CMSC451 - 0301

Ash Dorsey
ash@ash.lgbt

Last updated 2024-10-17.

Contents

Stable Marriage Problem

Find a stable arrangement from 2 equal-sized lists of preferences.

Example

Let's say we have 3 pairs of people.

Number	Men's women preferences	Women's men preferences
1	1,2,3	1,2,3
2	1,2,3	2,3,1
3	1,2,3	3,1,2

Man 3 starts.

He proposes to women 1, and women 1 accepts him.

Number	Men	Women
1		3
2		
3	1	

Man 2 then goes and proposes to 1. Women 1 accepts, and rejects man 3.

Number	Men	Women
1		2
2	1	
3		

Man 3 then tries again, and goes to women 2, and she accepts.

Number	Men	Women
1		2
2	1	3
3	2	

Man 1 then proposes to women 1, and she accepts, rejecting man 2.

Number	Men	Women
1	1	1
2		3
3	2	

Man 2 then proposes to women 2, and she accepts, rejecting man 3.

Number	Men	Women
1	1	1
2	2	2
3		

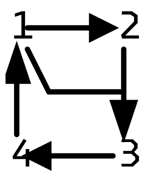
Man 3 then proposes to women 3, and she accepts.

Number	Men	Women
1	1	1
2	2	2
3	3	3

Graph Algorithms

Representing a Graph

Take this graph:



This is usually done in one of two ways:

Adjacency (bit) Matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Left to top.

So, for example, if we have a path from the 6th node to the 3rd node, also denoted $(6, 3)$.

If the graph is undirected, the matrix is symmetric.

The size of the matrix is $\Theta(n^2)$. Specifically, you can store it in n^2 **bits** (bits are small).

Adjacency list

$$[[2, 3], [3], [4], [1, 4]]$$

You may have a hell of pointers as well to connect stuff together to easily delete edges.

It takes $\Theta(m + n)$ space.

If you'd like to be silly, it takes $\Theta((m + n) \lg n)$ space because of the number of bits.

List all edges

Adjacency list

$\Theta(m + n)$ time.

Adjacency matrix

$\Theta(n^2)$ time.

You have to go to every place.

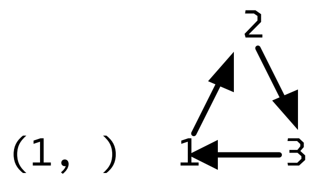
Something to note is that both of these algorithms run in linear time because they are linear with respect to the input size.

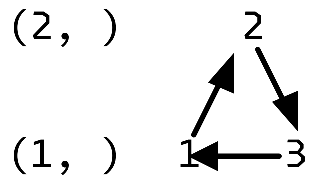
This also means that if you have a matrix multiplication algorithm that takes $\Theta(n^3)$ time, it takes $\Theta\left(N^{\frac{3}{2}}\right)$ time because the size is $N = n^2$.

Depth-First Search

You start at some point and select an arbitrary edge to walk to, dropping breadcrumbs as you go. When you get stuck, return to where you were when you weren't stuck and choose another path until you get to your destination.

You can also add discovery and finish times. Have one counter, and when you number a vertex, you increment the counter. You start at a vertex labeled with discovery 1 and then go to another vertex. Number the vertex's discovery time, and then walk on to another edge. If you explore every edge from a vertex, number the vertex's finish time and back up. For example:



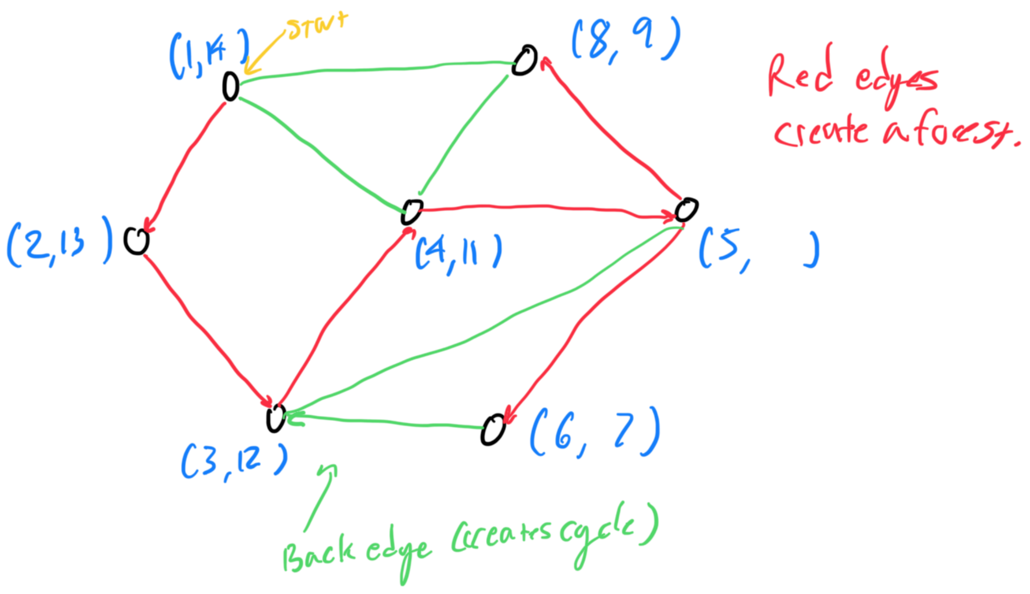


$$\begin{tikzcd}
(2,\ ) & & 2 \arrow[dr] & & \\
(1,\ ) & 1 \arrow[ur] & & 3 \arrow[ll] & (3,\ )\ .
\end{tikzcd}$$

$$\begin{tikzcd}
(2,\ ) & & 2 \arrow[dr] & & \\
(1,\ ) & 1 \arrow[ur] & & 3 \arrow[ll] & (3,\ 4)\ .
\end{tikzcd}$$

$$\begin{tikzcd}
(2,\ 5) & & 2 \arrow[dr] & & \\
(1,\ ) & 1 \arrow[ur] & & 3 \arrow[ll] & (3,\ 4)\ .
\end{tikzcd}$$

$$\begin{tikzcd}
(2,\ 5) & & 2 \arrow[dr] & & \\
(1,\ 6) & 1 \arrow[ur] & & 3 \arrow[ll] & (3,\ 4)\ .
\end{tikzcd}$$



Code of depth-first search

```
for x in vertices(graph):
    x.color = White
```

```
t = 0
for x in vertices(graph):
    if x.color == White:
        depth_first_search(x)

def depth_first_search(x: Node):
    x.color = Gray
    t += 1
    x.discovery_number = t
    for y in adjacent(x):
        if y.color == White:
            depth_first_search(y)
    x.color = Black
    t += 1
    x.finish_time = t
```

This is the generic code for DFS and abstracts finding the vertices and the adjacent nodes.

Adjacency matrix vs adjacency lists

```
for x in vertices(vertices):
```

For an adjacency list or matrix, this part is the same:

```
for x in range(0, n):
```

But this:

```
for y in adjacent(x):
```

Becomes this for the adjacency matrix `A`:

```
for y in range(0, n):
    if A[x, y] != 0:
```

This then leads to $\Theta(n^2)$ complexity because, for every node, you're looping through every other vertex.

