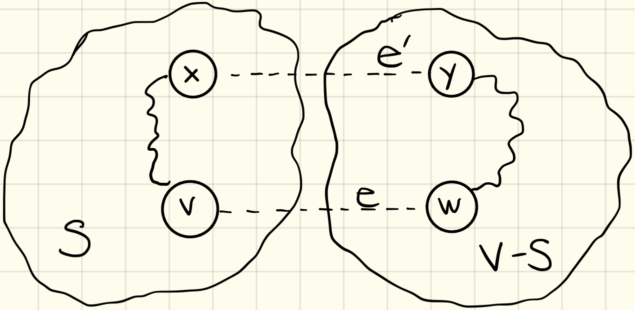
The <u>Cut Property</u> says which edges <u>must</u> appear in some MST.

Is there a way to guarantee the opposite?

(4.20) <u>The Cycle Property</u>: Let $G=(V,E)$ be a weighted graph with distinct edge weights and let C be some cycle in G. Then if $e=\{v,w\}$ is the heaviest edge in C, it is not in <u>any</u> MST of G.

Proof: Assume such a G and C, and let T be a spanning tree of G that contains e. Consider removing e from T. This partitions T into 2 disjoint components, S (containing v) and $V-S$ (containing w). In the original graph, because there was a cycle, there was some other path that connected v and w. Consider the following diagram:



$C = v, \ldots, x, y, \ldots, w, v$

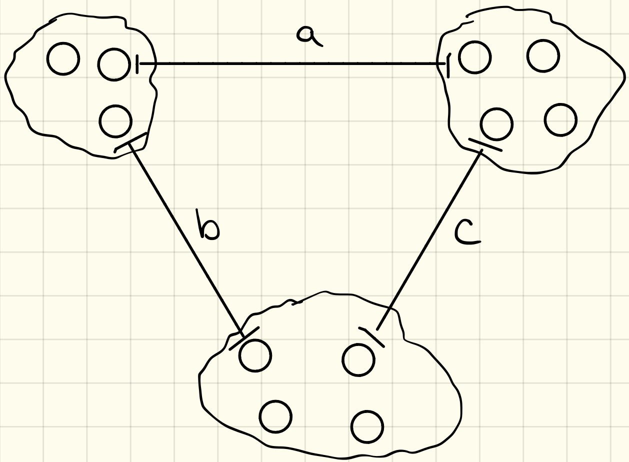
wlog consider labeling the nodes participating in the cycle as above. Since v and x are in the same component, there exists some v-x path in S. Likewise for y and w. We have removed e from T, but we can re-connet T by adding e'. Since we removed e, then adding e' won't create cycles. Further, $T - \{e\} \cup \{e'\}$ is a spanner. Finally, since e was the heaviest edge in the cycle C, then $T - \{e\} \cup \{e'\}$ is a spanning tree with strictly lesser weight. So, e cannot be in any MST of G. ∎

Clustering : An application of MST

Given : A set of n items $p_1, p_2, \ldots, p_n$ and a "distance" function $d(p_i, p_j)$ that allows us to measure the distance / dissimilarity between any pair of objects. Note: we need that $d(p_i, p_i) = 0$ and $d(p_i, p_j) > 0$ for $p_i \neq p_j$ and $d(p_i, p_j) = d(p_j, p_i)$, but $d(\cdot, \cdot)$ <u>need not</u> be a metric.

Find : k non-empty groups partitioning the n items so that the <u>minimum</u> distance between different groups is maximized.

E.g. :



Idea:
- Maintain clusters as a set of connected components in a graph.
- Iteratively combine clusters containing the two closest items by adding an edge between them.
- Stop when there are k clusters.

Note: This is exactly Kruskal's algorithm with early stopping.

