Gabriel Uresti

CS 561 HW1

1) $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 s.t. $0 \le f(n) \le cg(n)$ for all $n \ge n_0$

a) $0 \le 3^{n+1} \le c\,3^n$

$0 \le 3 \cdot 3^n \le c \cdot 3^n$

with $c = 3$, $n_0 = 1$,

$0 \le 3^{n+1} \le 3 \cdot 3^n$ for all $n \ge 1$

b) $0 \le 3^{2n} \le c\,3^n$ Divide by 3^n

$0 \le 3^n \le c \cdot 1$

3^n is not $\le$ any constant if n is large enough so $3^{2n} \neq O(3^n)$

2)

	A	B	O	o	Ω	ω	Θ
a	$\lg^k n$	n^ε	yes	yes	no	no	no
b	n^k	c^n	yes	yes	no	no	no
c	$\sqrt{n}$	$n^{\sin(n)}$	no	no	no	no	no
d	2^n	$2^{n/2}$	no	no	yes	yes	no
e	$n^{\log c}$	$c^{\lg n}$	yes	no	yes	no	yes
f	$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

3. a) $\log(n!) = \Theta(n \log n)$ if $\log n! = \Omega(n \log n) \cap O(n \log n)$

Upper bound: $\log(n!) = \log(n(n-1)(n-2)\ldots 2 \cdot 1)$ $\quad \log(xy) = \log x + \log y$

$\le \log(n) + \log(n-1) + \log(n-2) \ldots + \log 2 + \log 1 \le \log(n) + \log(n) \ldots + \log(n)$

$\le n \log(n)$ $\quad c = 1$ $n_0 = 2$

So $\log n! = O(n \log n)$

Lower bound:

$\log(n!) = \log(n) + \log(n-1) \ldots + \log(n/2) \ldots + \log 2 + \log 1 \ge \log(n) + \log(n-1) \ldots + \log(\frac{n}{2})$

$\ge \log(n/2) + \log(n/2) \ldots + \log(n/2)$

$\ge \frac{n}{2} \log(n/2)$ $\quad c = \frac{1}{2}$ $n_0 = 2$

So $\log n! = \Omega(n \log n)$ for $c = \frac{1}{2}$ $n_0 = 2$

So $\log(n!) = \Theta(n \log n)$, $\frac{1}{2} n \log n \le \log(n!) \le 1 \cdot n \log n$ for all $n \ge 2$

b) $\omega(2^n) = n!$ if for any $c > 0$ there exists $n_0 > 0$ s.t. $0 \le c \cdot 2^n < n!$ $\quad n \ge n_0$

$2^n = \underbrace{2 \cdot 2 \cdot 2 \ldots 2}_{n \text{ times}} < 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$ for $n \ge 4$

For $n < 4$, Ex: $2^3 = 2 \cdot 2 \cdot 2$, $3! = 3 \cdot 2 \cdot 1$ (terms 2 & 3 $\le$ onward)

gives us trouble at $c > 1$

multiply by 2 so $2c$ and $c > 0$

$0 \le c \cdot 2^n < n!$ for $n \ge \text{Max}(4, 2c)$ so $n! = \omega(2^n)$

$c = \frac{1}{4}$
$c = .5$ $n_0 = 2$
$n! = 2 < n^n = 2$ for $\frac{1}{c}$

2)c) $o(n^n) = n!$ if $\forall c > 0 \; \exists n_0 > 0$ s.t. $0 \le n! < cn^n \quad \forall n \ge n_0$

For $c \ge 1$ and $n \ge 1$: $n^n = \underbrace{n \cdot n \cdots n}_{n \text{ times}} > \underbrace{n \cdot (n-1) \cdots 2 \cdot 1}_{n \text{ values multiplied}}$

For $0 < c < 1$ worried about $n = 2$ and $n = 1$

if $n_0 = \frac{2}{c}$ ← because $c = .5$ results in equality for $\frac{1}{c}$

then for $n = n_0$ $n!$ must be $< c \cdot \frac{2}{c} n^{n-1}$

$n!$ must be $< 2n^{(n-1)}$

for $n = 2$ $2! = 2 < 2(2)^1 = 4$ ✓

for $n = 1$ $1! = 1 < 2(1)^0 = 2$ ✓

so $n_0 = \text{Max}(\frac{2}{c}, 2)$

$0 \le n! < cn^n \; \forall \; n \ge \text{Max}(\frac{2}{c}, 2)$ therefore $n! = o(n^n)$

4a) $\log_2 f(n) = O(\log_2 g(n))$ if $\exists c > 0$ & $n_0 > 0$,

$0 \le \log_2 f(n) \le \log_2(cg(n))$ for $n \ge n_0$

$\log_2 f(n) \le \log_2 c + \log_2 g(n)$

for any coefficient in $f(n)$ we can adjust $c \ge$ than that coefficient, They have the same base and leading term so True

b) False, if $f(n)$ is $2n$, similar to 1b), then $O(g(n)) = O(n)$

$2^{(f(n))} = 2^{2n}$ which must be $\le c \cdot 2^n$

divide both sides by 2^n

$2^n \not\le c$ and there is no constant that can bound 2^n so $2^{(f(n))}$ is not $O(2^{g(n)})$

c) $f(n^2)$ is $O((g(n))^2)$ True

$f(n)$ and $g(n)$ have the same leading term

$f(n)^2 \le c \cdot g(n)^2$

whatever coefficient in $f(n)$ gets squared, we can find $c \ge$ the coefficient, and squaring the leading terms will have the same growth rate.

5a) Since there are n elements and each one has equal probability of being selected, the probability of a particular element being chosen is $1/n$, so $E[X_i] = 1/n$

b) $X_q = \begin{cases} 1 \text{ if } q \text{ is chosen as pivot} \\ 0 \text{ otherwise} \end{cases}$

partition = $\Theta(n)$ runtime because quicksort has to look at all the elements except the pivot

$E[T(n)] = E[T(\text{elements lower than partition}) + T(\text{elements higher than partition}) + T(\text{partition})]$ | if $X_q = 1$:

$= E[T(q-1) + T(n-q) + \Theta(n)]$

Using Linearity of Expectation for all pivot selections,

$E[T(n)] = E[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))]$

c) $E[T(n)] = E[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))] \overset{LOE}{=} \sum_{q=1}^{n} E[X_q (T(q-1) + T(n-q) + \Theta(n))]$

$= \sum_{q=1}^{n} E[X_q] E[T(q-1) + T(n-q) + \Theta(n)]$ $E[X_q] = 1/n$ so $= 1/n \sum_{q=1}^{n} E[T(q-1) + T(n-q) + \Theta(n)]$

$\overset{LOE}{=} 1/n \sum_{q=1}^{n} E[\Theta(n)] + 1/n \sum_{q=1}^{n} (E[T(q-1)] + E[T(n-q)])$