For the adjacency list, this becomes:

```
for y in adjacency_list[x]:
```

Which then takes a more complicated amount of time to process.

For this, each node will take the number of edges time. In sum, this will go over each edge twice, leading to $\Theta(n + m)$ time, where n is the number of nodes, and m is the number of edges.

Both of these take linear time with respect to the size of the input.

Connected components

A graph is connected if there is a path between every pair of vertices.

A connected subgraph of a graph is a subgraph of a graph if it is connected.

A connected component is a maximally connected subgraph. You cannot add any more vertices or edges to increase the size while still remaining connected.

Maximum connected subgraph: is as “large” as possible.

Find connected components

```
for x in vertices(graph):
    x.visited = False
```

```
num = 0

for x in vertices(graph):
    if not x.visited:
        num += 1
        depth_first_search(x)

def depth_first_search(x: Node):
    x.visited = True
    x.component = num

    for y in adjacent(x):
        if not y.visited:
            depth_first_search(x)
```

Note that the dot accesses on the nodes could be replaced with external arrays of size n.

This algorithm takes the same time as the generic depth-first search.

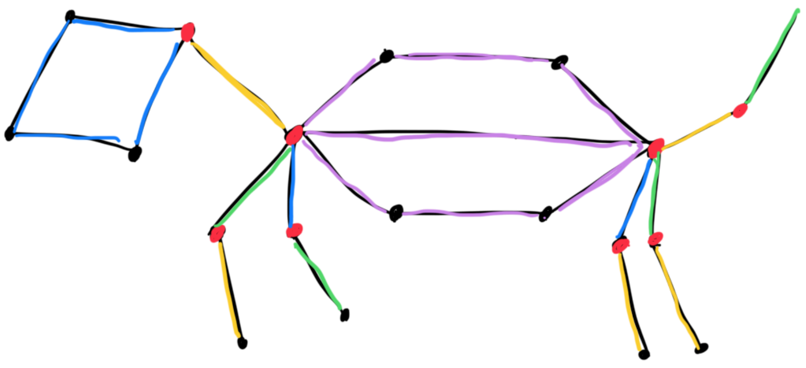
Biconnected
A graph is biconnected if the removal of any one vertex (and its edges) leaves the graph connected.

Articulation point
An articulation point is a vertex whose removal disconnects the graph.

This may be good to avoid if you want redundancy.

The articulation points separate the biconnected components.

Biconnected components
Biconnected components are maximal biconnected subgraphs

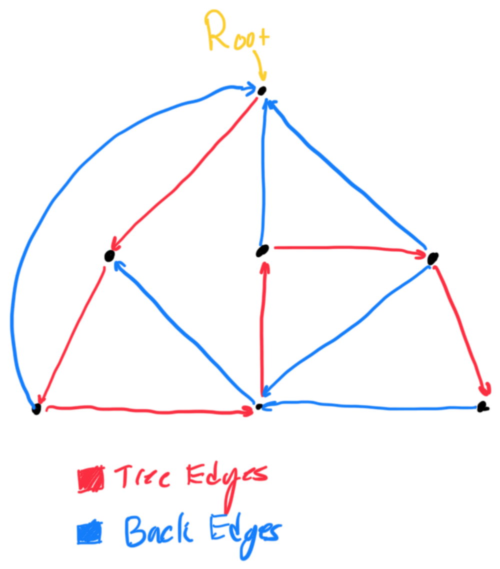


Articulation Points are in red
Different Colors are in different biconnected components

Find the tree edges and back edges

```
for x in vertices(graph):
    x.color = White

back_edges = []
tree_edges = []

for x in vertices(graph):
    if not x.visited:
        depth_first_search(x)

def depth_first_search(x: Node, parent: Node | None = None):
    x.color = Gray

    for y in adjacent(x):
        if y.color == White:
            tree_edges.append((x, y))
            depth_first_search(y, x)
        elif y.color == Gray and y != parent:
            back_edges.append((x, y))

    x.color = Black
```

Example Outcome



Find the biconnected components

1. Find the back edges and tree edges. If there is no back edge pointing to before a point (by the depth of the tree edges) for all child edges, then that point is an articulation point.
2. You only care if your subgraph goes above you, so you can just return the minimum discovery number you discover through back-edges.

Pseudocode

My code:

```
for x in vertices(graph):
    x.discovery_number = None
    x.color = White

articulation_points = []
num = 0

for x in vertices(graph):
    if x.color == White:
        dfs(x)

def dfs(x: Node, parent: Node | None = None) -> int:
    x.color = Gray
    num += 1
    x.discovery_number = num
    minimum = x.discovery_number

    for y in adjacent(x):
        if y.color == White:
            # tree edge
            minimum = min(minimum, dfs(y, x))
        elif y.color == Gray and y != parent:
            # back edge
            minimum = min(minimum, y.discovery_number)

    if minimum < x.discovery_number:
        articulation_points.append(x)

    x.color = Black
    return minimum
```

Kruskal's Code:

```
t = 0
stack = []
connected_components = []

for x in vertices(graph):
    x.discovery_number = 0

for x in vertices(graph):
    if x.discovery_number == 0:
        bicon(x, None)

def bicon(x: Node, parent: Node | None):
    t += 1
    x.discovery_number = t
    x.low_point = t
    for y in adjacent(x):
        if y.discovery_number == 0: # tree edge
            index = len(stack)
            stack.append((x, y))
            bicon(y, x)
            x.low_point = min(x.low_point, y.low_point)
            if y.low_point >= x.discovery_number:
                # x is either an articulation point relative to y
                # or x is the root of the tree.
                # Form a new connected component of all edges on
                # the `stack` up to and including (x, y). Remove these
                # edges from the stack.

                # this can be done in O(1) time with a linked list, but I'm lazy
                connected_components.append(stack[index:])
                while len(stack) != index:
                    stack.pop()
        elif y.discovery_number < x.discovery_number and y != parent: # back edge
            stack.append((x, y))
            x.low_point = min(x.low_point, y.discovery_number)
```

To form the connected components, the idea is that you find the articulation point, and when you return to it again, you found everything it's connected to and so what you found was all of what is in the connected component.

