Algorithms Notes

Simon Xiang

Lecture notes for the Spring 2022 section of Algorithms and Complexity (CS 331) at UT Austin, taught by Dr. Price. These notes were taken live in class (and so they may contain many errors). Source files: https://git.simonxiang.xyz/math_notes/files.html

Contents

1 Algorithms and complexity

Complexity is the broad classification of different problem types (problems that can be solved by computers). We will spend the first three months of class in polynomial time P, specifically the time within P. For example, we have insertion sort (complexity $O(n^2)$) vs merge sort (complexity $O(n\log n)$).

Today we look at a way to reuse our computation to do things efficiently, called dynamic programming. Our first example is a classical algorithm we have been using our entire lives.

1.1 Multiplication

I'm not writing this out but we multiplied 331×388 using the algorithm we learned in elementary school. Then we added two large numbers which was much easier. So adding two n-digit numbers is $O(n)$ while multiplying results in n^2 digits, so adding them up means that multiplication is $O(n^2)$.

The reason why we don't care about constants in algorithms is that different operations take different amounts of time for different computers. A human would have a larger constant than a computer, the limiter may be writing down the numbers or low RAM. Who knows. The point is, relative time varies so we use big-O notation, which doesn't care.

We have a better algorithm for multiplication than schoolbook multiplication, called **Karatsuba multiplication**. Consider the numbers $A = 5124758048$ and $B = 617586337$. Split A and B into two halves as below:

$$\underbrace{51247}_{A_1}\underbrace{58048}_{A_2} \times \underbrace{61725}_{B_1}\underbrace{86337}_{B_2}$$

Then $A\cdot B = (10^{n/2}A_1+A_2)\cdot(10^{n/2}B_1+B_2) = 10^n\cdot A_1B_1+10^{n/2}(A_1B_2+A_2B_1)+A_2B_2$. We have n-digit multiplication equal to four instances of $n/2$ digit multiplication plus three additions, so $T(n) = 4T(n/2)+O(n)$ (where T represents n-digit multiplication). We solve this using the **Master theorem**, which says $T(n) = aT\left(\frac{n}{b}\right) - f(n)$. Here $a = 4, b = 2, f(n) = O(n)$, so $c_{\text{crit}} = \log_b a = 2$. Case 1 implies that $T(n) = \Theta(n^{c_{\text{crit}}}) = \Theta(n^2)$.

The total work is the number of nodes times the work of each notes. As each level we multiply by an order of 4. Each node is working at an order of $\frac{n}{2^k}$, so the total work is $4^k\cdot\frac{n}{2^k} = 2^k n$. In short, $T(n) = 4T(n/2)+O(n) = O(n^2)$. Note that

$$(A_1+A_2)(B_1+B_2) = A_1B_1+(A_1B_2+A_2B_1)+A_2B_2 \implies A_1B_2+A_2B_1 = (A_1+A_2)(B_1+B_2)-A_1B_1-A_2B_2.$$

So instead of four $\frac{n}{2}$ terms we have three. Then $T(n) = 3T(\frac{n}{2})+O(n)$, running it down means that the total work is now $\left(\frac{3}{2}\right)^k n$. So our time complexity is now $\left(\frac{3}{2}\right)^{\log_2 n} = n^{\log_2 \frac{3}{2}}$, which implies $T(n) = 3T\left(\frac{n}{2}\right)+O(n) = n^{\log_2 3} \approx n^{1.585}$ (also follows by Master theorem).

2 Recursion

Recursion is a very common idea used to write correct programs. It means we solve the problem by "recursively" solving smaller versions of the *same* problem. We have already seen this in various examples;

- Multiplication, as in the last class.
- Merge sort- we want to sort an array X. We sort the first half, then sort the second half, then merge. This is recursive because we have a problem, and we call the same function on a smaller instance. I
- Towers of Hanoi- see the book.

2.1 The n Queens problem

Consider the n queens problem. We have an n by n chessboard; can we place queens on the board such that they don't attack each other? The first question is, how many queens? Note that this is capped at n, since we only have n rows. When placing queens and it fails, we can go back and try other solutions, this leads to *backtracking*. Forgetting runtime, how would we implement this problem?

The strategy is as follows:

(1) Recast each answer as a *series of single choices*; (Row 1 Q, Row 2 Q, Row 3 Q, ...)

(2) Each successive choice is its own recursive call,

(3) If stuck, return.

Returning to the n queens problem.

```
    def collide(a, b):
    # a= (row, col), b= (row, col)

    return a[0] == b[0] or a[1] == b[1] or abs (a[0] - b[0]) == abs(a[1] - b[1])

def queens4():
    for q1 in range(4):
        for q2 in range(4):
            #Does placing a queen at (2, q2) collide with (1, q1)?
            if collide((1,q1), (2, q2)):
                continue
            for q3 in range(4):
                # Does placing a queen at (3, q3) collide with EITHER (1,q1) or (2,q2
                                                                    )?

                if collide((1, q1), (3, q3)) or collide((2, q2), (3, q3)):
                    continue
                for q4 in range(4):
                    if (collide((1, q1), (4, q4)) or collide((2, q2), (4, q4))
                        or collide((3, q3), (4, q4))):
                        continue
                    print(q1, q2, q3, q4)

queens4()
```

todo:fix code Okay, we've solved the four queens problem. But if we were to do this for eight queens, we would need eight nested loops. This is a huge pain. If we want to do this with general n, doing it iteratively is not a good solution at all.

```
    ok
```

todo:dicussion on just returning the first one, exponential time. This is the general strategy of recursion with backtracking, and is pretty generic.

2.2 Game Tree Evaluation

The question is "who wins at `[game]`"? Could be chess, checkers, tic-tac-toe, etc. We have a game state, say for white, and white has many different moves. For each board state, we have many different moves, and so on. todo:figure

Most of these alternate, but in principle but they don't have to. At the end we reach some position, say checkmate (for black). In this case black wins. How do we figure out who wins? (Suppose there are no draws). Say we are in the position `(player, state)`.

- If the state is now checkmate, we know who wins.
- If we are not in checkmate, we have some moves we can do, each leading to another state.
- Else, `winner(player, state)` wins if there exists any move to `state'` where `winner(other player, state')=player`.
- The other player wins otherwise.

We can apply this to draws with some sort of maximum or minimum score. We write this out as a recursion where for each player we try all possibilities and evaluate. If any of them lead to checkmate, that's a good move and we do it.