$= 1/n \cdot \Theta(n) + 1/n \sum_{q=1}^{n} (E[T(q-1)] + E[T(n-q)])$

$= \Theta(n) + \frac{2}{n} \sum_{q=1}^{n} E[T(q-1)] = \Theta(n) + \frac{2 E[T(0)]}{n} + \frac{2 E[T(1)]}{n} + 2/n \sum_{q=3}^{n} E[T(q-1)]$

$q - 1 = n - (n - (q-1))$ so

(linear runtime)

$= \Theta(n) + 2/n \sum_{q=2}^{n-1} E[T(q)]$

d) $\sum_{k=2}^{n-1} k \log_2(k) = \sum_{k=2}^{n/2-1} k \log_2(k) + \sum_{k=n/2}^{n-1} k \log_2(k)$

$\leq \sum_{k=2}^{n/2-1} k \log_2(n/2) + \sum_{k=n/2}^{n-1} k \log_2(n)$

$= \log_2(n/2) \sum_{k=2}^{n/2-1} k + \log_2(n) \sum_{k=n/2}^{n-1} k = \log_2(n) \sum_{k=2}^{n-1} k - \log_2(2) \sum_{k=2}^{n/2-1} k$

$\left(\sum_{k=2}^{n/2-1} k + \sum_{k=n/2}^{n-1} k\right)$

$= \log_2(n) \left(\frac{n(n-1)-2}{2}\right) - \left(\frac{\frac{n}{2}(\frac{n}{2}-1)-2}{2}\right)$

$= \log_2(n) \left(\frac{n^2-n-2}{2}\right) - \frac{n^2-2n}{8}$

$= 1/2 n^2 \log_2(n) - 1/8 n^2 - n/4 - 1 - n \log_2(n) - \log_2(n)$

$\leq 1/2 n^2 \log_2(n) - 1/8 n^2$

e) $E[T(n)] = 2/n \sum_{q=2}^{n-1} E[T(q)] + \Theta(n)$

IH $= 2/n \sum_{q=2}^{n-1} aq \log_2(q) + \Theta(n)$

$= 2/n \cdot a \cdot (1/2 n^2 \log_2(n) - 1/8 n^2) + \Theta(n)$

$= an \log_2(n) - an/4 + \Theta(n)$

we can set $a \geq 4 \cdot c$ in $\Theta(n)$ so that they cancel. So

Hint $E[T(n)] \leq an \log_2 n$

$\leq an \lg n$

$= \Theta(n \lg n)$

6a) Go up the ladder in $\sqrt{n}$ increments. When the first one breaks, drop one by one from the last drop that didn't break.

Worst case runtime will be dropping all $\sqrt{n}$ increments and then all $\sqrt{n}$ runs within the last increment. So $\sqrt{n} + \sqrt{n} = 2\sqrt{n} = O(n)$

b) This is a recursive algorithm of which part a) was one instance of. The base case is when the second to last phone left breaks, here we go up one by one for the size of where the 2nd to last broke to where the last one will break.
For other cases, we will have recursion. We will first go up the ladder in $\sqrt[k]{n}$ increments. When the phone breaks, we will go back to the highest rung it didn't break and go up that block in $\sqrt[k]{\text{block}}$ increments, Repeating the process until we are down to 1 phone. Then we will invoke the base case. The highest safe rung is 1 below where the last phone breaks.
$f(k,n)$ is $O(\sqrt[k]{n})$ This follows the hint as $\sqrt[k]{n} = O(\sqrt[k-1]{n})$
ie cubed roots are bounded by square roots.

7. a) For each attribute, after the first two cards are chosen there is one possible option. If the first two were different, then the third needs to be different. If the first two were the same, the third needs to be the same. We have $\binom{27}{2}$ total pairs that will determine what the last card needs to be, and $\binom{3}{2}$ pairs that account for the first two cards in any 1 match.
So the total number of possible matches is $\binom{27}{2} / \binom{3}{2} = \frac{351}{3} = 117$.
The total number of combinations of cards is $\binom{27}{3} = 2925$
So if we shuffle and turn over three cards, the third will form a match a probability of $\frac{117}{2925}$.

b) The expected number of matches X, for n number of cards is the total number of ~~card~~ triplets * the probability that a single card triplet is a match. So $X = \binom{n}{3}\left(\frac{117}{2925}\right)$
For $n=27$, $X = \binom{27}{3}\left(\frac{117}{2925}\right) = 117$ = total number of possible matches ✓

8) $X_i \begin{cases} 1 \text{ if } i \text{ ends up in own Porsche} \\ 0 \text{ ow} \end{cases}$ $E(X_i) = 1/n$ (selected independently, uniformly at random)

$E(X) = E\left[\sum_{i=1}^{n} X_i\right] \overset{\text{LOE}}{=} \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{n} = n\left(\frac{1}{n}\right) = 1 = E(X)$

Markov: $P(X \geq a) \leq \frac{E(X)}{a}$ for nonnegative random variable X & $a > 0$

$P(X \geq k) \leq \frac{E(X)}{k}$

$P(X \geq k) \leq \frac{1}{k}$

9) The expected number of pairs of points within distance $\theta(\frac{1}{n^2})$ of each other is $E(x) = E\sum_{1\le i<j\le n} x_{ij} \overset{\text{LOE}}{=} \sum_{1\le i<j\le n} E(x_{ij}) = \sum_{1\le i<j\le n} 1 \cdot Pr(i,j \text{ are within distance } \frac{k}{n^2})$

$$= \sum_{1\le i<j\le n} \frac{k}{n^2}$$

$$= \binom{n}{2} \cdot \frac{k}{n^2}$$

$$= \frac{n(n-1)}{2} \cdot \frac{k}{n^2}$$

$$= \frac{n-1}{2n} \cdot k > 1$$

$$k > \frac{2n}{n-1} \Big|^{n=2} \quad k > 4$$

So $E(x) > 1$ for $n \ge 2$ and $k > 4$.

1. BC: n=1, two posts, one red, one black
If the cat starts on the red post it will win, so BC holds
IH: The cat can always win in the case where there are between 2 and 2(n-1) posts
IS: For 2n posts with n red and n black where $n \geq 2$,
Moving in a clockwise direction, there will always be a red post followed by a black post some where in the circle.
Removing this pair, we are left with 2(n-1) posts with n-1 black and n-1 red posts. Using the Inductive Hypothesis, the cat can win in this case. Now adding the pair back in where we took it out, the same starting post will work, because including the pair will not change the red posts visited vs the black posts visited, except to add 1 to the red differential and immediately take it away on the black post of the pair. Thus there is a starting post such that the cat will win for 2n posts.

(if it starts on the right post.)

2) Take two trees that have been labeled and will be the child nodes of an empty string root node. Upon adding them to the root node, an "L" will be added to the front of all the left subtrees nodes, and an "R" will be added to the front of all the right subtrees nodes. So the total number of R's in the tree will be the number of Rs in the Left subtree before it was added, + the number of Rs in the Right subtree before it was added, + an "R" for every node in the right subtree upon adding it as a right child of the new root.