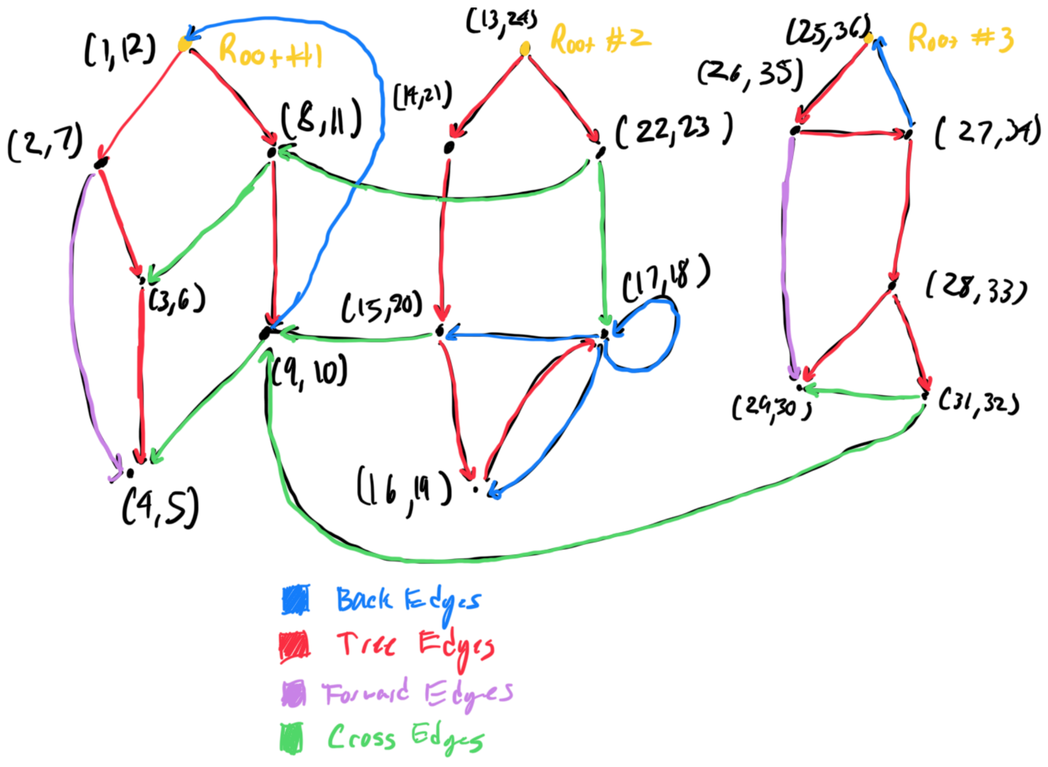
Alternate Algorithm

This uses depth.

```
t = 0
stack = []
connected_components = []

for x in vertices(graph):
    x.depth = 0

for x in vertices(graph):
    if x.depth == 0:
        bicon(x, 0)

def bicon(x: Node, depth: int):
    x.depth = depth + 1
    x.low_point = depth + 1
    for y in adjacent(x):
        if y.depth == 0:
            stack.append((x, y))
            bicon(x, d + 1),
            x.low_point = min(x.low_point, y.low_point)
            if y.low_point >= x.depth:
                # this can be done in O(1) time with a linked list, but I'm lazy
                connected_components.append(stack[index:])
                while len(stack) != index:
                    stack.pop()
        elif y.depth < x.depth:
            stack.append((x,y))
            x.low_point = min(x.low_point, y.low_point)
```

Directed Graphs

DFS



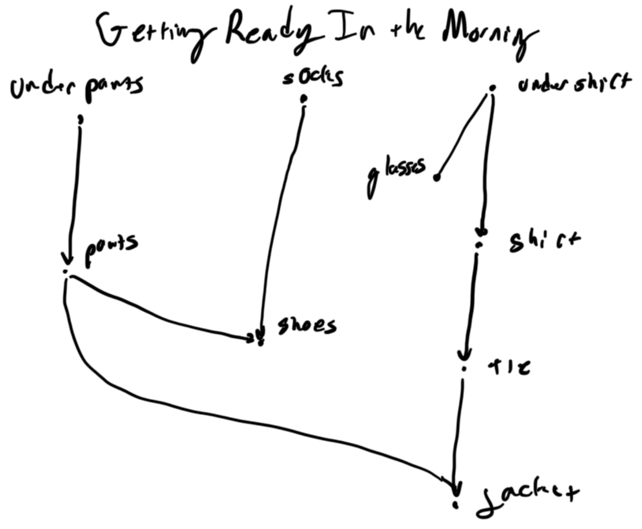
After running DFS, you can divide the remaining edges into forward or cross edges by the discovery and finish numbers.

Let d_1, f_1, d_2, f_2 be the discovery and finish numbers of two nodes that are connected by a remaining edge.

If $(d_1, f_1) \subset (d_2, f_2)$, then the edge from $1 \to 2$ is a forward edge.

Directed Acyclic Graph
A graph that has no cycles and is directed.

Getting Ready in the Morning Example



Iterate through all the edges, and add one to the node that they point to.

Take all the nodes with `0` edges. This is the set of nodes that you can immediately go to.

Then pull off an arbitrary node in the zero set and add that to a list of tasks.

For each of that node's edges, subtract one from the count on the node. If this causes something to become a zero, add it to the zero set.

When the zero set is empty, you're done, and you've constructed a valid topological sorting in the list of tasks.

The zero set can be implemented as an array or anything with $O(1)$ removal and $O(1)$ addition. Additionally, you know the maximum possible size (n) ahead of time.

The zero set could alternatively be constructed in sorted order (with a BTree, perhaps?), allowing one to retrieve a topological sorting additionally sorted by another comparator.

Alternative
You could do this with a depth-first search instead.

Whenever you leave a node, add it to the start of the list. All this requires is a visited array.

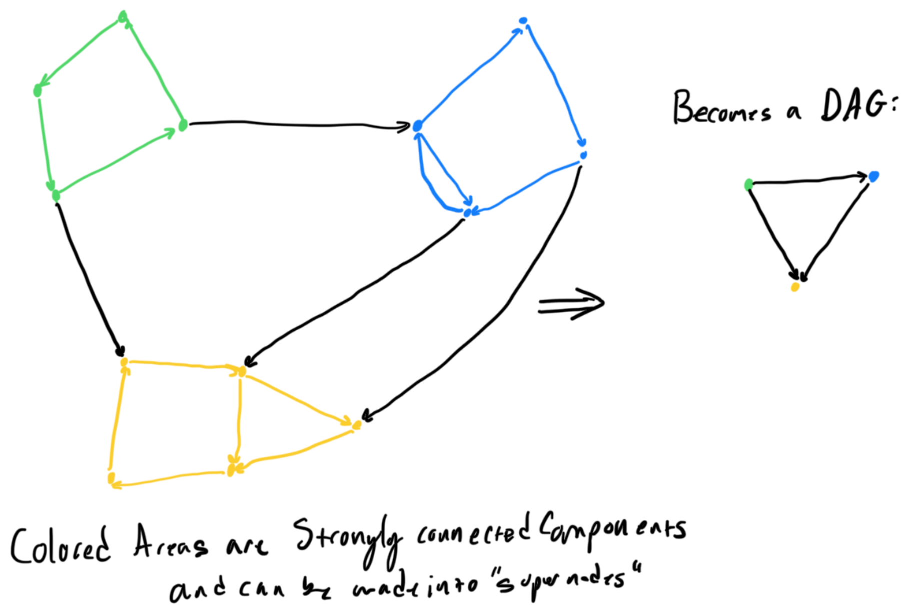
The list requires $O(1)$ prepending or $O(1)$ appending and $O(n)$ reversal.

Note that you know the full size ahead of time.