So writing a program to evaluate the winner is easy (computable), but finding a winner is hard.

2.3 Subset Sum

In this problem, we are given a set of positive integers X, say $[1, 4, 5]$, and a target T (say 9). The question is this: Find a subset of X that sums to T. This is a natural building block in a number of problems. How do we implement this? (Forget time and all of that).

todo:code

Listing out all subsets is hard.

```
    def subsetsum(X, T):
    """Find a subset of X that sums to T.

    Inputs:
        X: a list of positive integers, e.g. [1, 4, 5]
        T: a target positive integer, e.g. 9

    Outputs:
        None or a list that sums to T (e.g., [4, 5])
    """

    #base cases
    if T == 0:
        return []
    if not X or T < 0:
        return None

    #recursion
    take = subsetsum(X[1:], T - X[0])
    if take is not None:
        return [X[0]] + take
    skip = subsetsum(X[1:], T)
    return skip

print(subsetsum([1, 4, 5], 9))

import random
vals = [random.randint(10000, 20000) for _ in range(5)]
T = sum(v * random.randint(0, 1) for v in vals)
print(f"Instance: {vals} {T}")
print((subsetsum(vals, T)))
```

This code finds *a* solution. What if we want to find a *small* solution? todo:code This solution would be cleaner if we didn't have to rewrite our huge array each time. We do this by rewriting with an index.

```
    def subsetsum(X, T, i=0):
    """Find a subset of X that sums to T.

    Inputs:
        X: a list of positive integers, e.g. [1, 4, 5]
        T: a target positive integer, e.g. 9
        i: index we're looking at [so X[i:] is of interest]

    Outputs:
        None or a list that sums to T (e.g., [4, 5])
    """

    #base cases
    if T == 0:
        return []
    if len(X) <= i or T < 0:
        return None

    #recursion
    take = subsetsum(X[1:], T - X[0])
    if take is not None:
        return [X[0]] + take
    skip = subsetsum(X[1:], T)
    return skip

print(subsetsum([1, 4, 5], 9))

import random
vals = [random.randint(10000, 20000) for _ in range(5)]
T = sum(v * random.randint(0, 1) for v in vals)
print(f"Instance: {vals} {T}")

print((subsetsum(vals, T)))
```

Here's the idea; how many different inputs does `subsetsum` take? todo:2 to the n leaves? internal nodes are the same value?

3 Fibonacci numbers

Fibonacci numbers are a classic example of one of the first algorithms studied in the west. This is a recurrence relation defined by $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.

```
   def f(n):
    if n <= 1: return n
    return f(n-1) + f(n-2)
```

We write out a *giant* recursion tree just to compute $f(5)$, with 15 nodes, only increasing with each step. This number is getting very big: how long does this take? The number of steps $T(n) = T(n-1) + T(n-2) + 2$ has the same recurrence as F_n plus an additional factor 1, so $T(1) = 1, T(0) = 0$, and $T(n) \geq T(n-1) + T(n-2)$ which implies $T(n) \geq F_n$. How fast does F_n grow? We estimate it grows exponentially:

$$F_n \leq 2F_{n-1} \leq 4F_{n-2} \leq \cdots \leq 2^k F_{n-k} \leq \cdots \leq 2^{n-1}.$$

This is an upper bound. Is there a lower bound?

$$F_n \geq 2F_{n-2} \geq 4_{n-4} \geq 8F_{n-6} \geq \cdots \geq 2^k F_{n-2k} \geq \cdots \geq 2^{\frac{n}{2}-1} \approx 1.4^n.$$

So the Fibonacci numbers grow in between 1.4^n and 2^n. Therefore this is an exponential time algorithm, since the recursion tree is exponentially large. Our current algorithm is kind of dumb. We are doing lots of extra work.

```
memo = {}
def f2(n):
    if n in memo: return memo[n]
    if n <= 1: return n
    ans = f2(n-1) + f2(n-2)
    memo[n] = ans
    return ans
```

This is *much* faster because we only need to compute each value of n once, so it takes one addition to compute. We can rewrite this recursion tree in a different way where we reuse the nodes. There is a directed acyclic graph on the previous values; this computation graph has n vertices and $2n$ edges. The recursion time is the number of possible paths from n to the base case. However, the "memo-ized" recursion is the number of vertices times the time per vertex, which is one addition, giving us $O(n)$ time.

3.1 Dynamic programming

This is the idea of **dynamic programming**. We will see many more examples of this later. In dynamic programming, we solve recursive problems quickly by storing the answers to the subproblems. There are different ways we can do this:

- "Top-down": Recursion with memo-ization
- "Bottom-up DP": Compute the answers bottom to top iteratively

```
    def fibdp(n):
    fibs = [0, 1]
    for i in range(2, n+1):
        fibs.append(fibs[i-1] + fibs[i-2])
    return fibs[n]
```