So $f(v) = f(l(v)) + f(r(v)) + s(r(v))$

3) a) $f(n) = 3f(n/2) + \sqrt{n}$ $\quad T(n) = aT(n/b) + f(n)$

so $a=3, b=2, f(n) = \sqrt{n}$

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

$\sqrt{n} = O(n^{\log_2 3 - \epsilon})$

$n^{1/2} \leq c n^{1.585 - \epsilon}$ for $c \geq 1$ & $n > 0$ & $0 < \epsilon \leq \log_2 3 - 1/2$

so $f(n) = O(n^{\log_2 3})$

b) $f(n) = 3f(n/2) + \sqrt{n}$

① $n = 2^i$

$f(2^i) = 3f(2^{i-1}) + 2^{i/2}$

$t(i) = f(2^i)$

$t(i) = 3t(i-1) + 2^{i/2}$

$t(i) = (L-3)(L+\sqrt{2})$ $\quad t(i) = c_0 \sqrt{2}^i + c_1 3^i$

$f(2^i) = t(i)$ $\quad f(n) = c_0 \sqrt{2}^{\log_2 n} + c_1 3^{\log_2 n} = O(3^{\log_2 n}) = O(n^{\log_2 3})$

$2^i = n$

$i = \log_2 n$

so bounds match!

4) a) $f(n) = 2f(n-1) - f(n-2)$, $f(1)=5$, $f(0)=2$

This is a two-term homogenous linear recurrence relation with constant coefficients A & B which are nonzero.

$f(n) = A \cdot f(n-1) + B \cdot f(n-2)$ $\quad A=2, B=-1$

Denote by r, s the roots of $x^2 - Ax - B$

$x^2 - 2x + 1 = (x-1)(x-1)$ $\quad r=s=1$

Because $r=s$, there are constants α, β s.t. $\forall n \geq 0$,

$f(n) = (\alpha + \beta n) \cdot r^n = (\alpha + \beta n) \cdot (1)^n$

$f(0) = 2 = \alpha + \beta(0) \cdot (1)^0 = \alpha$

$f(1) = 5 = (\alpha + \beta(1)) \cdot (1)^1 = 2 + \beta$ $\quad \beta = 3$

$f(n) = 2 + 3n$

$f(2) = 2(5) - (2) = 8 = 2 + 3(2)$ ✓

$f(3) = 2(8) - (5) = 11 = 2 + 3(3)$ ✓

$f(4) = 2(11) - (8) = 14 = 2 + 3(4)$ ✓

Assume $f(n) = 2 + 3n$

BC: $f(0) = 2 + 3(0) = 2$ so base case holds. (and $f(1) = 2 + 3(1) = 5$)

IH: $f(k) = 2 + 3k$

IS: Assuming $f(k)$ holds, show $f(k+1) = 2 + 3(k+1)$

$f(k+1) = 2f(k) - f(k-1)$

Using IH, $f(k) = 2 + 3k$ & $f(k-1) = 2 + 3(k-1)$

$f(k+1) = 2(2+3k) - (2 + 3(k-1))$

$f(k+1) = 4 + 6k - 2 - 3k + 3$

$f(k+1) = 3k + 5 = 2 + 3(k+1)$ ← so $f(n) = 2 + 3n$ is the solution.

Did this before we covered annihilators

$T = \langle T_n \rangle$

$LT = \langle T_{n+1} \rangle$

$L^2T = \langle T_{n+2} \rangle = \langle 2T_{n+1} - T_n \rangle$

$L^2T - 2LT + T = \langle 0 \rangle$

$(L-1)^2$

$f(n) = (c_0 n + c_1)(1^n)$

b) $T(n) = T(n-1) + T(n-2) + O(1)$ ← mult & subtraction; ↑ $f(n-1)$; ↑ $f(n-2)$

Homogenous

$T = \langle T_n \rangle$

$LT = \langle T_{n+1} \rangle$

$L^2T = \langle T_{n+2} \rangle = \langle T_{n+1} + T_n \rangle$

$(L^2 - L - 1)T = \langle 0 \rangle$

$= (L - \phi)(L - \hat{\phi})$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$ →

Nonhomogenous

$O(1) = K$

$T = \langle K \rangle$

$LT = \langle K \rangle$

$LT - T = \langle 0 \rangle$

$= (L-1)T = \langle 0 \rangle$

Annihilator for both:

$= (L - \phi)(L - \hat{\phi})(L-1)$

$\Rightarrow T(n) = c_0 \phi^n + c_1 \hat{\phi}^n + c_2 1^n$

$= O(\phi^n)$

n=2

5)a) BC Silly-Sort(A,1,2), first if statement will swap A[1] and A[2] if A[1]>A[2], if not, no swap happens. The second if statement will return as 1+1=2.

IH: Silly-Sort correctly sorts input array of size ℓ where $1 \leq \ell < n$

The first call to silly sort correctly sorts the first $2/3\,n$ elements using the Inductive Hypothesis. So the elements from 1 to $n/3$ are less than the elements from $\frac{n+1}{3}$ to $2/3\,n$. The second call correctly sorts the $2/3\,n$ elements from $n/3$ to n using the Inductive Hypothesis. So the elements from $\frac{2n+1}{3}$ to n are the largest $n/3$ of the original input array. Now as the middle has been changed from the second call, we need to resort the first $2/3\,n$. The third call correctly sorts the elements from 1 to $2/3\,n$ using the Inductive Hypothesis. Now, the elements from 1 to $n/3$ are less than $n+1/3$ to $2/3\,n$ and both these ranges' elements are less than $2n+1/3$ to n elements. So the array is sorted.

b) $T(n) = 3T(2n/3) + \Theta(1)$

amount of operations at level k: $(2/3)^k n$ lowest level $1 = (2/3)^k n, \frac{1}{(2/3)^k} = n = (3/2)^k$

$k = \log_{3/2} n$ number of leaves $= 3^k$.

worst case runtime $= 3^{\log_{3/2} n} = \left((3/2)^{\log_{3/2} 3}\right)^{\log_{3/2} n} = \left((3/2)^{\log_{3/2} n}\right)^{\log_{3/2} 3} =$

$= n^{\log_{3/2} 3} = \Theta(n^{2.7\ldots})$

c) Insertion-Sort $= \Theta(n^2)$, Merge-Sort $= \Theta(n \lg n)$, Heapsort $= \Theta(n \lg n)$ Quicksort $= \Theta(n^2)$, Silly-Sort $= \Theta(n^{2.7\ldots})$ Sillysort is the worst of the sorting algorithms.

