the closest distance is smaller than $\boldsymbol{\delta}$, that's the true closest pair
- Otherwise, we use the smaller of the pairs from $\boldsymbol{Q}$ and $\boldsymbol{R}$

Divide points



Running time

- Pre-processing:
  - Sort the points by ***x***: O(***n*** log ***n***)
  - Sort the points by ***y***: O(***n*** log ***n***)
- Recursion:
  - If there are three or fewer points, find the closest pair by comparing all pairs
  - Otherwise, divide into sets ***Q*** and ***R***: O(***n***) time
  - Make lists Q_x, Q_y, R_x, and R_y, giving the points in ***Q*** and ***R*** sorted by ***x*** and ***y***, respectively: O(***n***) time
  - Construct S_y: O(***n***) time
  - For every point in S_y (of which there can only be ***n***), compute the distance to the next 15 points: O(***n***)
- $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ which is $O(n \log n)$

Integer Multiplication

We need a trick

- We want $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0$
- What if we compute
  - $a = (x_1 + x_0) \cdot (y_1 + y_0)$
    $= x_1 y_1 + x_0 y_1 + x_1 y_0 + x_0 y_0$
  - $b = x_1 y_1$
  - $c = x_0 y_0$
- Then, $b \cdot 2^n + (a - b - c) \cdot 2^{\frac{n}{2}} + c =$
- $x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0$

Running time

- We do two additions before the multiplies: O(***n***)
- We do three recursive multiplies of ***n***/2-bit numbers
- We do two additions and two subtractions after the multiplies: O(***n***)
- $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$
- Which is $O\left(n^{\log_2 3}\right) \approx O(n^{1.59})$, which is better!

Master Theorem

Basic form of the Master Theorem

- For recursion that on a problem size ***n*** that:
  - Makes ***a*** recursive calls
  - Divides the total work by ***b*** for each recursive call
  - Does ***f***(***n***) non-recursive work at each call
- Its running time can be given in the following form, suitable for use in the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1$$

Case 1

- If $f(n)$ is $O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant $\epsilon > 0$, then

$$T(n) \text{ is } \Theta\left(n^{\log_b(a)}\right)$$

Case 2

- If $f(n)$ is $\Theta\left(n^{\log_b(a)} \log^k n\right)$ for some constant $k \geq 0$, then

$$T(n) \text{ is } \Theta\left(n^{\log_b(a)} \log^{k+1} n\right)$$

Case 3

- If $f(n)$ is $\Omega\left(n^{\log_b(a)+\epsilon}\right)$ for some constant $\epsilon > 0$, and if
$$af\left(\frac{n}{b}\right) \leq cf(n)$$
for some constant $c < 1$ and sufficiently large n, then
$$T(n) \text{ is } \Theta(f(n))$$

Example Problems

Huffman Codes

- Make a Huffman encoding for the following alphabet, given the frequencies of each letter:

a	0.04
b	0.18
c	0.23
d	0.21
e	0.10
f	0.02
g	0.21

Recursive sequence example

- $g_k = \frac{g_{k-1}}{g_{k-1}+2}$ for all integers $k \geq 1$
- $g_0 = 1$

- Give an explicit formula for this recurrence relation
- **Hint:** Use the method of iteration

Sample master theorem problem

- $T(n) = 9T(n/3) + 13n^2 \log^3 n$

- Sometimes it helps to think about how I create questions:
  - Generate a recurrence relation that fits Case 1
  - Generate a recurrence relation that fits Case 2
  - Generate a recurrence relation that fits Case 3

Upcoming

Next time…

- Exam 2!
- Review Chapters 4 and 5 and the Master Theorem

Reminders

- Finish Homework 4
  - **Due tonight before midnight!**
- Study for Exam 2
  - Wednesday in class