This gets around the (non-theoretical) issue of not being able to call recursion too many times. We can also do what is called a "sliding window" DP, where we only store the last couple of recursions and discard the rest, helping with space issues.

```
    def fibwindow(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

This algorithm is $O(1)$ space and $O(n)$ time, while the previous algorithm is both $O(n)$ space and time. Can we do better than linear time?

3.2 Matrix exponentiation method

Each step the sliding window algorithm can be thought of as applying a matrix $\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]$ to a vector $\left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]$. In essence, a, b starts at $\left[\begin{smallmatrix}F_0\\F_1\end{smallmatrix}\right]$ and ends at

$$\begin{bmatrix}F_n\\F_{n+1}\end{bmatrix}=\begin{bmatrix}0&1\\1&1\end{bmatrix}^n\cdot\begin{bmatrix}0\\1\end{bmatrix},\quad \begin{bmatrix}0&1\\1&1\end{bmatrix}^n=\begin{bmatrix}F_{n-1}&F_n\\F_n&F_{n+1}\end{bmatrix}.$$

Given A, how quickly can we compute A^n? We split the squares; $A^{13}=A^8A^4A$, and compute each by repeated squaring. So this results in $\log n$ matrix multiplies. Therefore we can compute F_n with $O(\log n)$ arithmetic operations. Numbers get really big; we have $1.4^n \le F_n \le 2^n$, $F_n = 2^{\Theta(n)}$ which implies $\log F_n = \Theta(n)$. So writing down F_n takes $\Theta(n)$ bits.

something, assume arithmetic is constant only if numbers are polynomial large, wordram model?

4 Dynamic programming

So far we've been doing recursion, which entails solving a problem by formulating it as a series of choices. For example, we pick one choice and recuse on the rest. For the n queens, we decide where to do on the first row, then decide where to go on the rest. The benefits is that this is easy to find an accurate solution for many problems. The problem is that this is usually exponential time.

Dynamic programming can be thought of as "smart recursion", or "recursion without repetition". Last times example was subsetsum. This tree in principle could have 2^n leaves, but we store answers to intermediate problems. We think of recursion graphs as a DAG.

4.1 Interval scheduling

Say that you're an airbnb host and many people are requesting different times to stay at your vacation home, and different people will pay you to stay at different times. We are given a set I of **intervals** consisting of a start time, finish time, and value (s_i, f_i, w_i) and we want to find a disjoint subset of maximum weight. In other words, we want to find $S \subseteq I$ that doesn't overlap $(s_i, f_i) = \emptyset$) maximizing weight, or

$$\sum_{i\in S} w_i \ge \sum_{i\in G} w_i$$

for any $G \subseteq I$. How would we write a scheduling function $\text{Sched}(I)$? For now, our goal is the weight of S. Pick any interval i; then return the maximum $(\text{Sched}(I \setminus i), w_i + \text{Sched}(I \setminus \{i \text{ or any } j \text{ overlaps } i\})$. How do we make this fast? Pick $i = 1$ each time. All recursive calls only ever consider suffixes of I, so the number of different inputs is $n+1$.

4.2 Longest increasing subsequence

Given $A_1, A_2, \cdots, A_n$, we want to find $s_1 < s_2 < \cdots < s_k \in [n]$ with $A_{s_1} \le A_{s_2} \le \cdots \le A_{s_k}$ of maximum weight. How do we do this? We claim this corresponds to the maximum weight path of a graph. We build a graph as following: let $f(i) :=$ LIS strating with (i, A_i).

5 Practice problems

Problem 1. *n local businesses free products, certain times, business i will be giving away free times only at time T_i. for each pair (i, j), the amount of time $t_{ij} > 0$ to walk from i to j (satisfying the triangle inequality), design an $O(n^2)$*

DP algorithm to compute maximum number of businesses,

Solution. how to do this? First solve this as a recursion. Cast the problem as a **series of choices**; the choices are a sequence of merchants/stalls visited. Possible if $T_i - T_j \geq t_{ij}$ (can make it in time).

```
def solve(choices):
    for each next choice i
    if valid (T_i - T_choices) geq t_{i,choices}
        solve(choice, t[i])
```

Another option is this; "do I show up to stall i"? How good is a set of choices? We need to know where we end up (last stall visited, hard to compare at different stalls) and the number of visits. Define $f(i) :=$ maximum number of stalls you can visit before reaching stall i's giveaway. The answer is $\max_j f(j) + 1$, such that $T_i - T_i \geq T_{ij}$ for every i. The reason being we came from somewhere, being j. Therefore, this is the **recurrence**:

```
f(i)=max_j f(j)+1, T_i-T_j\geq t_{ij}
```

This is the **base case**,

```
f(0):=0
```

where business 0 is located at ??, T_0. The **recurrence subproblem** is

```
f(i):= max number of stalls you can visit
before ending up at stall i's giveaway
```

which we solved with the first algorithm. This has space $O(n)$, time $O(n^2)$. ∎

Solution. latex. choices = which word to break after? $f(i)$ = min sum of sequences of gaps before a linebreak after i. two solutions end at the same word; what is the cost? subproblem; given word best subproblem.

If we end at a given word, given that we line break after word, all that matters is the sum of squares before. Write down base case, recurrence, answer. Basecase: $f(0) = 0$ no words. Answer: $\min_i f(i)$, wordds $i+1, \cdots, n$ fit in the line. Recursive subproblem: $f(i) = \min_{j<i} f(j)+$ cost of $i+1, \cdots, j$ line. ∎

6 Knapsack

A classic example of DP is robbing a store. You have a knapsack, and you can only carry so much weight. The store has n items, each with value v_i and weight w_i. You have capacity C. What is the max value subset of items to carry with total weight $\leq C$? We could also write this as $\max_{S\subseteq[n], \sum_{i\in S} w_i \leq c} v_i$. How do we write a recursive solution?

```
knapsack(items, C)
    w,r = items[-1]
    return max(knapsack(items[:-1],C),
    knapsack(items[:-1],C-w)+v)