6) a) If the red ball was in bin 1 in round $(n-1)$, it stays with probability $3/4$. If the red ball was in bin 2, it swaps with probability $1/4$. The probability it is already in bin 1 is $p(n-1)$ and because there are two bins, the probability it is in bin 2 is $1-p(n-1)$.

so $p(n) = 3/4(p(n-1)) + 1/4(1-p(n-1))$, $p(0)=1$

b) $p(1) = 3/4(1) + 1/4(1-1) = 3/4 = \frac{48}{64} = \frac{3}{4}$

$p(2) = 3/4(3/4) + 1/4(1-3/4) = 10/16 = \frac{40}{64} = 5/8$

$p(3) = 3/4(10/16) + 1/4(1-10/16) = \frac{30}{64} + \frac{6}{64} = \frac{36}{64} = \frac{9}{16}$

lowest = 3/3
if 1/3 → 3/3

Guess $p(n) = \frac{2^n+1}{2^{n+1}}$

BC: $p(0) = \frac{2^0+1}{2^1} = 1$ so the base case holds. (red ball is in bin 1 to start)

IH: $p(k) = \frac{2^k+1}{2^{k+1}}$ for $k \geq 0$, k is an int

IS: Show that $p(k+1) = \frac{2^{k+1}+1}{2^{k+2}}$

$p(k+1) = 3/4\, p(k) + 1/4(1-p(k))$

$p(k+1) = 3/4 \frac{2^k+1}{2^{k+1}} + 1/4\left(1 - \frac{2^k+1}{2^{k+1}}\right)$ Using Inductive Hypothesis

$= \quad // \quad + 1/4\left(\frac{2^{k+1}-2^k-1}{2^{k+1}}\right)$

$= \frac{3(2^{k+1})+2^{k+1}-2^k-1}{4(2^{k+1})}$

$= \frac{3\cdot 2^k+3+2^{k+1}-2^k-1}{4(2^{k+1})}$ ✓

$= \frac{2\cdot 2^k+2+2^{k+1}}{4(2^{k+1})} = \frac{2(2^{k+1})+2}{4(2^{k+1})} = \frac{2^{k+1}+1}{2(2^{k+1})} = \frac{2^{k+1}+1}{2^{k+2}}$

c) $E(x) = E\left[\sum_{n=1}^{m} X_n\right] \overset{LOE}{=} \sum_{n=1}^{m} E[x] = \sum_{n=1}^{m}\left(\frac{1}{2} + \frac{1}{2^{n+1}}\right)$

$= \frac{m}{2} + \sum_{n=1}^{m} \frac{1}{2^{n+1}}$

$S = \sum_{n=1}^{m} \frac{1}{2^{n+1}} = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots \frac{1}{2^{m+1}}$

Finite geometric series $S_x = \frac{a_1(1-r^x)}{1-r}$ where x is number of terms a_1 is first term and r is the common ratio, $a_1 = 1/4$, $r = 1/2$, $x = m$

$S_m = \frac{1/4(1-1/2^m)}{1-1/2} = 1/2\,(1-1/2^m)$

$E(x) = m/2 + 1/2\,(1 - \frac{1}{2^m})$

7) a) BC: $x=2$ 2 factors into at most $\log 2$ unique primes.
True, 2 factors into 1 prime which is 2.
IH: for $n \geq 2$ where n is an integer >0, any integer x where $2 \leq x \leq n$ factors into at most $\log x$ unique primes.
IS: show that for $x = n+1$, $n+1$ factors into at most $\log(n+1)$ unique primes.
Case 1: If $n+1$ is prime then it factors into 1 prime, and $1 \leq \log(n+1) \vee$ True
Case 2: If $n+1$ is composite, $\exists$ positive integers $i \& j$ s.t. $2 \leq i \leq n \& 2 \leq j \leq n$ & $i \cdot j = n+1$. Using IH, i factors in to $\log i$ unique primes and j factors into $\log j$ unique primes. Most unique primes will result from $i \neq j$, so total number of unique primes $= \log i + \log j = \log ij = \log(n+1)$ so for any positive integer x, x factors into at most $\log x$ unique primes.

b) number of primes less than n is $\Theta \frac{n}{\log n}$
p is chosen uniformly from all primes less than or equal to m, $p = O\left(\frac{1}{\frac{m}{\log m}}\right) = O\left(\frac{\log m}{m}\right)$
x factors into at most $\log x$ unique primes
number of primes less than or equal to m that divide x is $O(\log x)$
so $O(\log x) \cdot O\left(\frac{\log m}{m}\right) =$ number of p's $\cdot$ probability for each one to be chosen
so $Pr(p|x) = O\left(\frac{\log x \log m}{m}\right)$

c) $Pr(x \equiv y \pmod p) = Pr(p | (|x-y|)$ $x \leq n$, $y \leq n$, so $(|x-y|) < n$
$x \neq y$, so $|x-y| = cn$ for some $0 < c < 1$
Therefore $Pr(p|(|x-y|)) = Pr(p|n) = O\left(\frac{\log n \log m}{m}\right)$ from part b where $x = n$.

d) $Pr(x \equiv y \pmod p) = O\left(\frac{\log n \log m}{m}\right)$
for $m = \log^2 n$: $= O\left(\frac{\log n \log(\log^2 n)}{\log^2 n}\right)$
$= O\left(\frac{\log(\log^2 n)}{\log n}\right)$
$\lim_{n \to \infty} \frac{\log(\log^2 n)}{\log n} = \lim_{u \to \infty} \frac{\log u^2}{u}$ where $u = \log x$ $= \lim_{u \to \infty} \frac{2 \log u}{u} = \lim_{u \to \infty} \frac{\frac{d}{du} 2 \log u}{\frac{d}{du} u}$
$= \lim_{u \to \infty} \frac{2}{u} = \lim_{u \to \infty} \frac{2}{u} = 0$, so $O\left(\frac{\log(\log^2 n)}{\log n}\right)$ is "small"

7)e) Alice should pick a random prime p, where $2 \le p \le r$, where $r = \log^2 n$
she should then find $x \bmod p$ for her number x and send at most $\log r +$
$\log(x \bmod r)$ (logging the values is their conversion to binary) bits to Bob.
This will be less than $\log n$ bits as $r = \log^2 n$ and $x \bmod r < r$
so $\log x \bmod r + \log r \le 2 \log(\log^2 n)$ bits $= O(\log(\log^2 n))$ bits
Bob should then find $y \bmod p$ and if it is equal to the $x \bmod p$ Alice
sent along with p, the algorithm says x and y are equal.
The probability of failure is $O(\frac{\log(\log^2 n)}{\log n})$ as calculated in part d
as $m = r = \log^2 n$ in this algorithm.