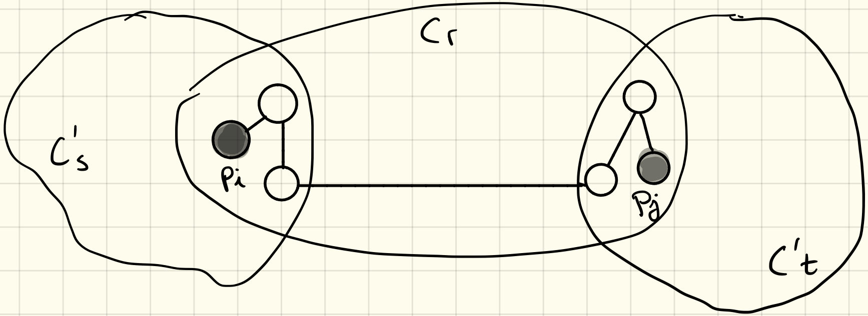
This is often called "single-linkage, agglomerative clustering"

Theorem (MST clust): The MST clustering algo. produces a set of clusters $\mathcal{C} = \{C_i\}_{i=1}^{k}$ with a maximum spacing.

Proof: First, observe that stopping Kruskal's early leads to k clusters, this is equivalent to taking the full MST and removing the $k-1$ most expensive edges. The spacing of $\mathcal{C}$ is the length of this $(k-1)^{st}$ most expensive edge.

Let $\mathcal{C}'$ be some other k clustering. $\mathcal{C}'$ must have the same or smaller separation as $\mathcal{C}$, why?

Since $\mathcal{C} \neq \mathcal{C}'$, there must be some pair p_i, p_j that are in the same cluster C_r in $\mathcal{C}$ but in different clusters C'_s, C'_t in $\mathcal{C}'$.



Since p_i, p_j are in C_r, there is a path P_{ij} between them with all edges $\leq d$. Some edge of this path must pass between C'_s and C'_t, so the separation of $\mathcal{C}'$ is at most d. ∎

Divide and Conquer

- A different algorithm design technique than greedy.
- Decompose the problem into subproblems - solve recursively - recompose
- Will start with how to analyze using recurrence relations and then cover some D+C algorithms.
- Recurrence relations are useful to analyze running times even when algos are not efficient.

Recall the Fibonacci Sequence:

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = F_2 = 1$$

consider a naive impl of fib():

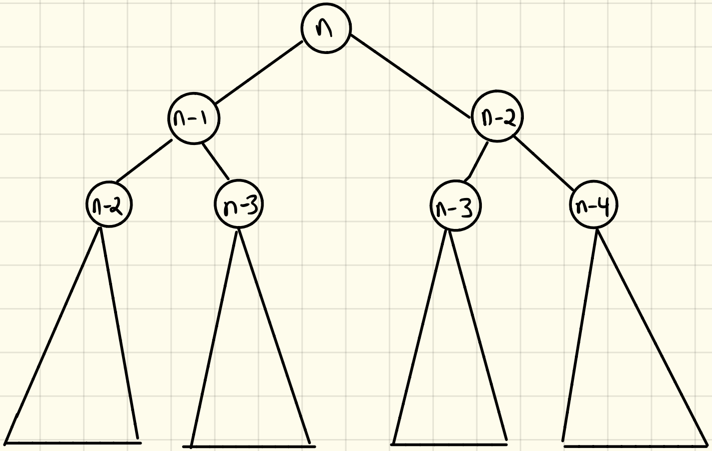
```
fib(n):
    if n = 0: return 0
    if n = 1 or n = 2: return 1
    return fib(n-1) + fib(n-2)
```

How can we analyze the running time of Fib(·)?

- We know that $T(n) = T(n-1) + T(n-2) + O(1)$
  - That is, the time to compute fib(n) is the time to compute fib(n-1), fib(n-2) and add them (which we assume above is constant).