#base cases
    if C < 0: return -infinity
    if len(items)=0: return 0
```

What does DP say we are doing? We should make this fast by memoizing. Then the time depends on the number of vertices of our directed acyclic graph (DAG), representing todo:?? two dimensional table

todo:sliding window?

7 Greedy Algorithms

Consider interval scheduling. What do we do for greed?

(1) Earliest finish=latest start

(2) Duration (shortest)

(3) Earliest start (first come first serve)

(4) Smallest number of intervals it intersects

(2) doesn't work (small interval intersecting two), (3) doesn't work (long interval at the beginning). It turns out (1) works and (4) doesn't. For (4), duplicate intervals mess it up, you trick the algorithm (pushing it toward the wrong choice) by duplicating intervals that intersect bad choices. So early finish works in all cases, with runtime $O(n \log n)$.[1] The point is that it's easy to convince yourself that algorithms work (particularly with greedy algorithms), so we need proofs of correctness.

So how do we go about proving correctness?

Proof. We use induction on n. The base case is $n = 0$, which returns $[]$, which is correct. Let $n \geq 1$, $|I| = n$. By the inductive hypothesis, suppose `GreedySchedule`(I') is correct for every I, $|I'| \leq n-1$. For any t, define $I_t := \{(s,f) \leq I \mid s \geq t\}$. In (s_1, f_1)=the first to finish in I, `GreedySchedule`$(I) = [(s_1,f_1)]+$`GreedySchedule`(I_{f_1}). Then $|I_{f_1}| \leq n-1$ implies `GreedySchedule`(I_{f_1}) is correct in I_{f_1}.

Let S^* be a true solution for I, then define $S^* = (s_1^*, f_1^*), (s_2^*, f_2^*), \cdots, (s_{k^*}^*, f_{k^*}^*)$, where $s_1^* < f_1^* \leq s_2^* < f_2^* \leq \cdots \leq s_{k^*}^* < f_{k^*}^*$. Then `GreedySchedule` returns from S some $k^* \leq k$. The claim is that $k \leq k^*$. We know $f_1 \leq f_1^*$ by the definition of `GreedySchedule`. But $f_1^* \leq s_2^*$ by definition, so I_{f_1} contains $\{(s_2^*, f_2^*), (s_3^*, f_3^*), \cdots, (s_{k^*}^*, f_{k^*}^*)\}$.

We want to show that k is big. We have $k = |$`GreedySchedule`$(I)| = 1 + |$`GreedySchedule`$(I_{f_1})| = 1 + |\mathrm{OPT}(I_{f_1})|$ by the induction hypothesis. So this is greater than $1 + A$, for A *any* disjoint subset of I_{f_1}. Take $A = \{(s_2^*, f_2^*), (s_3^*, f_3^*), \cdots, (s_{k^*}^*, f_{k^*}^*)\}$, so $\mathrm{OPT} \geq 1 + (k^* - 1) = k^*$. Therefore $k \geq k^*$. ⊠

Example 7.1. Let us discuss another example. Consider files with length $L_1, L_2, \cdots, L_n$. If placed in order $\pi_1, \pi_2, \cdots, \pi_n$[2] (permutations of $[n]$), we have $\mathrm{cost}(k) = \sum_{i:\, \pi_i \leq \pi_k} L_i$. Our goal is to define the layout π minimizing the average $\frac{1}{n}\sum_{k=1}^{n} \mathrm{cost}(k)$. Every file access has to access the first one, so you want to make sure the earliest ones are smallest. How do we prove this?

Suppose we exist we have some two out of order.

swapping $\to$ bubble sort. know bubble sort gets to a better solution, which implies quicksort also gets a better solution.

8 Stable marriage

We want to match n students and n positions. Each student/position has a ranked order of preference. Replace students/positions with men/women and we get **stable marriage**, where the matching is "stable"; we need partners to prefer their matches. The goal is the following: given preference lists, find a stable matching.

[1] Details: first sort the f_i by earliest end time $(\log n)$, then take first interval (s,f). Repeatedly remove the first interval (s',f') if $s' < f$.

[2] ***Not*** the homotopy groups!

Example 8.1. Let Men $= \{A, B, C\}$ and Women $= \{X, Y, Z\}$. Here A prefers $Y > X > Z$, B prefers $Y > X > Z$, C prefers $X > Y > Z$. Unanimously the women prefer $A > B > C$. The pairing is then $(A, Y), (B, X), (C, Z)$.

Another example. The preferences are:

(A) $X > Z > Y$, (X): $B > A > C$
(B) $Z > X > Y$, (Y): $C > A > B$
(C) $X > Y > Z$, (Z): $A > B > C$

First you form the initial assignments, assigning $(A, X), (B, Y), (C, Z)$. This is not stable. (B, Y) will break up, (C, Z) will break up, A prefers Z, ?? follow the tree, four breakups will go back to where we start.

It is not obvious that any stable solution exists, and it is quite surprising that we can prove that one exists. This is proven by the algorithm we will give.

8.1 Gale-Shapley Stable Marriage Algorithm

We start with nobody matched. We repeat the following process: Let Men $= \{A_i\}$, Women $= \{X_i\}$ for $i \in \{1, 2, \cdots, n\}$. For $n = 3$, pick any unattached man. If each man A_i has preference $X_{\sigma_i(1)} > X_{\sigma_i(2)} > \cdots$ for some permutation $\sigma_i \in S_n$, A_i will ask $X_{\sigma_i(1)}$ on a date. If $X_{\sigma_i(1)}$ is unattached, she will accept (or if she prefers to her current partner). If the man is rejected, the man crosses her off his list (so if we return to A_i, the next candidate will be $X_{\sigma_i(2)}$). Repeat forever. We go down to the bottom of the list, where someone will eventually accept. So this matches everybody.

Theorem 8.1. *The Gale-Shapley Stable Marriage Algorithm returns a stable matching in order $O(n^2)$ time.*

Proof. We need to show that this algorithm terminates first. To do this, we need to make progress every step of the way. In each round, either a man crosses a woman off, or a woman moves up the ladder (improved happiness by rejection). We never uncross the list, and women never get traded down. The maximum number of things we can cross off is n^2 (men times women), and you can do n^2 improvements per woman. Overall, there are only $O(n^2)$ rounds. This shows that the algorithm terminates, and the final solution is indeed a matching (if there wasn't, repeated with the unattached man).

The question is, is this matching stable? Suppose the matching is not stable. That means there exist (A, B, X, Y) such that (A, X), (B, Y) are matched but (WLOG) (A, Y) would elope. This means $Y > X$ on A's list, and $A > B$ on Y's list. How did A get mapped to X? This implies A proposed to X at some point, which means X is at the top of A's list. This can only happen if Y crossed A off. So at some point, A proposed to Y and was rejected. Why? Because at some point, Y was rejected by some $C > A$. This is a problem because no trading down implies that the final match $B > A$ on Y's list, contradicting the assumption. Therefore the marriage is stable. ⊠

Example 8.2. Consider the pairing

(A) $X > Y > Z$, (X): $C > B > A$
(B) $Y > Z > X$, (Y): $A > C > B$
(C) $Z > X > Y$, (Z): $C > B > A$

People end up bery unhappy. ?? Here Gale-Shapley is optimal for the men (proposers) and pessimal for women (receivers). That is to say, let S_A denote the set of possible matches for A in any stable assignment. Then best(A) is the highest ranked in S_A, analogously worst(A) is the lowest ranked in S_A. Gale-Shapley then sends $A \to$ best(A) for every man A and sends $B \to$ worst(B) for every woman B. This is sort of surprising, let us prove it.

Lemma 8.1. *Each man is only rejected by women who cannot match them in any stable matching.*

Proof. We use induction on the number of steps taken (rounds). After 0 steps, this is true because no man has been rejected by anybody. We know by induction on previous rounds that this is true on the $(k-1)$th round. In a given round, if a man A is rejected by a woman X, that means she prefers B. This implies that B has X at the top of his list (everybody above X is crossed off). By induction, these rejections resulted in unstable matchings. This implies in any stable matching, B matched to X or lower on his list. So in any stable matching, X is matched to B or higher on her list. This means $A < B$ or (X, A) are not matched. Each man then gets the best possible outcome, or $A \to \text{best}(A)$ for every man A. ⊠

9 ok

10 Graph searching

Consider a graph $G = (E, V)$, where E is a set of pairs of elements of V. Then $E \leq \binom{V}{2}$ if G is directed, and $E \leq V(V-1)$ if G is directed. Adjacency matrix, adjacency list (how to store). linked list. list comparing complexities. hash table. binary search tree. help i didn't take data structures. i also cannot read the board.

look at the complexity list in the textbook.

Reachability: given $G = (V, E)$ with source s, the goal is to find all the vertices u reachable from s.

Let visited equal {}. Depth first search:

```
def DFS(v)
    if v in visited return
    visited.add(v)
    for w in v.adj:
        DFS(w)
```

Now we do Breadth first search (BFS). It visits vertices in order of distance from s. prinn's minimum spanning tree algorithm.

11 graph reductions

The big idea is to turn things into graphs then run BFS on them.

Example 11.1. snakes and ladder, with max k and n^2 board size.