1) a) if n=1 or n=2, then all inhabitants are knights because we have a strict majority of knights, thus no questions are necessary. The base case, where we have at most 1 knave, and n=3 and thus uncertainty, will be treated by the "case where n is odd" part of our algorithm. The algorithm will pair off as many people as possible (for the base case, 2) and then ask each n-1 (everyone except the person not paired) person what type the odd man out is. By virtue of the knights all telling the truth, they will all vote together, so we know that the response with the most votes will have to include their at least 50% voting block (majority - 1 possibly for the odd man out being a knight). By the same token, if there is a tie, the odd man out must be a knight, as a tie can only occur if the knaves all vote in opposition to the knights and there is 1 knight missing from the voting population, the odd man out. If we identify the odd man out as a knight, we can then ask him t-1 questions, where t is the total number of people on the island, about each person to determine every persons type. Initially t=n, but there are two other cases where we will recurse and in their recursive calls, t>n.

If we identify the odd man out as a knave, we will ask all the other pairs about the other person in the pair. Pairs that answer knave-knave, knave-knight, or knight-knave can have at most 1 knight, so we can remove these pairs and we will still maintain a majority as at worst we will remove 1 of each, at best we will remove 2 knaves. For pairs that answer knight-knight, we know that either both are knights telling the truth, or both are knaves lying. We will remove 1 from each of such pairs, as even though we may remove more knights than knaves, we will have the numbers in these pairs for each type and maintain majority of knights. So this is the worst case, as we can only remove half of each pair instead of the full pair. We will remove the odd man out knave and then recurse on at worst $\frac{n-1}{2}$ people. So for n=3, either the poll will reveal a knight in n-1=2 questions or we will reduce the size by getting knight-knight answers to n-1=2 more questions pairing up the remaining two, and recurse on $\frac{n-1}{2} = \frac{3-1}{2} = 1$ from removing the odd knave out and one of the knights, and because n=1 we know we have a knight.

Continued—>

→ continued: We have shown for $n=3$, we will need 2 questions if the odd man out is a knight or 4 questions if the odd man out is a knave to identify a knight. In general, the worst case for our "case where n is odd" will be when we select a knave as the odd man out, as we will have to recurse after pairing up with $n-1$ questions.
What about cases where n is even? In those cases we will skip the $(n-1)$ questions to poll, as there will be no odd man out. We will remove as many people as possible and recurse, which will be at worst $\frac{n}{2}$, in the case of all pairs answering knight knight. It will take n questions to pair everyone up, as we have an even number. We will keep recursing through these even cases until we have an odd n, or $n=2$ at which point we will select either of the two knights left and ask them t questions to identify everyones type. That is the algorithm, what is its worst case # of questions? Well, odd cases ask almost double the questions as even cases because of the polling. Odd cases always lead to either termination or Even cases if we throw out the odd knave out, and divide by at worst 2 or if better remove some multiples of 2 as well. So the worst case would really alternate between odd and even, but to make it simpler lets say the worst case is bounded by an impossible case where every level is odd. The recurrence for this impossible case is $Q(n) = Q(\frac{n-1}{2}) + (2(n-1))$ To make it simpler lets say that this is bounded by $Q(n) = Q(\frac{n}{2}) + 2n$ as $\frac{n}{2} > \frac{n-1}{2}$ and higher n = more levels = more questions.

$Q(n) \leq 2(n + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots)$

$\leq 2n(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots)$

$\leq 2n(S)$ where $S = (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots)$

$\frac{1}{2}S = (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots)$

$S - \frac{1}{2}S = 1 = \frac{1}{2}S$

$S = 2$

To make it simpler, we are also bounding by $2n$ instead of $2n-2$ as this is just 2 more questions

So the worst case is bounded by $4n$ questions to find a knight, then we ask $n-1$ questions to the knight. Thus the worst case is $< 5n-1$ questions $0 \leq 5n-1 \leq cn^2 \; \forall \; n \geq Max(\frac{5}{c}, 5]$ therefore our algorithm is $o(n^2)$.

than the person answering

11b) We can only ask questions about other people's types. Thus we can only learn from the answers of the inhabitants. We cannot trust the knaves answers so we must identify types by asking a knight. to do this it follows that we must be able to identify a knight.

There are two ways to identify a knight, by polling every one else on the island about them, or by excluding groups by their answers until we know we are left with only knights.

Without a majority of knights, polling can't tell us if someone is a knight, as if we select a knight, we will have a majority of knaves, and because they can vote in either direction they can put a majority behind either response and thus we can't trust a majority. If we select a knave, at least a majority of +1 knights will say knave, but this will be indistinguishable from selecting a knight and having the knaves all vote knave. So in either case, without a majority of knights, we can't trust polling.

for the at most half knights case

When we pair up people and ask them about each others type, the four possible combined responses are knave-knave, knave-knight, knight-knave or knight knight. The first could be said by 2 knaves or a knight-knave pair. The second and third by 2 knaves or a knight-knave pair. The fourth by two knights or 2 knaves. If we have ~~almost~~ half knights, removing any of the four pairs does not guarantee removing more knaves than knights. Also our trick of removing one from pairs that say knight knight does not guarantee removing more knaves than knights. (reduction of)

So there is no way to get a majority of knights from a half knight population and thus there is no way to identify a knight, so the problem is impossible to solve without a knight majority.

2) BC: $n=1$ $=n-1=0$ pairs of parenthesis, True, a single matrix doesn't need parentheses.

IH: Any full parenthesization of n matrices has exactly $n-1$ pairs of parenthesis

IS: For $n+1$, $A_1 \dots A_{n+1} = (BC)$ where $B = A_1 \dots A_k$ & $C = A_{k+1} \dots A_{n+1}$ for some k

Using IH, B has $k-1$ pairs of parentheses, C has $((n+1)-k)-1 = (n+1)-k-1$

BC = B parentheses + C parentheses $= (n+1)-k-1+k-1 = (n+1)-2$, (BC) = BC parentheses $+1$

$(BC) = (n+1)-2+1 = (n+1)-1$ So for $(n+1)$ full parenthesization has $(n+1)-1$ matrices.