Alternatively, just reverse the edges first, and the order works out. With an adjacency matrix, this is just switching the meaning of the indices. (Kruskal doesn't think there's something nice with an adjacency list to reverse the edges).

Strongly Connected Graph
If every vertex is reachable from every other vertex, then a graph is strongly connected.

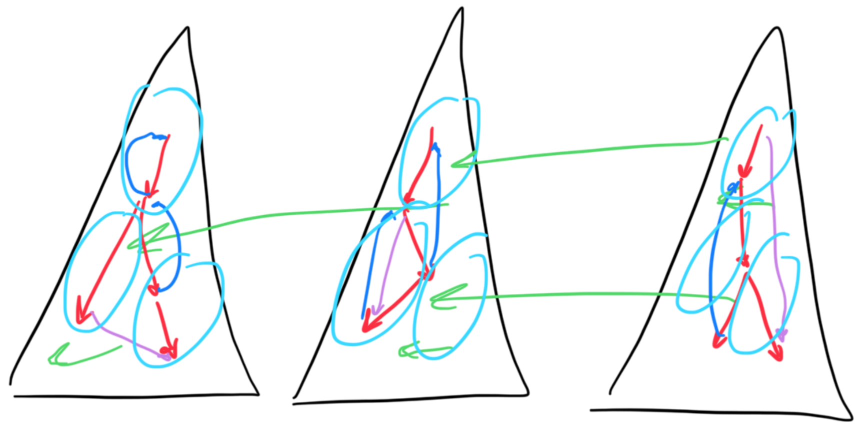
Strongly Connected Component
A maximal subgraph of a graph that is strongly connected.


Becomes a DAG:
Colored Areas are Strongly connected Components
and can be made into "super nodes"

Finding them

Note that if you do a depth-first search, strongly connected components are only within each subtree.

Reverse the edges of the graph, then do a depth-first search in reverse order of finish numbers. When you return from a depth-first search, remove all the nodes you found. They form a strongly connected component.



Breath-First Search

Create a queue. Add a node to the queue.

Repeatedly take a node off the queue and add all nodes attached to that node to the queue. When the queue is empty, you're done.

Order Notation

Proofs are:

1. Boring
2. Unhelpful

Justifying is not generally helpful, particularly if you already have a function.

How relations relate to order notation:

Relations	Order Notation	Vibes	Latex
$=$	Θ	Grows at the same rate	`\Theta`
$\leq$	O	Grows at most as fast	`O`
$\geq$	Ω	Grows at least as fast	`\Omega`
$<$	o	Grow faster than	`o`
$>$	ω	Grow slower than	`\omega`

To remember things, Θ has an equal sign in it and Ω points up.

Examples

- $2n^2 + 7 = \Theta(n^2)$

  If you wrote something like $\Theta(2n^2 + 7)$, that is unhelpful (albeit true), so you will get points off for it.

- $2n^2 + n \lg n = O(n^3)$
- $$\begin{aligned} 2n^2 + 4n + 3 &= 2n^2 + O(n) \\ &= 2n^2 + o(n^2) \\ &\sim 2n^2 \end{aligned}$$

  $\sim$ is used for approximately equal to.

Limit Definitions

$$f(n) = o(g(n)) \leftrightarrow \lim_{h \to \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \leftrightarrow \lim_{h \to \infty} \frac{f(n)}{g(n)} = \infty$$

Note that there should probably be some $\sup$'s thrown around here, but that's almost never a real issue.

Greedy Algorithms

Minimum Spanning Tree

You're given an undirected graph with weights on the edges, and you aim to minimize the total weight of the tree.

Theorem: Adding Minimum Edges Keeps minimum spanning tree

Assume we have a partial minimum spanning tree with subtrees separated by a boundary. Then the cheapest edge crossing the boundary is in a partial minimum spanning tree compatible with the previous one.

Proof
The general idea is that there is a certifier that can tell if an edge is in the minimum spanning tree.

Then, at some time, it says a minimal edge can't be part of the algorithm and constructs the rest of the tree to prove it.

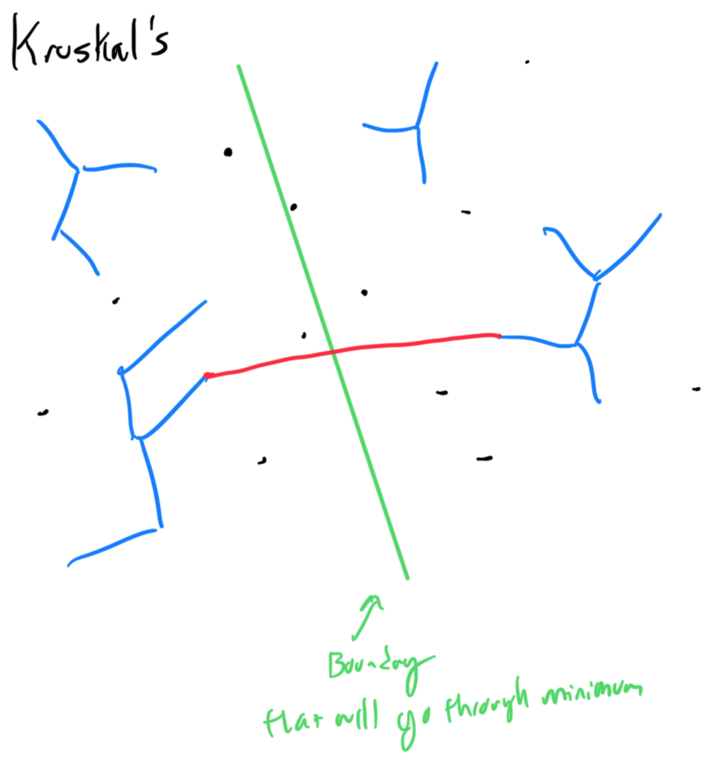
Given the rest of the tree, since it's a spanning tree, it must have constructed edges crossing the boundary to reach all vertices.

Add the minimal edge the certifier didn't approve onto the tree. Since this tree was already spanning, adding this edge creates a cycle. You can then remove any edge across the boundary that the certifier chose in its optimal construction to form a spanning tree again. But because that edge was, at worst, the same cost, it was totally fine to add our minimal edge instead, and the certifier was wrong.

Kruskal's Algorithm
First, sort the edges by weight. Let this be in the list A. Go through the edges. If adding the edge to the tree wouldn't create a cycle, add it.

1. Sort edges:

Proof
Construct the boundary such that the minimum edge would be through the boundary.



Algorithm
Sort edges such that $e_1 \leq e_2 \leq e_3 \leq \cdots \leq e_m$.

Sorting will take $\Theta(m \log m) = \Theta(m \log n)$ time (since $n \leq m \leq n^2$).

For i from 1 to m, put edge e_i onto the tree if it does not create a cycle.

But how do you find these cycles?

By using the union-find problem.