```
#creating the graph G
for i in [n^2]
    for j in [k]
        if (i+j,y) for j in [k] is in (set of snakes/ladders)
            create edge (i,y)
        else
            create edge (i,i+j)

#run BFS on the graph
BFS(G)
```

Example 11.2. mnongolian puzzle, swapping tokens. Let A be an $n \times n$ matrix. make the state of the graph as a vertex (two points $A_{ij} = n$, $A_{k\ell} = m$). Each vertex is a transformation $((A_{ij}, A_{k\ell}), (A_{(i+m,j)}, A_{(k+n,\ell)}))$ etc depending on which moves are valid (ie $A_{(i+m,j)}$,$A_{k+n,\ell}$) satisfies $i+m, j, k+n, \ell \in [n]$). Then run BFS on G (graph made according to the above rules) to find the shortest path from (A_{in},A_{ni}) to (A_{ni},A_{in}).

12 DFS and topological sort

dags have a topological ordering.

13 A* search

Recall: Dijkstra.

```
Dijkstra(G,S):
    parent, dist = {}, {}
    Q = priority queue([(0,S, None)])
    while Q:
        d, u, p = Q.pop()
        if u in dist:
            continue
        parent(u)=p, dist[u]=d

        for v in u.adj:
            Q.append((dist[u]+w(u -> v),v,u))
        return dist, parent
```

append and pop gives runtime of $O(E \log V)$ or $O(E + V \log V)$. doesn't work for negative edges (simple counterexample). modify the line `if u in dist` to `if u in dist and d = dist[u]`. this is correct, but no longer visits each vertex once, visit them an exponential amount of times. so this becomes a lot worse than bellman ford. but it's correct.

example of exponential time: "spring graph". one half is minimal positive, while the bottom half is big positive then big negative. this takes $n!$ iterations (goes to the end, loops back. then starts from second and goes all the way to the end, loops back). so with k negative edges, this is a 2^k factor, but with only one it's alright. correct and slow is better than fast and wrong.

say we have a destination in mind, `Dijkstra(G,S,t)`. without negative weights, once we pop t off the queue. so after the line `parent[u]=p, dist[u]=d`, add the line `if u==t, break`. basically, stop after visiting t. well what about negatives.

13.1 A* search

dijkstra visits a lot of extra things (eg consider map of the US, get from austin to NY/san fran. visits almost all the continental US in the meantime). how to fix this? have some sort of measure of where the destination is. this leads to $A*$ **search**. we use a **heuristic** or **potential** h. $h(u)$ is some measure of distance u to t. Idea: visit vertices in order of increasing $\text{dist}(s,u) + h(u)$. the heuristic prefers to visit vertices closer to our destination.

alternative view: running Dijkstra shortest paths on a *reweighted graph G* with weight $w_h(u \to v) := w(u \to v) + h(v) - h(u)$. think of it as taking each edge and "adjusting" the weight. let $\text{length}_h(u_1 \to u_2 \to \cdots \to u_n) = \text{length}(u_1 \to u_2 \to \cdots \to u_n) + h(u_k) - h(u_1)$ (since the other terms will all cancel). this length doesn't depend on path, only depends on start and end. therefore shortest path in w_h is precisely the shortest path in w. so

$\text{dist}_h(s,u) = \text{dist}(s,u) + h(u) - h(s)$. running Dijkstra on this reweighted graph sorts by distance under reweighted graph, sorting by $\text{dist}(s,u)+h(u)-h(s)$. but $h(s)$ is the same. therefore Dijkstra on $w)h$ visits u in order of increasing $\text{dist}(s,u) + h(u)$. so these two ideas are the same.

when does this work? we could choose any h and this equivalence is true. what do we need from h? for correctness, when we visit t we should be done. one condition is **admissibility**, that is, h is said to be admissible if $h(t) = 0$ and $h(u) \leq \text{dist}(u,t)$ for every u. if this holds, then when we visit t, every unvisited u has $\text{dist}(s,u)+h(u) \geq \text{dist}(s,t) \implies \text{dist}(s,u) + \text{dist}(u,t) \geq \text{dist}(s,t)$. this implies the shortest s leads to a path. so A^* returns the correct shortest path.

well correctness is good, but we want it to be *fast*. how do we make this polynomial time? we don't want it to be worse than regular dijkstra, ie not worse than $E(\log V)$ time. so we require the heuristic to be non-negative, since this is what causes dijkstra to be slow. the name is **consistent**. that is, $w_h(u \to v) \geq 0$ for all u, v. this is equivalent to saying that $w(u \to v) + h(v) - h(u) \geq 0 \iff h(u) - h(v) \leq w(u \to v)$. so if we are consistent and $h(t) = 0$, then we are admissible. why? we know that $\text{dist}_h(u,t) \geq 0$ by consistency. but $\text{dist}_h(u,t) = \text{dist}(u,t) - h(u)$ (removing $h(t)$), which implies $h(u) \leq \text{dist}(u,t)$. So A^* runs in $O(E + V \log V)$ time. this shows that A^* may not necessarily be faster than Dijkstra, but it is not worse.

next pset; we will run this on some examples. popular example: set $h(u) = \|u - t\|$, ie $w(u \to v)\|u - v\|_2$. so $h(u) - h(v) = \|u - t\| - \|v - t\| \leq \|u - v\| = w(u \to v)$ by the triangle inequality. so h is consistent and admissible. the general picture is that the DP may make you consider going through houston to LA from Austin, but not through NY. ie instead of considering most of the continental US in concentric circles, we consider a smaller radius around the "general direction" to LA.

there might be other ways, ie getting off the highway at phoenix and looking around for a direct highway through local roads. the algorithm is conservative (and consistent), always considers going through a small street. with this heuristic it's hard to change the "level of conservativeness" or "aggression" of the algorithm. in the next pset we will look at the ALT heuristic, rather than Euclidian distance it runs a few shortest paths. this never gets you off the highway in some small town.

13.2 All-pairs shortest paths

Last week we were talking about SSSP (single source shortest paths), with Dijkstra and bellman ford. the question now is, what about all pairs shortest paths (APSP)? for every pair of vertices, what's the shortest way to get there. one idea is to run SSSP for all $s \in V$. this gets $O(VE + V^2 \log V)$ if non-negative, and $O(V^2 E)$ if negative.

another cute/short algorithm is the following. this is called **Floyd-Warshall** (APSP algo). we start with a distance matrix

$$\text{dist}(u,v) = \begin{cases} w(u \to v) & \text{if } u \to v \in E \\ \infty & \text{otherwise} \end{cases}$$

Then the algorithm is as follows:

```
for u in V:
    for v in V:
        for w in V:
            dist(v,w)=min(dist(v,w),dist(v,u)+dist(u,w))
```

This has runtime $O(V^3)$ and works for negative edges. so better than running Bellman ford on everything, matches Dijkstra on dense graphs, but worse than Dijkstra on sparse graphs. intuition: it is important that the outer for loop corresponds to the inner vertex.

this is all and good, but wouldn't it be nice if we could get the runtime $O(VE + V^2 \log V)$ even for negative edges. this is called **Johnson's algorithm**, with runtime $O(VE + V^2 \log V)$ with negative edges. the general idea:

(1) find any consistent heuristic h

(2) run Dijkstra on w_h for all sources s

the hard part is finding a consistent heuristic, which will take VE time. what algo takes VE time? that's right, bellman ford. so

(1) run bellman ford from an arbitrary v^* that can reach the entire graph. then set $h(u) = -\text{dist}(v^*, u)$. we have $\text{dist}(v^*, v) - \text{dist}(v^*, u) \leq \text{dist}(v^*, v) \leq \text{dist}(v^*, u) + w(u \to v)$. so $\text{dist}(v^*, u) \leq \text{dist}(v^*, u) + w(u \to v)$, which is precisely $h(u) - h(v)$. now we can run Dijkstra, we converted this tricky graph into a nice graph that has the same set of shortest paths.

14 Linear programming

Regarding the exam, apparently some of us made "incomprehensible" mistakes. Okay onto new topic.

Onto linear programming (LP). Say we can produce cars, trucks, and cars take 2 metal, 1 wood, trucks take 3 metal and 5 wood. We have 12 metal, 15 wood. Trucks carry 2x as much as cars. Question: what to product to maximize amount we can carry?

Let T be amount of trucks and C the amount of car, and we can carry $2T + C$ things. Then $3T + 2C \leq 12$ (metal constraint) and $C + 5T \leq 15$. Furthermore $T, C \geq 0$. todo:figure orthogonal line, zero inner product?

Formally; we have a **feasible region**, a set of (T, C) satisfying constraints, and the answer is at a vertex of such feasible region (regardless of objective). Polytopes- intersection of half spaces.

Let's solve this by hand. There are two variables and four constraints. This results in four vertices and two more "vertices", intersections of constraints. What about d variables and n constraints? Then vertices are defined by d constraints, and there are at most $\binom{d}{n} \leq n^d$ vertices. Suppose $T = 0$, then $C = 0$ or $C = 6$ ($C = 15$ violates $3T + 2C \leq 12$). If $C = 0$, then $T = 3$ ($T = 4$ violates $5T + C \leq 15$). So $3T + 2C = 12, 5T + C = 15$, so $7T = 18$, and $T = 18/7, C = 15/7$. Plug in our four vertices: $(0,0), (6,0), (0,3), (15/7, 18/7)$: the highest yield is the last one, getting $51/7$. Then we are done.

Simple algo: start at a vertex, and walk to a neighbor with a higher objective. A vertex is defined by d constraints, and a neighor is one where you remove one of the three constraints. Repeat.

14.1 General linear programming

Optimize (max or min) linear objective, subjected to linear constraints $(\geq, =, \leq)$.

Example 14.1. Maximize $5x_1 + 6x_2 - 3x_3 + x_4$ such that $x_3 \geq x_1 + 2x_2 + 3, x_4 = x_1 + x_2, x_1 \geq 0, x_2 \leq 1$.

This can get complicated, so we often walk about the **standard form** (or symmetric). We change the form and write it in linear algebra terms. We want to maximize $c^T x$[3] for $Ax \leq b$ and $x \geq 0$. What we mean is that after expanding the matrix multiplication, all the resulting constraints from Ax are less than or equal to b.

So for our example, $A = \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix}, b = \begin{bmatrix} 12 \\ 15 \end{bmatrix}, c = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. Two other common forms include the **alternative form** (or asymmetric). Here we want to maximize c^T for $Ax \leq b$. There is also the **equational form**, which wants to maximize $c^T x$ such that $Ax = b, x \geq 0$.

[3] c^T refers to coefficients.

Claim. *The standard, alternative, and equational forms are equivalent problems. In other words, given an algorithm to solve any one, we can solve the others by transforming the input or output.*

Proof. (Standard form → alternative form). Simply add a negative identity at the bottom.

$$\begin{bmatrix} A \\ -I \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \leq \begin{bmatrix} b \\ 0 \end{bmatrix}$$

So $Ax \leq -x < 0 \iff x \geq 0$.

(Alternative form → standard form). Let $x = x' - x''$. So

$$Ax \leq b \iff \left[\begin{array}{c|c} A & -A \end{array} \right] \begin{bmatrix} x' \\ x'' \end{bmatrix} \leq b$$

So our objective is to maximize $\left[\begin{smallmatrix} c \\ -c \end{smallmatrix}\right]^T \left[\begin{smallmatrix} x' \\ x'' \end{smallmatrix}\right]$ such that $\left[\begin{array}{c|c} A & -A \end{array} \right] \left[\begin{smallmatrix} x' \\ x'' \end{smallmatrix}\right] \leq b, x', x'' \geq 0$.

(Equational → standard). Given $Ax = b, x \geq 0$, we have $Ax \leq b, Ax \geq b, x \geq 0$. Negative $Ax \geq b$ to get $-Ax \leq -b$, which is, in fact, in standard form.

(Standard → equational). The trick is to add *slack* variables. We want to go from $Ax \leq b, x \geq 0$ to $Ax = b, x \geq 0$. The former is true iff $Ax + s = b, x \geq 0, s \geq 0$ (the slack variables represent the difference).

$$\begin{bmatrix} A \\ I \end{bmatrix} \begin{bmatrix} x \\ s \end{bmatrix} = b,\ x, s \geq 0,\ \text{maximize} \begin{bmatrix} c \\ 0 \end{bmatrix}^T \begin{bmatrix} x \\ s \end{bmatrix}.$$

⊠

14.2 Algorithms to solve LP

We briefly discussed the **simplex algorithm**, which is to walk along the vertices. None of the algorithms we have work well (exponential in worst case scenario) but are pretty good in practice. So they fail in theory (unless smoothing). Then came the **ellipsoid method**. The point is we have an ellipse around the set of solutions, and we repeatedly shrink it. This works in theory but not very well in practice. Then came long **interior point methods**, which go through the middle of the region. A popular one runs in $O(n^4 L)$ time, which L represents the bits of precision, n is the number of variables plus the number of constraints. It is an open question whether or not LP is strongly polynomial.

There is a remaining question for the simplex algorithm. We described how it works; find a vertex, find its neighbors, pick a random/best improvement. We did not explain how to start at a vertex. It turns out finding an initial is as hard as LP in general. What we do is look at a different version of the problem with a simple solution (eg $x = 0$), and if this is feasible we move around.

15 Duality in LP

todo:this

16 Complexity theory

We have seen lots of algorithms to solve many problems. Can we classify problems by difficulty? What can't we solve? The answer to this question "what can't we solve" is almost everything. An informal statement: take a known solvable problem, and tweak it, the result is probably not solvable.

Example 16.1. We can solve shortest path. What about the *longest* simple path. Nope. We can solve minimum spanning tree, but what about **minimum Steiner tree**, just connecting some subset $S \subseteq V$. We don't know how to solve it in polynomial time.