3) $4^{2}A_1 \cdot 2^{5}A_2 \cdot 5^{1}A_3 \cdot 1^{3}A_4$

$m[1,2] = 4 \cdot 2 \cdot 5 = 40$

$m[2,3] = 2 \cdot 5 \cdot 1 = 10$

$m[3,4] = 5 \cdot 1 \cdot 3 = 15$

$m[1,3] = \min\{$

$m[1,1] + m[2,3] + 4 \cdot 2 \cdot 1 = (50)$

$m[1,2] + m[3,3] + 4 \cdot 5 \cdot 1 = 60\}$

$m[2,4] = \min\{$

$m[2,2] + m[3,4] + 2 \cdot 5 \cdot 3 = 45$

$m[2,3] + m[4,4] + 2 \cdot 1 \cdot 3 = (16)\}$

$m[1,4] = \min\{$

$m[1,1]^{0} + m[2,4]^{16} + 4 \cdot 2^{24} \cdot 3 = (40)$

$m[1,2]^{40} + m[3,4]^{15} + 4 \cdot 5^{60} \cdot 3 = 115$

$m[1,3]^{50} + m[4,4]^{0} + 4 \cdot 1^{12} \cdot 3 = 62\}$

$s[1,4] = (A_1 (m[2,4]))$

$s[2,4] = (A_1 ((m[2,3])(A_4)))$

$= (A_1 ((A_2 A_3) A_4))$

cost is 40

m	j=1	2	3	4
i=1	0	40	50	40
2		0	10	16
3			0	15
4				0

s	j=1	2	3	4
1	-	1	1	1
2	-	-	2	3
3	-	-	-	3
4	-	-	-	-

4. a) $m(0) = 0$

$m(x) = \inf$ if $x < 0$

$m(x) = \min(m(x-x_1), m(x-x_2), m(x-x_3)) + 1$

↑ all the possibilities recursing on 3 different boxes

↑ non homogenous, filling 1 box at each level

"m" ↓

b) Fill a 1d array of size n with the values of $m(x)$ starting with 0. so $m(0) = 0$ so $m[0] = 0$, $m[1] = m[x<0]$ for all three cases as x_1, x_2, x_3 are all > 1 in the example, so $m[1] = \inf$, this continues until $m[4]$ which for $m[4-x_1] + 1 = m[0] + 1 = 1$. The other two options are infinity so we put 1 in $m[4]$. We can store a quadruple with the box values and increment x_1 here if we want to keep the combination.

This takes $3n$ time, 3 look ups for each min we take as we go up the array n times. So $O(n)$, which is polynomial time. The running time is $n *$ the number of different sized boxes we are looking at.

5(a) if $i > 10$, $r(i) = \max(b_i + r(i-10), r(i-1))$
if $i > 1$ & $i \leq 10$, $r(i) = \max(b_i, r(i-1))$
$r(1) = b_1$

We start with b_1 on day 1, then for the next 9 days we only update the max revenue if that day's is greater than the day before's otherwise we keep the day before's value. After 10 days we only update if (that day's revenue + the max 10 days before) is greater than the previous day's, otherwise we keep the previous day's value.

b) We initialize an array r of size n, indices will be 1 less than in the recurrence

```
r[0] is set to b[0]
for i = 1 to i = min(9, n-1)      // incase (n-1) < 9
    r[i] = max(b[i], r[i-1])      // set the first 10 values
if (n-1) > 9 { for i = 10 to n-1
    r[i] = max(b[i] + r[i-10], r[i-1]) }
```

answer is in r[n-1]

We iterate through an array of size n and perform operations at each index that are absorbed into the coefficient. $\Theta(n)$

i starts at 1, j starts at 0

6) a) if $i > 1$ and $j \geq w_i$: $r[i,j] = \max(r(i-1,j), b_i + r(i-1, j-w_i))$
if $i > 1$ & $i \leq 10$ and $j \geq w_i$: $r[i,j] = \max(r(i-1,j), b_i)$
if $i > 1$ & $j < w_i$: $r(i,j) = r(i-1,j)$

$r(1,j)$ $\begin{cases} = b_1 \text{ if } j \geq w_1 \\ = 0 \text{ if } j < w_1 \end{cases}$

indices are all day-1

i starts at 0, j starts at 0

b) Initialize a 2d array of size n and W - would be W+1 because we're starting at 0, but we want the value of $j = W-1$ so we have W elements on the j side, including 0.

```
for j = 0 to W-1:
    r[0,j] = (j >= w[0]) ? b[0] : 0      Filling base column to allow for r[i-1,j] calls
for i = 1 to n-1:
    for j = 0 to W-1:
        if j < w[i]:                      if j < work required that day, max revenue
            r[i,j] = r[i-1, j]            is same as previous days for that j.
        else if i < 10:
            r[i,j] = max(r[i-1,j], b[i])  Same as 5
        else:
            r[i,j] = max(r[i-1,j], b[i] + r[i-10, j-w[i]])   Similar to 5,
                                                              subtract work as well to see what max value is for combined work
answer is in r[n-1][W-1]
```

we iterate through an iterate of size $n \cdot W$ and perform operations at index that are absorbed into the coefficient, so $O(nW)$.

7. We have the recurrence relation
$r(i,k)$ where i is the starting index of the first chunk and k is the number of parts the bar will be divided into

$r(i,k) = 0 \quad \forall k > n$

$r(i,k) = \text{sum}(V_i, \ldots, V_n) \quad k = 1$

$r(i,k) = \max\{\min((\text{sum } V_i \ldots V_{i+j}), r(i+j, k-1))\} \; \forall i,j = [1 \ldots n], i < j, i+j \leq n$
$\uparrow k > 1 \; \& \; k \leq n$

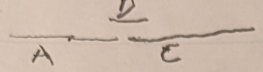
iterate from 1 to $k \rightarrow n$ bounded by n

nested loop iterate from 0 to $n-1$ for $i \rightarrow n$ " "

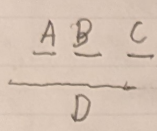
iterate from i to $n-1$ for $j \rightarrow \frac{n}{n^2} *$ " "

to fill a table with memoization values
for all calls of $r(i,k)$ for $i = [1 \ldots n]$ & $k = [1 \ldots k]$
answer is in $\text{memo}[i,k]$

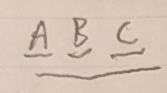
$O(n^3)$ is runtime

1. a) 

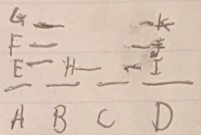
optimal is A, C, this would take B so disproven

b) 

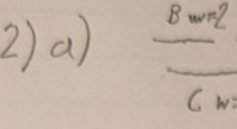
optimal is A, B, C, But D is chosen because it starts first, so disproven

c) 

optimal is A, B, C But D is chosen because it ends latest (and all others conflict), so disproven

d) 

optimal is A, B, C, D (4) But this would take H and A/E/F/G and D/I/J/K (3) so disproven

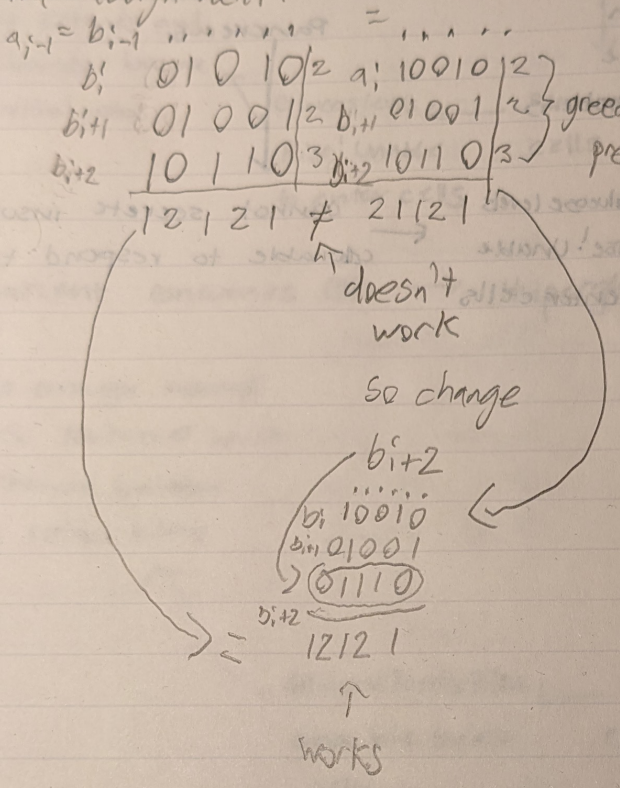
2) a) 