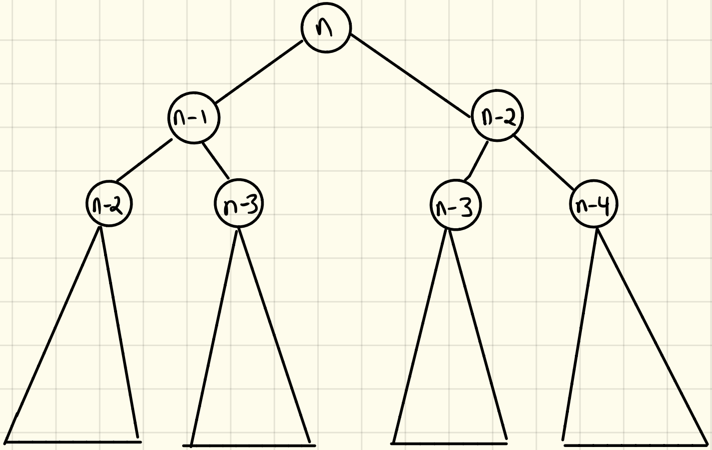
- What does the "tree" of recursive calls look like?

Fib tree



Leaves take constant time

Fib tree



leaves take <u>constant</u> time

What is the depth of this tree? $\Rightarrow$ def. bounded by n

How many leaves? $\Rightarrow \leq 2^n$

Do constant work per leaf and per internal node. $\Rightarrow fib(n) \in O(2^n)$

<u>But</u> is this bound tight? How fast does the rightmost branch fall off compared to the leftmost?

$\rightarrow$ for fib(n), we can do better than $O(2^n)$

(1) The root node has value fib(n).
(2) Each leaf contributes exactly 1 to this sum → fib(n) leaves
(3) This is a binary tree, so # internal nodes is # leaves − 1 = fib(n) − 1
(4) Total # of nodes is $(2 \cdot \text{fib}(n)) - 1 = O(\text{fib}(n))$
→ it turns out that this is $O(\varphi^n) \approx O(1.618^n)$

Drawing a recursion tree is a common way to analyze the runtime of recursive (D+C) algorithms.

Lets try with another:

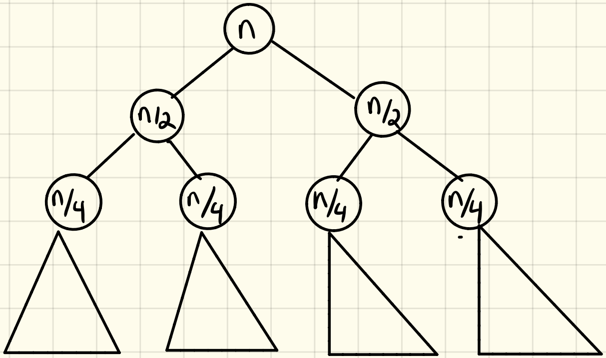
```
MergeSort(L):
    if |L| = 2 : return [min(L), max(L)]
    else:
        L1 = MergeSort(L[0 : ⌊|L|/2⌋])
        L2 = MergeSort(L[⌊|L|/2⌋+1 : |L|-1])
        return Combine(L1, L2)
```

this is a simple merge of 2 sorted lists, takes $O(|L1| + |L2|)$ time

Total time $T(n) \le 2T(n/2) + cn$, want an upper bound

- 2 methods
  - (A) Recursion tree
  - (B) Guess & check (via induction)

(A)



cn work

$2\left(\frac{cn}{2}\right)$ work

$4\left(\frac{cn}{4}\right)$ work

Steps:

(1) write out the work done at each level
(2) find the height of the tree
(3) sum over all levels

(1) Here, we do cn work per level
(2) Each level reduces n by a factor of $2 \rightarrow$ at most $\lg n$ levels
(3) Sum:

$$\sum_{i=1}^{\lg n} cn = \lg n \cdot cn = c(n \cdot \lg n) = O(n \lg n) \text{ work}$$

(B) Substitution

steps:

(1) Show $T(K) \leq f(K)$ for some small K

(2) Assume $T(K) \leq f(K)$ for all $K < n$

(3) Show $T(n) \leq f(n)$

Consider this for Merge Sort

$$T(n) \leq 2T(n/2) + cn$$

Base Case : $T(2) \leq 2 \cdot c \lg 2$

IH : $T(K) \leq c \cdot m \lg m \qquad m < n$

IS :