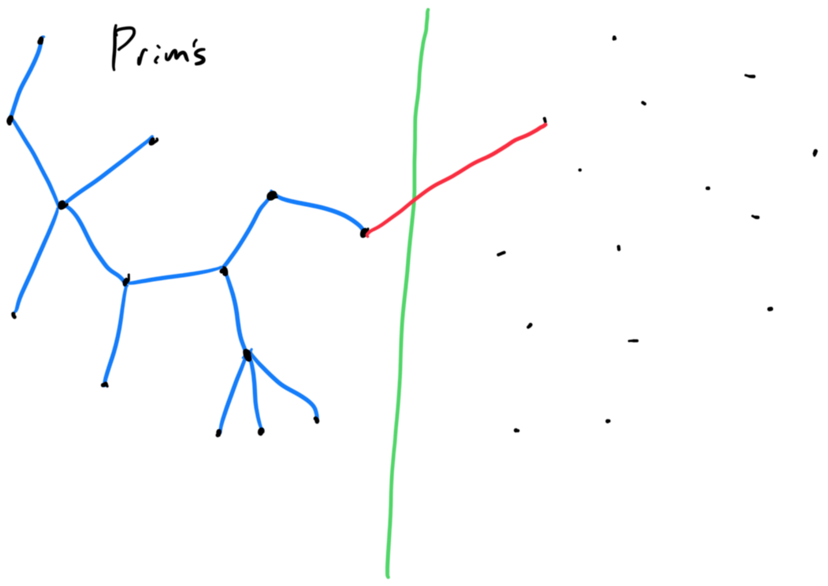
If you're adding another edge, that edge should connect two different trees together. If it connected to the same tree, that would be a cycle.

(Something to note is that these trees can start from any vertex, and they will still be a tree)

Trees of size s will have height at most $\log s$, when you balance the trees. (try to prove this with induction)

Prim's Algorithm
Start with any vertex. Put in the closest (least weight) vertex in the tree.

Proof
Construct the boundary such that the MST built so far is within the boundary, with no other nodes.



Interval Scheduling
Given some intervals, $I_i = [a_i, b_i]$, find the intervals that are nonoverlapping that maximizes= the number of tasks you can do.

Theorem
Scheduling by the earliest finish time gives the optimal solution.

Proof

Minimize Lateness
Minimize your maximal lateness.

Given tasks that are required to be done by d_i and take t_i time, find the arrangments such that $\max(\{f_i - t_i\})$ is as small as possible.

Algorithm
Sort by deadline.

$$d_1 \le d_2 \le \cdots \le d_n$$

Then, work on each task, one at a time, 1 to n.

Basically, do the thing that's due next.

Theorem
Sorting by deadline finds the optimal solution.

Proof
Let a certifier reject some choice k, $d_1 \le d_2 \le d_k$, which would then result in $d_1 \le d_2 \le \cdots \le d_k \le \cdots \le d_n$.

They instead choose something of the form $d_1 \le d_2 \le \cdots \le d_{k-1} \le \cdots \le d_k \le \cdots \le d_b$.

In doing so, the increased the lateness of d_k by $t_a + t_{a+1} + \cdots + t_{k-1}$. This was not an improvement since the time must have increased because the deadlines are further in the future.

Caching

Keep stuff you access often in a cache, and then you can do things faster.

The optimal solution for this is to discard information that is needed farthest in the future. This is called Bélády's algorithm.

It relies on temporal locality, where you use things in the future "soon."

Spatial locality would rely on data that is close together being used together.

Least Recently Used (LRU) cache

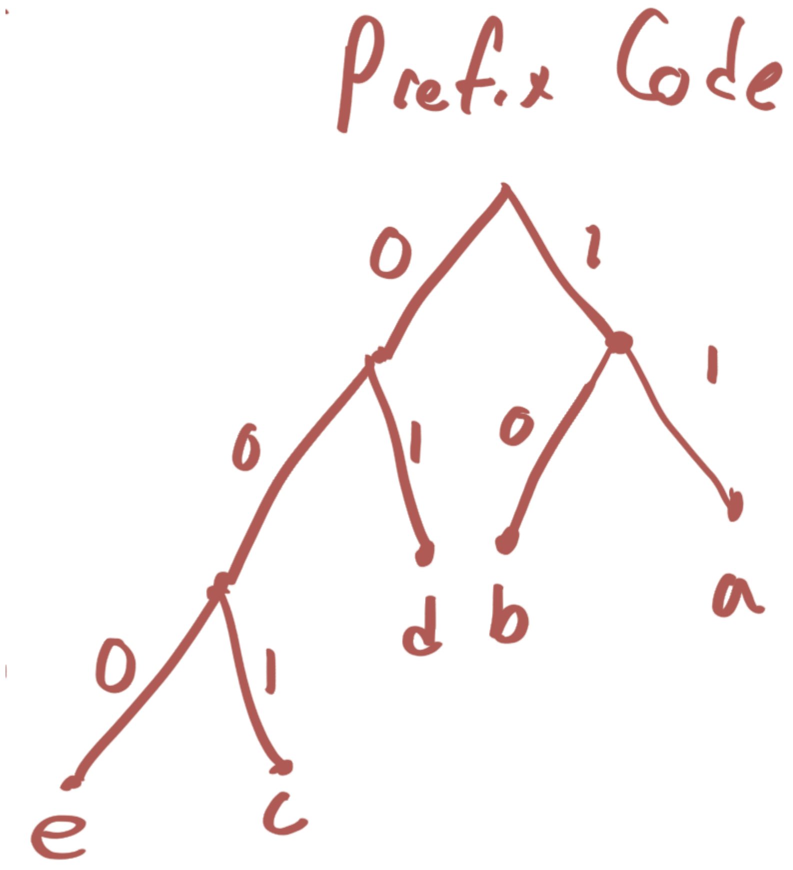
Evict what was the least recently used. Generally a good heuristic. It is what is usually used, or an approximation of it.

Huffman Encoding

Prefix Code

Guarantee that every character has a unique path on a binary tree, and you have a prefix code.

Letter	Frequency	Basic Encoding	Prefix Code	Huffman
a	0.32	`000`	11	`00`
b	0.25	`001`	1`0`	`01`
c	0.20	`010`	`001`	1`0`
d	0.18	`011`	`01`	11`0`
e	0.05	`100`	`000`	111