The greedy algorithm in class would select B, A because its optimized for earliest end time, but C has a greater combined weight.

b) Sort the activities by finish times in increasing order.
This takes $n \log n$ time
Create a $1 \times n$ array, m, set $m[0]$ to 0.
for each index up to n, find x_j by using binary search to find the max index whose endtime $\leq$ start time of the index were on. This takes $\log n$ time $\cdot n$ indices $= n \log n$.
Also at each index, compute $m[j] = \max(m[j-1], w[j] + m[x_j])$
The max weight will be in $m[n]$
The schedule can be made by looking at $m[n]$ and deciding if $m[j]$ used $w[j]$ or not based on $m[j-1]$ and $m[x_j]$, and recursing. Both the weight and schedule are $n \cdot$ constant time ops.

So the algorithm runs in $O(n \log n)$.

3. Want to construct the ith row of A using greedy.
Is will be placed where there is max $(c_j - a_j)$
Replace this row in th ith row of B which is another valid assignment.



∴ this process can be repeated for all rows in A into corresponding rows for B with some other row change, so the algorithm is correct

4. After an expansion, the table will be $\frac{3}{4} \cdot 2 = \frac{3}{8}$ full.
After a contraction, the table will be $\frac{1}{4} \cdot 2 = \frac{1}{2}$ full

- Case 1: we just delete: We have $\frac{1}{8}$ operations to pay for $\frac{1}{8}$ deletions and $\frac{2}{8}$ reinsertions, we need 3 credits taxed each deletion.
- Case 2: we only insert, we have $\frac{3}{8}$ operations to pay for $\frac{3}{8}$ insertions and $\frac{6}{8}$ reinsertions, we need 3 credits taxed each insertion.
- Case 3: we just delete: we have $\frac{1}{4}$ operations to pay for $\frac{1}{4}$ deletes and $\frac{1}{4}$ reinsertions, we need 2 credits taxed each deletion.
- Case 4: we just insert: We have $\frac{1}{4}$ operations to pay for $\frac{1}{4}$ insertions and $\frac{3}{4}$ reinsertions, we need 4 credits taxed each insertion.

Case 5: We have some deletes and some insertions that cancel each other out before heading to another expansion or contraction. 1 of the other four cases will be enough, the canceled out ops will provide a surplus on top of these.

So if we take the max of the deletion cases and insertion cases, we have amortized cost of 3 for deletion and 4 for insertion, so each is $O(1)$.

5. a) For every power of 2, we will need a surplus in the operations between the last power of 2 and it to pay for it. the previous power of 2 will be ½ the next one so we will have ½ operations to pay for the ith one in their surplus. so if we have a surplus of 2, $2 \cdot \frac{1}{2}$ ops will pay for the ith operation. We also need to pay for each operation on its own with a cost of 1, so 1+2 = 3 amortized cost. $\Theta(1)$

b) Again we will be using the operations in between powers of 2 to pay for their larger costs. We will be using some constant times n, now that the powers are n^2. for a power of 2 its value will be n^2. We again have $n/2$ operations to pool money for it. Each one will be $\geq n/2$, as that is the smallest index in the $n/2$ range from one power of 2 to the other. so $n/2 \cdot n/2 = n^2/4$ so $4n$ is enough to pay for the powers of 2. We also need to pay for the operations costing 1 so $4n+1$ amortized cost should be enough. $\Theta(n)$

6. We will use a 1xn array A, to store the 1s and 0s and a Union find data structure to store the indices of groupings of contiguous 1's. We SetToOne(i) by checking that A[i] = 0, and if it is, we set it to 1. Additionally in that case, we check its neigbors to the right and left if they exist, and if they are 1's, we call FindSet on their index and Union on their sets parent with i or i's set if it was already merged with the neighbor. We can make the parent of our sets to always be the highest index. For GetValue(i) we simply access the array at A[i]. For GetClosestRightZero(i), we simply call FindSet(i) and add 1, as it will return the highest index in a group of contiguous 1's, so adding 1 will be the index of the closest right zero. The first two operations are just array calls and comparisons, so they run in $\Theta(1)$ time. Get closest right zero is just a call to Find set on i, and a constant addition. So it's $\log^* n$.

we call make set(i) and,

1.) In the worst case, there are n MakeSet Union and FindSet Operations. There are n MakeSet operations, at most $n-1$ Union operations, which leaves $m-n-(n-1)$ FindSet operations. In the worst case, we only call union when sets are the same size, forcing the rank to grow by 1, whenever possible. Eg call union on 2 sets of rank 0 to get rank 1, 2 sets of rank 1 to get rank 2 with 4 nodes, 2 sets of rank 2 to get rank 3 with 8 nodes and so on. So Findset in the worst case is $\log(n)$ where n is the number of nodes/elements in the set, as it will have to traverse every level of the rank in the worst case to find the leader.

We assume the Findset operations in Union to get the leader are being counted in the $m-n-(n-1)$ Findset operations. So we have n Makesets, costing $\Theta(1)$ time + n Unions costing $\Theta(1)$ time + $(m-n-(n-1))$ Findsets costing $\log(n)$ time. n dominates the others in the Findset, and Findset dominates all 3 operations for $\Theta(m\log(n))$ cost.

If we count union's findsets differently, we have an additional $(2n-1)\cdot\log(n)$ cost because there are potentially two findsets for every union, but n still dominates $2n-1$, so we would still have $\Theta(m\log(n))$.

# of elements in array

64 32 16 8 4 2 1

2.) 0 0 1 0 0 1 1

6 5 4 3 2 1 0 ← i

a) In the worst case we search every array
i ranges from 0 to log(n) = # of arrays
each array is length 2^i and is sorted,
so can be searched in $O(i)$ time $= O(\lg(n))$ = time for each one

so search $= O(\log(n)\log(n)) = O(\log^2(n))$

b) If we merge 2 arrays of size 2^i each, we will need a resulting 2^{i+1} array to hold the elements.
If both arrays $A_0, A_1, \ldots A_{k-2}$ are full, then the time to fill the new $A_{k-1} = 2(2^0+2^1+\ldots 2^{k-2})$

$= 2(2^{k-1}-1)$

$= 2^k - 2 = \Theta(n)$ worst case

total cost for an insert $= 2^{k+1}$

$* \text{"k digits"} = O(k \cdot 2^k)$

$= O(n \log n)$

$\div$ n operations

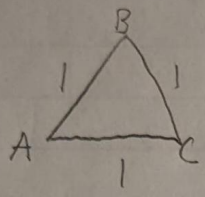
$O(\log(n))$ amortized

c) Unless all other elements are marked for deletion in the array, simply mark the element for deletion. Then if you have to insert, later, insert into a space marked for deletion. If all elements are marked for deletion, drop the array and reduce k by 1.