We have talked about a min (s,t) cut, what about a max (s,t) cut? We don't know. In flows, we talked about max (s,t) flow. What about multi-commodity flow? Nope, even with just two commodities. We can solve interval packing, which has job packing as an application—what about preferences? We don't know.

We can say *something* however about these problems, which is the theory of NP completeness. We classify problems into "complexity classes". Let n be the size of the input in bits, and only consider **decision problems** (the answer is yes or no).

Example 16.2. Shortest path is an optimization problem. Let (G,s,t,k). To rephrase this as a decision problem, say "does there exists a (s,t) path in G of length $\leq k$". Then try different ks on some binary search to find the smallest one. The number of bits n wrt V,E,s,t,k,w (where w represents edge weights) is $V\log V$ for vertices, $E\log V$ for edges, $2\log V$ for s,t, $E\cdot w$ for edge weights, $w+\log E$ (since $k \leq E\cdot 2^w$). Summing this up, we get $n = O(E\log V + EW)$.

Complexity theorists care about classes. The class P contains problems solvable in **polynomial time**, $n^{O(1)}$. This includes most problems we covered in class. The class NP contains problems that a solution can be *verified* in polynomial time. A prototypical example is the traveling salesman problem (cycle visiting all cities exactly once). Formally:

Definition 16.1 (NP)**.** There exists a polynomial time verifier A such that **completeness** holds: for all YES instances x, there exists a proof $y \in \{0,1\}^{\text{Poly}(n)}$ such that $A(x,y) = \text{YES}$. Another condition **soundness** must hoold: for all NO instances x, there must not exist a proof $y \in \{0,1\}^{\text{Poly}(n)}$ such that $A(x,y) = \text{YES}$.

We have $P \subseteq NP$. There are some problems not in NP, for example, *how many* longest paths are there? Or, does white win this chess position? The final problem is the **halting problem**, which is not even computable.

We can say many problems are **equally hard**: they are all in P or none in P (NP-completeness). We do this by **reductions**. To show that problem A is harder than B (say TSP), tragically my laptop ran out of battery at this point in time

17 Reductions

Dr. Price forgot his notes at home today, so the lecture will be slightly scatterbrained. Recall: P consists of problems solvable in polynomial, NP consists of verifiable problems in polynomial time, NP-hard problems once solved can solve all NP problems. Today we show how to show problems are NP-hard, hence implying they are probably not in P.

Cook's Theorem. *Circuit SAT is NP-complete, i.e., CSAT is both in NP and NP-hard.*

A description of CSAT is as follows: "Does there exists an input such that a given circuit outputs 1?"[4] We can see that CSAT is in NP because we just plug it in. Furthermore, CSAT reduces to SAT, which consists of satisfiable boolean *formulas*. It is easy to formulate a SAT formula as a SAT circuit, but the other way is not as easy. The idea is that you make a new variable for each wire. The formula is "3-CNF" (3-conjunction normal form), the intersection of unions of three clauses at a time. So CSAT, SAT, and 3SAT are all NP-complete.

[4]Here circuit refers to set of logic gates, a Boolean circuit.

Consider the problem max independent set, where given a graph G, an *independent set I* is a subset of V such that no edge $e \in E$ has $|e \cap S| = 2$. The max independent set asks if there exists an independent set S of size $|S| \geq k$. The claim is that max independent set is NP complete, or that it in NP and is NP-hard.We do this by reducing from 3SAT. Recall that reducing from X (3SAT) to Y (max independent set) means that we can solve X (3SAT) from Y (max independent set), or Y (max independent set) is no harder than X (3SAT, hard). How do we do this?

(1) Construct a transformation from an *arbitrary* X instance x to a *special* Y instance y.
(2) If x is a YES instance, then y is a YES instance.
(3) If y is a YES instance, then x is a YES instance.

Given a 3SAT instance $(a \cup b \cup c) \cap (b \cup \overline{c} \cup \overline{a}) \cap (a \cup b \cup d) \cap (a \cup \overline{c} \cup d)$, this is YES because set a and b to be true. Make a graph with vertices as above, grouped into threes. Edges just form some strongly connected components (formally called a "clause gadget"). Now we need some variable gadgets, expressing the fact that each triple relates to each other. We do this by drawing an edge from each instance of a to each instance of $\overline{a}$, b to $\overline{b}$, etc.

The claim is that if 3SAT is satisfiable, then the corresponding graph has an independent set of size greather than or equal to the number of clauses. 3SAT being satisfiable means that there exists an assignment of x_i to $\{0,1\}$ such that each clause has greater than or equal to 1 satisfied variable. For our specific instance, our assignment is $a = 1, b = 1, d = 1, c = 0$ (one for each clause). Take S to be one of those satisfied variable vertices per clause gadget, where $|S| = \#$ of clauses, then S is independent. So if 3SAT is satisfiable, there exists a large independent set.

Now we need to show the other direction, that is, if our independent set instance is a YES, then so is the 3SAT instance. First observe than in each clause, S has exactly one vertex for each clause gadget. Furthermore, S never has both $x_i$and $\overline{x}_i$ labeled vertices. Set

$$x_i = \begin{cases} 1 & \text{if any } x_i \text{ vertex} \in S \\ 0 & \text{if any } \overline{x}_i \text{ vertex} \in S \\ 0 & \text{otherwise} \end{cases}$$

This assignment is consistent and satisfies all clauses. So this is a complete NP hardness proof.

Example 17.1 (Mario)**.** Now for a "fun" reduction example, Super Mario Bros the original. This is NP hard. We can construct a giant level in Mario Maker, and we detail why solving a general Mario level is NP hard. We do this by reducing from 3SAT. Now our clause gadgets and variable gadgets, rather than being graphs, are pieces of a level. Mario often needs to make a choice if some thing x_1 is true or false (variable gadgets), some choice x_2, etc. Then he runs back through clause gadgets to make it to the final. Zelda is also NP hard (opening doors with swords).

???? Stars/mushrooms? So clauses are like stages. So someone could give you a password, to PSAT, to SAT, to 3SAT, to a certain Mario level. Zelda is even harder, it is PSPACE-hard. In order be in NP, we need to be able to check an answer; it takes exponential time to prove (play levels, due to backtracking).

Karp's 21 problems

18 Computability Theory

Today we talk about *uncomputable* problems, specifically the **halting problem**. This is a problem with two inputs,

$$\text{Halt}(f, x) := \text{will program } f \text{ halt on input } x \text{ rather than an infinite loop}$$

First we show that Halt is NP-hard. We reduce TSP to Halt as follows. We write a slow algorithm for TSP (try all paths). Then define