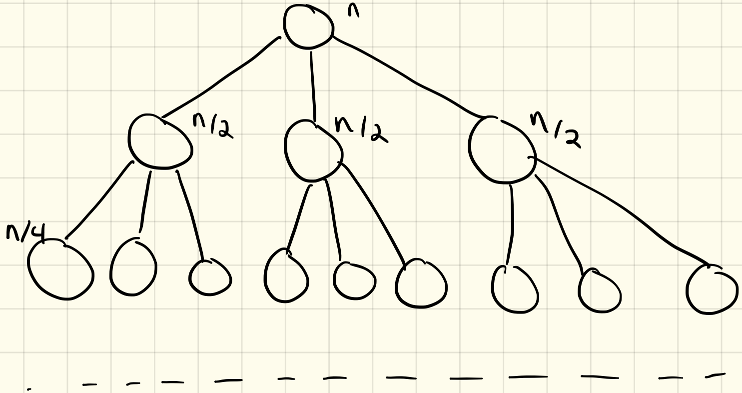
$$\begin{aligned}
T(n) &\leq 2T(n/2) + cn \\
&\leq 2c(n/2) \lg (n/2) + cn \\
&= cn \cdot \lg (n/2) + cn \\
&= cn\,[\lg(n) - 1] + cn \\
&= cn \lg(n) - cn + cn \\
&= cn \lg(n)
\end{aligned}$$

$\blacksquare$

Mergesort solves 2 equal sized subproblems, but what if we divide into more or fewer parts?

Consider $T(n) \le q\,T(n/2) + cn$ (where $q > 2$)

e.g. $q=3$



cn

$3\left(\frac{cn}{2}\right)$

$9\left(\frac{cn}{4}\right)$

Still $\lg(n)$ levels, and each does $q^j\left(\frac{cn}{2^j}\right)$ work $= (q/2)^j\, cn$ work

Summing over all levels:

$$T(n) \le \sum_{j=0}^{\lg(n)-1} (q/2)^j\, cn = cn \underbrace{\sum_{j=0}^{\lg(n)-1} (q/2)^j}_{\text{geometric sum with } r>1}$$

$r = (q/2)$ $\quad T(n) \le cn\left(\frac{r^{\lg(n)} - 1}{r-1}\right) \le cn\left(\frac{r^{\lg(n)}}{r-1}\right)$

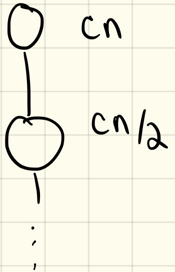
$$T(n) \le \left(\frac{c}{r-1}\right) n\, r^{\lg(n)}$$

- for all $a, b > 1$ $\quad a^{\log b} = b^{\log a}$, so $r^{\log n} = n^{\log r}$

$$T(n) \le \left(\frac{c}{r-1}\right) n \cdot n^{\lg(r)} = \left(\frac{c}{r-1}\right) n \cdot n^{\lg(q/2)} = \left(\frac{c}{r-1}\right) n \cdot n^{\lg(q)-1}$$

$$\le \left(\frac{c}{r-1}\right) n^{\lg(q)} = O\left(n^{\lg(q)}\right)$$

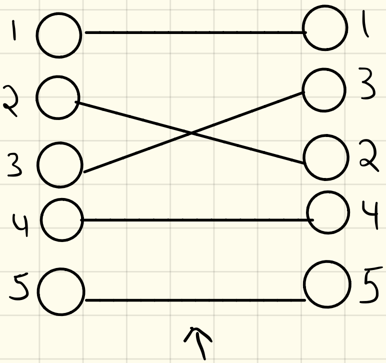
What about for $q = 1$?