Furthermore, the expected length of a letter is

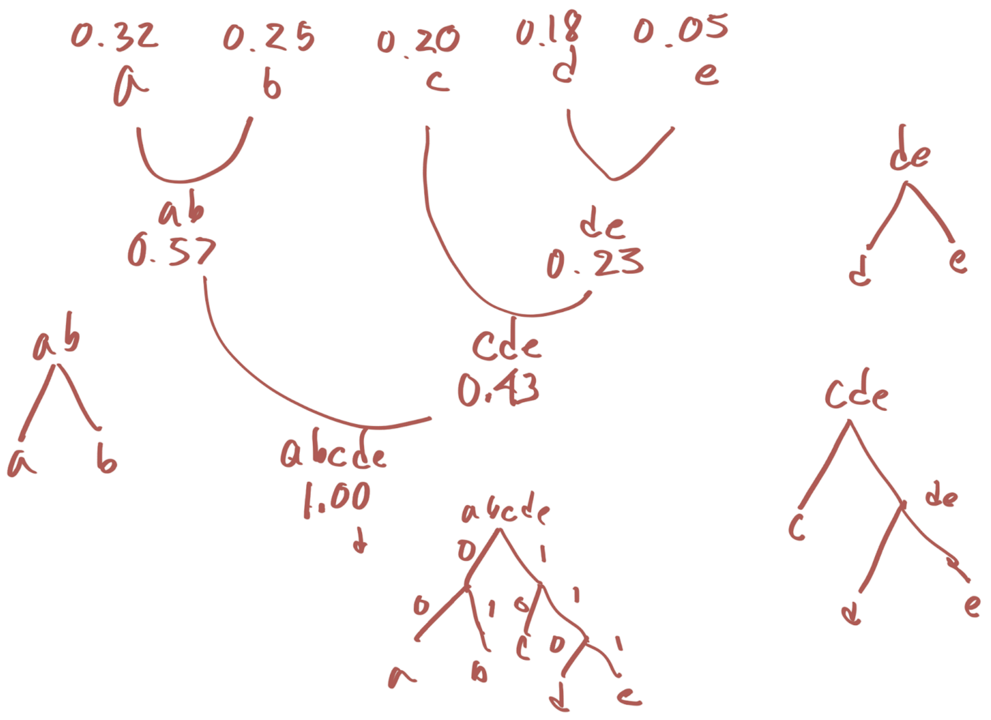
$$E[X] = \sum_{c \in \text{ chars}} p(c) \text{ len}(c)$$

where p is the probability of getting the letter.

For our prefix code, this results in $0.32 \times 2 + 0.25 \times 2 + 0.20 \times 3 + 0.18 \times 2 + 0.05 \times 3 = 2.25$.

The real thing

The idea of Huffman encoding is to build up a binary tree.



For the Huffman code, this turns out to be $0.32 \times 2 + 0.25 \times 2 + 0.20 \times 2 + 0.18 \times 3 + 0.05 \times 3 = 2.23$, which is barely better.

Divide and Conquer

Merge Sort

In lecture online.

Other Examples

- Merge sort
- Maximum contiguous sum (bad solution)
- Quick sort
- Karatsuba

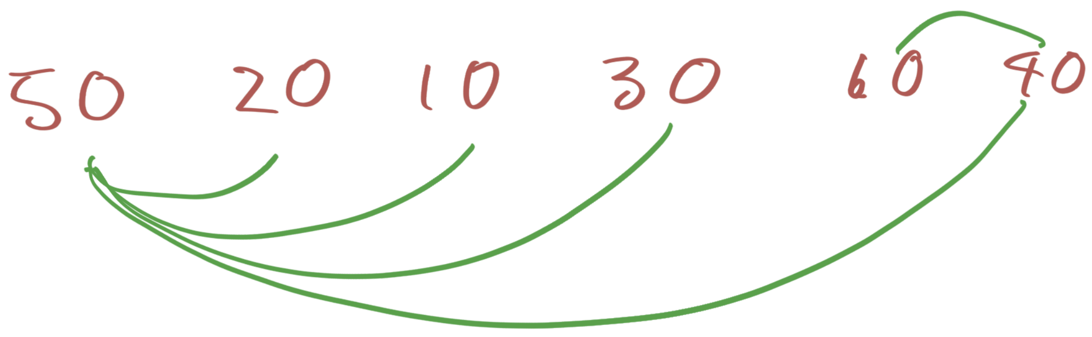
In general, you do something to create subproblems, then solve the subproblems, and then do something to the data you get.

Counting Inversions

$$A = [50, 20, 10, 30, 60, 40]$$

The inversions are numbers that are out of place relative to other numbers.

Inversions



There are 6 inversions in this list.

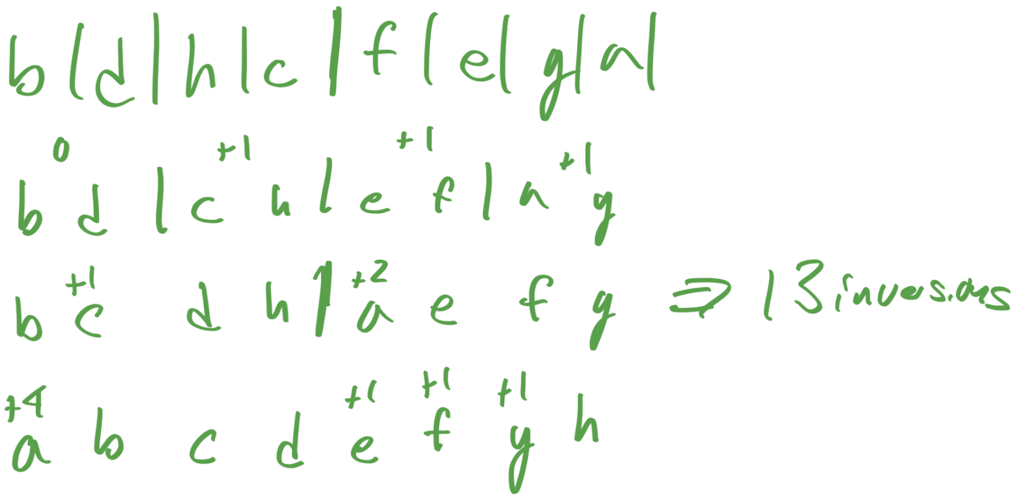
The number of inversions is the same as the number of exchanges that bubble sort would do.

The worst case occurs when the list is reversed, resulting in $\binom{n}{2}$ inversions.

Then the average case is $\frac{1}{2}\binom{n}{2}$ inversions.

To count inversions efficiently, perform merge sort. While merging, compare how far you invert.

For example, if you were merging bd and ch, when you moved the c down, it was inverted with the d.


b | d | h | c | f | e | g | a
0 +1 +1 +1
b d | c h | e f | a g
+1 +2
b c d h | a e f g ⇒ 13 inversions
+4 +1 +1 +1
a b c d e f g h

Closest Pair

You are given n points with distances between them. Find the closest distance.

Bruteforce algorithm:

```
m = 0
for i in range(n):
    for j in range(i + 1, n):
        m = min(m, dist(i, j))
```

What about on a line?

1. Sort points.
2. Find the closest pair next to each other.

This is then a $n \log n$ algorithm, bounded by sorting.

```
m = 0
# sort points
for i in range(n-1):
  m = min(m, dist(i, i + 1))
```

Divide and Conquer a Line

High-level:

1. partition on the median such that half is on the left and half is on the right
2. Find the closest pair on the left and on the right side (recursive)
3. check the rightmost left point and the leftmost right point, and check their distance

Then $T(n) = \Theta(n) + 2T\left(\frac{n}{2}\right) + \Theta(1) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, which implies $\Theta(n \log n)$.

Alternatively:

1. Sort points
2. Recurse on the left and right. And then find the endpoints, and compare them.

On a plane!

Similarly to the divide and conquer solution, cut the solution space in half.

We will decide to split on a line $x = x_0$.