```
def TSP(X):
    (try all solutions)
    return answer

def f(X):
    if TSP(x) == False:
         while 1: pass
    else: return True
```

This reduces TSP to the halting problem, so TSP is true iff the halting problem is true. We did not use anything about TSP specifically, just *some* slow problem to solve. So we have TSP $\preccurlyeq_P$ Halt (polynomial time reducible), furthermore $g \preccurlyeq_P$ Halt for every computable function g. This shows the halting problem is as hard as any computable function, but does not show it is not computable at all.

To show Halt is uncomputable, we show a paradox. Suppose Halt is computable then we write the following code with input a program x:

```
import Halt

def g(x):
    if Halt(x, x):
        while 1: pass
    else:
        return
```

This is a program. What does $g(g)$ do? It asks if g halts on g or not. If it does halt, it doesn't halt. If it doesn't halt, it halts. Neither of these is consistent, a paradox, so we conclude there is no code to solve the halting problem on every input and always terminates with the correct answer.

Suppose you want to solve the halting problem in practice. You could use print statements. If there's a bound on states we can check the amount of states. However the easiest thing we can do it to see how long it's *supposed* to run in. Run $f(x)$ for L steps, for L some large number. If it has terminated, return true, and else false. This is correct if L is bigger than the time it takes to compute $f(x)$ if (f, x) is n bits long.

busy beaver. We can solve it using the halting problem. Define

```
def Halt(f,x):
    L \geq B(|f(x)|)
    run f(x) for L steps
    return True if f(x) halts
```

There exists a solution to Halt on all n-bit programs. The code for Halt is $n + O(1)$ bits long. This says that there exists a way of checking smaller programs, but there is no way to check larger programs. How do we know if L is big enough? The answer is you can't.

Gödel's second incompleteness theorem

This says that no consistent set of mathematical axioms can prove their own consistency. Rip ZFC. Define a function

```
def FindContradictionInZFC()
    for s in {0,1}^*
        check if S is valid ZFC proof that True = False
        if so, return s
```

This program halts iff ZFC is inconsistent. Therefore we cannot prove that this program does not halt. So ZFC cannot prove its own consistency, which implies we cannot prove that FindContradictionInZFC doesn't halt. Say FindContradictionInZFC has 10^6 megabytes, therefore we cannot prove any upper bound on BB(10^6).

This is a lazy upper bound of 10^6. Can we understand busy beaver numbers better? Let's be more formal about the languages. Let $S(n)$ be the maximum number of steps taken by any n state Turing machine on a blank binary tape before halting. What is a Turing machine? It's a particular formalization of computing, it takes in a tape, and starts at a head. There is an internal state (say 1), and a huge table says that "if we are at a certain state" it reads under the tape, and it can

- edit
- move left or right
- halt
- add a new state

Then $S(1) = 1, S(2) = 6, S(3) = 21, S(4) = 107$, and $S(5) = 47$ million (probably, about 15 of them might halt). We have $S(6) \gg 10^{36000}$, $S(7) \gg 10^{10^{\cdot^{\cdot^{\cdot}}}}$. These number have to grow very fast since they're not computable. By earlier, this is a fixed number where we can't formally prove an upper bound on $B(10^6)$. This is a compilers problem; we have easy to write code, now we compile that code into the smallest Turing machine possible. Aaronson showed we could write this as an 8000 state Turing machine (now 2000), so $S(2000)$ is impossible to prove an upper bound on in ZFC. Another program in $S(741)$, which we don't know if its possible to prove an upper bound, but if it were we could reduce the Riemann hypothesis to a finite number of cases.

Models of computation

At the same time Turing was coming up with Turing machines, Alonzo Church was coming up with Church numerals and Lambda calculus. So there are two competing models on how to do computation.

Theorem 18.1. *The set of functions computable by Turing machines is equal to the functions computable by Church numerals.*

The Church-Turing Thesis (conjecture) is that both of these are equal to the set of functions computable by any "physical process". Computer scientists came up with the *extended* Church-Turing thesis, which says that the set of polynomial time computable functions is the same in every physical process. This statement is probably false, because quantum computers can solve BQP problems. In particular, factoring lies in BQP and factoring probably doesn't lie in P.