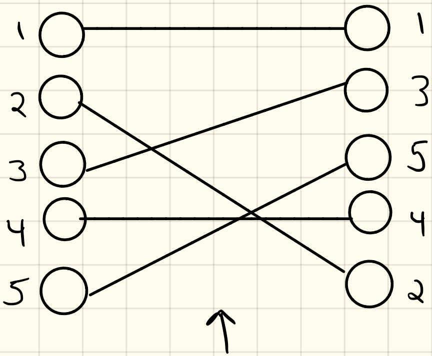
Turns out to be $O(n)$, try to show this.

Problem: Counting Inversions

- Suppose customers rank a list of movies
- How can we compare the similarity of these rankings?



similar



dissimilar

One measure is # of <u>inversions</u>

- assume one ranking is $1, 2, \ldots, n$
- let other be $a_1, a_2, \ldots, a_n$

- An <u>inversion</u> is a pair (i, j) s.t. $i < j$ but $a_j < a_i$.

- two identical rankings have 0 inversions
- How many for opposite rankings? . . . $\binom{n}{2}$

How can we count inversions quickly?

- Check every pair? $O(n^2)$

- Some orderings may have $O(n^2)$ inversions, so, to do better, we will have to count multiple inversions at the same time.

- A smart D&C algo. will give us $O(n \lg n)$

Suppose I had a "recursive" algo that would tell you for $a_1, \dots, a_n$ # of inversions in each half:

$$\underbrace{\boxed{a_1, \;.\;.\;.\;.\;, a_{n/2}}}_{\text{Inv } 1} \qquad \underbrace{\boxed{a_{(n/2)+1}, \;.\;.\;.\;, a_n}}_{\text{Inv } 2}$$

# of inversions in each half

What inversions are missed by simply taking Inv 1 + Inv 2?

- The inversions crossing the split! (half crossing inversions).

Consider the following alg.

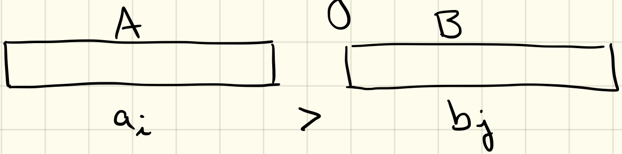
```
SortAndCount(L):
    if |L| = 1: return 0, L
    A, B = first + second halves of L

    invA, sortedA = SortAndCount(A)
    invB, sortedB = SortAndCount(B)
    crossInv, sortedL = MergeAndCount(sortedA, sortedB)

    return invA + invB + crossInv, sortedL
```

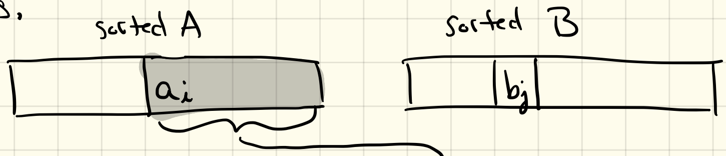
Note: Sorting happens as a byproduct of this algorithm

Half-crossing inversions



What if each Sublist is sorted?

- If we find some a_i, b_j with $a_i > b_j$, we can <u>infer</u> many other inversions.



Suppose $a_i > b_j$, then all items here are <u>also</u> larger than b_j but we can obtain # of items in the shaded area in constant time.

```
            sorted
             ↓ ↓
MergeAndCount(A, B):
    a=b=crosscount=0 , outList=[]
    while a < |A| and b < |B|:
        next = min(A[a], B[b])
        outList.append(next)
        If B[b] = next
            b=b+1
            crosscount = crosscount + |A|-a
        else
            a=a+1
    End While
    append the non-empty list to outList
    return crossCount, outList
```

- Note: MergeAndCount takes $O(n)$ time

- What is the running time of SortAndCount?

  - Breaks the problem into 2 halves, solves recursively; merging is $O(n)$.

$$T(n) \le 2T(n/2) + cn$$

  - we have seen exactly this recurrence before. It solves to:

$$T(n) \in O(n \lg n).$$