The Cut Property says which edges must appear in some MST. Is there a way to guarantee the opposite?

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

![Diagram](image.png)

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

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 $T - e \cup e'$ has 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 \(d(\cdot, \cdot)\) need not be a metric.

Find: \(k\) non-empty groups partitioning the \(n\) items so that the minimum 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 $C = \sum_{i=1}^{k} C_i$ 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 \( C \) is the length of this \((k-1)^{st}\) most expensive edge.

Let \( C' \) be some other \( k \) clustering. \( C' \) must have the same or smaller separation as \( C \), why?

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

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 \( C' \) 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 \( \text{fib}(n) \):

\[
\text{fib}(n) : \\
\text{if } n = 0 : \text{ return } 0 \\
\text{if } n=1 \text{ or } n=2 : \text{ return } 1 \\
\text{return } \text{fib}(n-1) + \text{fib}(n-2)
\]
How can we analyze the running time of Fib\((n)\)?

- 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
What is the depth of this tree? ⇒ def. bounded by \( n \)
How many leaves? ⇒ \( \leq 2^n \)

Do constant work per leaf and per internal node. ⇒ \( \text{Fib}(n) \in O(2^n) \)

But is this bound tight? How fast does the rightmost branch fall off compared to the leftmost?
⇒ for \( \text{fib}(n) \), we can do better than \( O(2^n) \)
1. The root node has value $\text{fib}(n)$.
2. Each leaf contributes exactly 1 to this sum, $\Rightarrow \text{fib}(n)$ leaves.
3. This is a binary tree, so $\#$ internal nodes is $\#$ leaves $-1 = \text{fib}(n) - 1$.
4. Total $\#$ of nodes is $(2 \cdot \text{fib}(n)) - 1 = O(\text{fib}(n))$.
   $\Rightarrow$ it turns out that this is $O(\phi^n) \approx O(1.618^n)$.

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

Let's try with another:

Merge Sort $(L)$:

- if $|L| = 2$ : return $[\min(L), \max(L)]$
- else:
  - $L_1 = \text{Merge Sort} (L[0: \lfloor L/2 \rfloor])$
  - $L_2 = \text{Merge Sort} (L[\lfloor L/2 \rfloor + 1: |L| - 1])$
  - Return Combine $(L_1, L_2)$

This is a simple merge of 2 sorted lists, takes $O(|L_1| + |L_2|)$ time.
Total time $T(n) \leq 2 T(n/2) + cn$, want an upper bound

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

(A)

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 $\log n$ levels
(3) Sum: $\sum_{i=1}^{\log n} cn = \log n \cdot cn = c(n \cdot \log n) = O(n \log n)$ 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: $T(2) \leq 2 \cdot c \cdot \log 2$

IH: $T(k) \leq c \cdot m \cdot \log m \quad m < n$

IS:

$T(n) \leq 2T(n/2) + cn$
$\leq 2c \cdot (n/2) \cdot \log (n/2) + cn$
$= cn \cdot \log (n/2) + cn$
$= cn \cdot \left[ \log(n) - 1 \right] + cn$
$= cn \log(n) - cn + cn$
$= cn \log(n)$
Mergesort solves 2 equal sized subproblems, but what if we divide into more or fewer parts?

Consider \( T(n) \leq q \cdot T(n/2) + cn \) (where \( q > 2 \))

e.g. \( q = 3 \)

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

Summing over all levels:

\[
\frac{T(n)}{cn} \leq \sum_{j=0}^{\lg(n)-1} \left( \frac{q}{2} \right)^j \cdot cn = cn \cdot \left( \sum_{j=0}^{\lg(n)-1} \left( \frac{q}{2} \right)^j \right)
\]

geometric sum with \( r > 1 \)
\[ r = \left( \frac{8}{2} \right) \]

\[ T(n) \leq c n \left( \frac{r \log(n)}{r-1} \right) \leq c n \left( \frac{\log(n)}{r-1} \right) \]

\[ T(n) \leq \left( \frac{c}{r-1} \right) n \log(n) \]

- For all \( a, b > 1 \), \( a^{\log b} = b^{\log a} \), so \( r^{\log n} = n^{\log r} \)

\[ T(n) \leq \left( \frac{c}{r-1} \right) n \cdot n \log(r) = \left( \frac{c}{r-1} \right) n \cdot n \log(\frac{8}{2}) = \left( \frac{c}{r-1} \right) n \cdot n \log(8) - 1 \]

\[ \leq \left( \frac{c}{r-1} \right) n \log(8) = \mathcal{O}(n \log(8)) \]

What about for \( q = 1 \)?

\[ 0 \]

\[ c n/2 \]

\[ ; \]

Turns out to be \( \mathcal{O}(n) \), try to show this.
Problem: Counting Inversions

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

One measure is the number of inversions.

- Assume one ranking is $1, 2, \ldots, n$.
- Let the other be $a_1, a_2, \ldots, a_n$.
- An inversion is a pair $(i, j)$ s.t. $i < j$ but $a_j < a_i$. 

*Similar* 

*Dissimilar*
- Two identical rankings have 0 inversions.
- How many for opposite rankings? \( n \) \(^2 \) \( (1) \)

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 Divide and Conquer algo. will give us \( O(n \log n) \)

Suppose I had a "recursive" algo. that would tell you for \( a_1, \ldots, a_n \) the # of inversions in each half:

\[
\begin{array}{c|c}
\hline
\text{Inv 1} & \text{Inv 2} \\
\hline
a_1, \ldots, a_{n/2} & a_{n/2}+1, \ldots, a_n \\
\hline
\end{array}
\]

What inversions are missed by simply taking \( \text{Inv 1} + \text{Inv 2} \)?

- 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, sorted A = SortAndCount(A)
invB, sorted B = SortAndCount(B)
crossInv, sorted L = MergeAndCount(Sorted A, sorted B)

return invA + invB + crossInv, sorted L

Note: Sorting happens as a byproduct of this algorithm

Half-crossing inversions

A    0    B
\[
\begin{align*}
A_i &> b_j \\
\end{align*}
\]
What if each sublist is sorted?

If we find some \( a_i, b_j \) with \( a_i > b_j \), we can infer many other inversions.

Suppose \( a_i > b_j \), then all items here are also larger than \( b_j \) but we can obtain the number of items in the shaded area in constant time.

```
MergeAndCount(A, B):
    a = b = cross_count = 0, outList = []
    while \( a < |A| \) and \( b < |B| \):
        next = min(A[a], B[b])
        outList.append(next)
        if B[b] = next
            b = b + 1
            cross_count = cross_count + |A| - a
        else
            a = a + 1
    EndWhile
    append the non-empty list to outList
    return crossCount, outList
```
- Note: Merge And Count takes $O(n)$ time

- What is the running time of Sort And Count?

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

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

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

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