First, sort by x values and by y values (separately).

Choose the middle value, and split it such that half is to the left and half is to the right. You can split by the x_0, and this will then give you two sorted sublists (you may want to pass these down).

You can split the y into sublists by splitting them in two by comparing the x value of a point to x_0.

Solve the problem on the left, resulting in δ_1 and solve the problem on the right, resulting in δ_2.

Let $\delta = \min(\delta_1, \delta_2)$.

Going down the y values on one half within the delta, compare to the next and previous 2 y-values on the other side.

Then, this is:

Before recursing, you do $\Theta(n \lg n)$ work to sort.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n \lg n)$$

And therefore, the total time is $\Theta(n \lg n)$

Strassen's algorithm

For multiplying matrices more quickly than $\Theta(n^3)$.

Let there be two 2×2 matrices A and B.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Traditionally this is:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

For a general matrix, this can be $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, where n is the dimension of the matrix.

In general, if A and B are $n \times n$ matrices, where $n = 2^k$, $k \in \mathbb{Z}^+$, we can multiply A and B by partitioning A and B into 4 submatrices each of size $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$. Interestingly, the above rule for multiplying 2×2 matrices still works where $A_{ij}B_{jk}$ means doing matrix multiplication on two $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrices. This is called block matrix multiplication.

This can be converted into a recursive (divide and conquer) algorithm for matrix multiplication. Since the algorithm performs 8 recursive multiplications and 4 matrix additions on matrices of size $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$, the running time is derived from the recurrence:

$$T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2 \alpha$$

Assuming $T(1) = \mu, T(n) = n^2(n-1)\alpha + n^3\mu = \Theta(n^3)$.

This is the same as just doing it normally, so it doesn't really help.

But there is a clever way to it instead with 7 multiplies and 18 adds.

This results in

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \alpha, T(1) = \mu$$

$$T(n) = \mu n^{\lg 7} + 6\alpha\left(n^{\lg(7)} - n^2\alpha\right)$$

For multplications this becomes $T(n) = n^{\lg 7} \approx n^{2.80735}$

Carry look ahead addition

If you have two 0s added, you know there will not be a carry. If you know that there are two 1s added, you know there will be a carry.

You can use this to do work ahead of time.

Log Transform

If you have a lot of multiplies, you can take the logarithm since it converts multiples to additions.

Fast Fourier Transform

$$\begin{aligned}
A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\
B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1} \\
C(x) &= A(x) B(x) \\
&= c_0 + c_1 x + c_2 x^2 + \cdots + c_{2n-2} x^{2n-2}
\end{aligned}$$

Then, evaluate A, B at $2n$ values.

$$\begin{aligned}
& x_1, x_2, ..., x_{2n} \\
\Rightarrow\ & A(x_1), A(x_2), ..., A(x_{2n}) \\
\Rightarrow\ & B(x_1), B(x_2), ..., B(x_{2n})
\end{aligned}$$

Using those values, compute C at $2n$ values:

$$\begin{aligned}
C(x_1) &= A(x_1) B(x_1) \\
C(x_2) &= A(x_2) B(x_2) \\
&\vdots \\
C(x_{2n}) &= A(x_{2n}) B(x_{2n})
\end{aligned}$$

You can then reconstruct $C(x)$ using those values.

$$\begin{aligned}
A_{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{\frac{n-2}{2}} \\
A_{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{\frac{n-2}{2}}
\end{aligned}$$

Then notice:

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$$

Using this, we can evaluate polynomials via divide and conquer:

```python
def evaluate(A, x):
    if A.degree == 0:
        return A[0]
    x_squared = x**2
    a_even = evaluate(A.even, x_squared)
    a_odd = evaluate(A.odd, x_squared)
    return a_even + x * a_even
```

On its own, this takes $T(n) = 2T\left(\frac{n}{2}\right) + 2$ time, and $T(1) = 0$, counting multiplies.

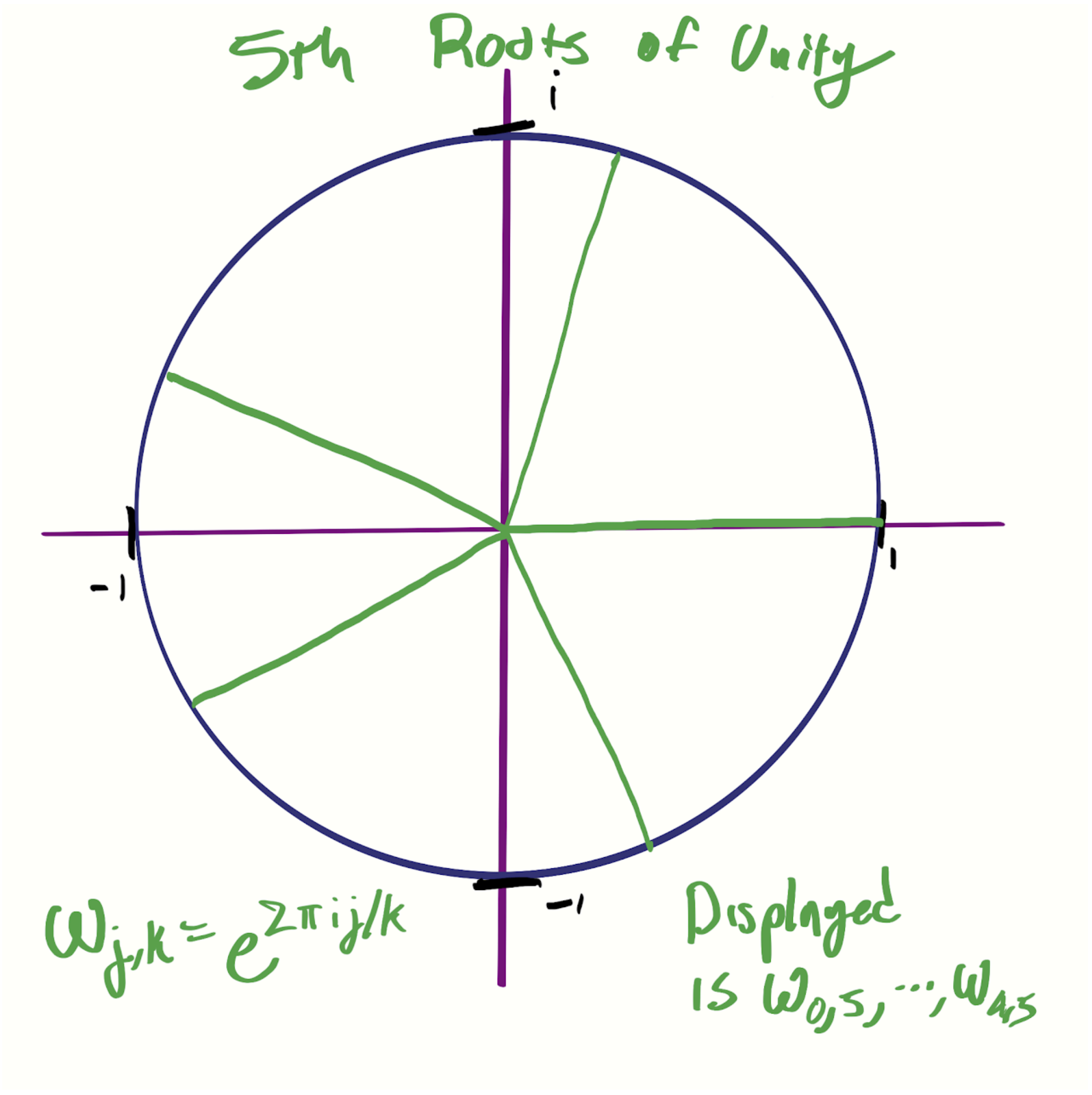
Then, $T(n) = 2(n - 1) = \Theta(n)$, which is the same complexity as the normal way to evaluate polynomials.

This then makes the $2n$ evaluations take $\Theta(n^2)$ time.

Finding the $c_0, ..., c_{2n}$ values takes $\Theta(n)$ time.

But we can select our roots better!

$$\omega_{j,k} = e^{\tau i j / k}$$


5th Roots of Unity
i
-1
1
-i
$\omega_{j,k} = e^{2\pi i j/k}$
Displayed is $\omega_{0,5}, \dots, \omega_{4,5}$

Evaluate $A(x)$ at the $2n$ roots of unity.

Assume n is a power of two.

$$\begin{aligned} A(\omega_{j,2n}) &= A_{\text{even}}(\omega_{j,2n}^2) + \omega_{j,2n} A_{\text{odd}}(\omega_{j,2n}^2) \\ &= A_{\text{even}}(\omega_{j,n}) + \omega_{j,2n} A_{\text{odd}}(\omega_{j,n}) \end{aligned}$$

Done with the example $A(x) = 1 + 4x - 3x^2 + 2x^3$, and do one split:

x	$A(x)$	x^2	$A_{\text{even}}(x^2)$	$A_{\text{odd}}(x^2)$	$xA_{\text{odd}}(x^2)$	$A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$
1	4	1	-2	6	6	4
-1	-8	1	-2	6	-6	-8
i	$4+2i$	-1	4	2	$2i$	$4+2i$
$-i$	$4-2i$	-1	4	2	$-2i$	$4-2i$

See that there is only 1 and –1. This is very convenient; we have reduced the solution space.

To produce the values for all the roots of unity, this will take:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Longrightarrow T(n) = \Theta(n \lg n)$$

Which is a very nice improvement on the previous $\Theta(n^2)$.

We want

$$C(x) = \sum_{s=0}^{2n-1} c_s x^s$$

But define

$$D(x) = \sum_{s=0}^{2n-1} d_s x^s$$

Where $d_s = C(\omega_{s,2n})$, which we know from evaluating before.

Then:

$$\begin{aligned}
D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} d_s (\omega_{j,2n})^s \\
&= \sum_{s=0}^{2n-1} C(\omega_{s,2n})(\omega_{j,2n})^s \\
&= \sum_{s=0}^{2n-1} \left[\sum_{t=0}^{2n-1} \left(c_t \cdot (\omega_{s,2n})^t\right)\right] (\omega_{j,2n})^s \\
&= \sum_{t=0}^{2n-1} \sum_{s=0}^{2n-1} c_t \cdot (\omega_{s,2n})^t (\omega_{j,2n})^s \\
&= \sum_{t=0}^{2n-1} c_t \sum_{s=0}^{2n-1} \left(e^{\tau is/(2n)}\right)^t \left(e^{\tau ij/(2n)}\right)^s \\
&= \sum_{t=0}^{2n-1} c_t \sum_{s=0}^{2n-1} e^{\tau its/(2n)} e^{\tau isj/(2n)} \\
&= \sum_{t=0}^{2n-1} c_t \sum_{s=0}^{2n-1} e^{\tau its/(2n) + \tau isj/(2n)} \\
&= \sum_{t=0}^{2n-1} c_t \sum_{s=0}^{2n-1} e^{\tau is(t+j)/(2n)} \\
&= \sum_{t=0}^{2n-1} c_t \sum_{s=0}^{2n-1} (\omega_{t+j,2n})^s
\end{aligned}$$

Let $\omega_{k,2n} \neq 1$, and $k = t + j$:

$$\sum_{s=0}^{2n-1} (\omega_{k,2n})^s = \frac{1 - (\omega_{k,2n})^{2n-1+1}}{1 - \omega_{k,2n}} = \frac{1 - (\omega_{k,2n})^{2n}}{1 - \omega_{k,2n}} = \frac{1 - \left(e^{\tau ik/(2n)}\right)^{2n}}{1 - \omega_{k,2n}} = \frac{1-1}{1 - \omega_{k,2n}} = \frac{0}{1 - \omega_{k,2n}} = 0$$

However, when $\omega_{k,2n} = 1$,

$$\sum_{s=0}^{2n-1} 1 = 2n$$

Let $t + j = 2n$ since everything else becomes zero. Then $t = 2n - j$:

$$\begin{aligned} D(\omega_{j,2n}) &= c_{2n-j} \sum_{s=0}^{2n-1} (\omega_{2n,2n})^s \\ &= c_{2n-j} 2n \\ &= 2n c_{2n-j} \end{aligned}$$

And therefore, $D(\omega_{k,2n}) = 2n c_{2n-j}$, and finally

$$c_{2n-j} = \frac{D(\omega_{j,2n})}{2n}$$

Using this property, you can find c_m by varying j.

You can evaluate $D(\omega_j, 2n)$ for all j in $\Theta(n \lg n)$ time. Dividing takes $\Theta(1)$ time for each, and so takes $\Theta(n)$ for all values of j. In sum, this takes $\Theta(n \lg n)$ time.

Adding this on to the other program to evaluate the $C(\omega)$'s, this takes in total $\Theta(n \lg n)$ time to generate the polynomial $C(X)$.

Dynamic Programming

Fibonacci

Fibonacci, to find the `n`th number, using variables:

```
F_n_minus_1 = 1
F_n = 1

for i in range(3, n + 1):
    F_n_minus_2 = F_n_minus_1
    F_n_minus_1 = F_n
    F_n = F_n_minus_1 + F_n_minus_2
print(F_n)
```

You can also just use arrays:

```
F = [None] * n
F[0] = 1
F[1] = 1
for i in range(2, n + 1):
    F[i] = F[i-1] + F[i - 2]
print(F[n])
```

Or do it recursively:

```
def f(n):
    if n == 1 or n == 2:
        return 1

    return f(n - 1) + f(n - 2)
```

Comically terrible complexity, but oh well.

If you memoize the results, it works fine.

```python
fib = [None] * (n + 1)

def f(n):
    nonlocal fib
    if fib[n] != None:
        return fib[n]

    if n == 1 or n == 2:
        return 1

    fib[n] = f(n - 1) + f(n - 2)
    return fib[n]
```

Alternatively, in python:

```python
from functools import cache

@cache
def f(n):
    if n == 1 or n == 2:
        return 1
    return f(n - 1) + f(n - 2)
```