3) a) $$m(i,j) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i > j \\ m(i+1, j-1) + 2 \text{ if } A[i] = A[j] \\ \max(m(i+1,j), m(i,j-1)) \text{ otherwise} \end{cases}$$

b) $$m(i,j) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i > j \\ m(i+1, j-1) + 2 \text{ if } A[i] = A[j] \\ \min(m(i+1,j), m(i,j-1)) + 2 \text{ otherwise} \end{cases}$$

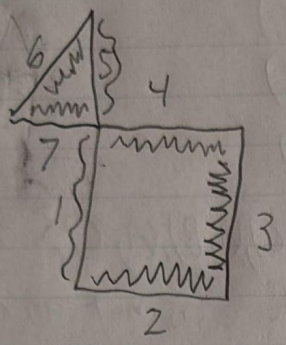
4)



for cut($\{A,B\},\{C\}$) $(u,v) = (B,C)$
for cut($\{A,C\},\{B\}$) $(u,v) = (A,B)$
for cut($\{B,C\},\{A\}$) $(u,v) = (A,C)$

set of edges $\{(B,C),(A,B),(A,C)\}$ forms a cycle, and therefore is not an MST.

5.



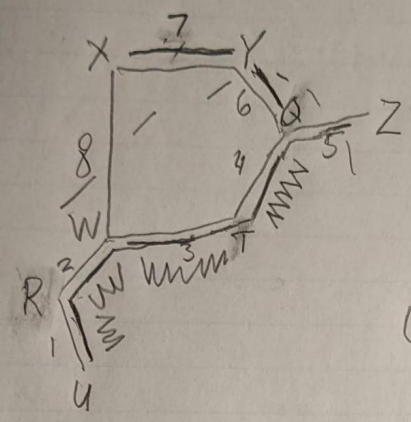
~~~ minimum cycle cover

wwww maximum spanning tree

Apply Kruskal's Algorithm, except instead of always considering whether to include the minimum edge, do it for the maximum. The complement of this maximum spanning tree is the minimum cycle cover.

The runtime for Kruskals is $O(m \log n)$
runtime to find complement is bounded by m
so $m \log n + m = O(m \log n)$
~~~

6.



— MST

$A = \{Q, T, W, R, U\} = S$

$--- (S, V-S) = (\{Q, T, W, R, U\}, \{Z, Y, X\})$

$w(AU(Y,Q)) = 16$

↑ safe edge

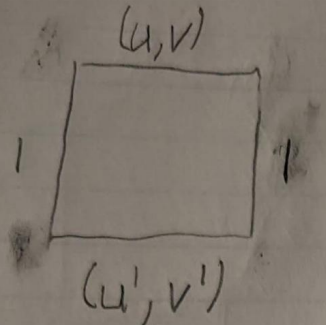
$w(AU(Q,Z)) = 15$

↑ safe edge & light edge

(Y,Q) is a safe edge for A that crosses $(S, V-S)$, yet it is not a light edge for the cut.

So, False!

7.



exchange (u,v) for (u',v')

if (u,v) is a light edge crossing then exchanging (u',v') for (u,v) will increase the weight of MST; unless (u,v) and $(u'v')$ have the same weight in which case there are multiple MSTs, at least one including each edge.

If exchanging the two, lowers the weight of the MST, then (u,v) couldn't have been in an MST. $\therefore w(u,v) \leq w(u',v')$ so a cut across (u,v) and (u',v') will have (u,v) as a light edge crossing.

1) BC: For $n=1$ node, there are no other nodes to be connected, $n-1 = 1-1$ edges = 0 edges. So the Base case holds.
IH: for any tree with $k<n$ nodes, the number of edges is $k-1$.
IS: Let $n = k+1$ nodes
IH: $n = (k-1)+1$ edges
$n = k$ edges
So a tree with n nodes, has $n-1$ edges and the IS holds.

2) a)
```
Viterbi(G, v0, s){
  if s.isempty, return v0;
  for each outgoing edge (v0,v1) in G{
    if label on edge (v0,v1) == σi{
      path = Viterbi(G, v1, s.pop(σi));
      if path ≠ "NO-SUCH-PATH" return v0 ++ path;
    }
  }
  return "NO-SUCH-PATH";
}
```

The recursive calls in worst case will visit |V| vertices * k suffixes. Each outgoing edge might go to |V| vertices. So the runtime is $O(k|V|^2)$.

b)
```
ProbablePath(G, v0, s){
  initialize probpath.path = No Such Path and probpath.prob = 0
  if s.isempty, return v0;
  for each outgoing edge (v0,v1) in G{
    if label on edge (v0,v1) == σi{
      remPath = ProbablePath(G, v1, s.pop(σi));
      if prob(v0,v1) * remPath.prob ≥ probpath.prob{
        probpath.prob = prob(v0,v1) * remPath.prob and probpath.path = v0 ++ remPath.path
      }
    }
  }
  return probpath;
}
```

We are just adding constant time mults and comparisons to the worst case, so the runtime is still $O(k|V|^2)$.

3) BC: When the number of edges in the predecessor chain from s to v is zero, s=v and thus dist(v)=0 (dist of s to itself). So BC holds

IH: if dist(x) ≠ ∞ then dist(x) = w(predecessor chain: s → ... → pred(pred(x)) → pred(x) → x) for all k<ℓ where k is the length of the predecessor chain of x and ℓ is the number of edges.

IS: dist(v) = dist(pred(v)) + w(pred(v) → v)

(dist(pred(v)): ℓ-1 edges; w(pred(v) → v): 1 edge)

IH: dist(pred(v)) = w(s → ... → pred(pred(v)) → pred(v)) (ℓ-1 edges)

dist(v) = w(s → ... → pred(pred(v)) → pred(v) → v) (ℓ-1+1 = ℓ edges)

Thus we have proven the claim.

4) BC: for $l=0$, $s=v$, $dist(v) \leq w(s \rightsquigarrow v) = 0$ so BC holds.
IH: If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for any path $s \rightsquigarrow v$ with k edges, where $k \geq 0$
IS: Consider a path $s \rightsquigarrow v$ with $k+1$ edges
$s \rightsquigarrow v = s = u_0 \rightarrow u_1 \rightarrow \ldots u_k \rightarrow u_{k+1} = v$
IH $dist(u_k) \leq w(s \rightsquigarrow u_k)$
Adding in the edge from u_k to u_{k+1} through relaxation:
$dist(u_{k+1}) \leq dist(u_k) + w(u_k, u_{k+1})$
$\rightarrow dist(u_{k+1}) \leq w(s \rightsquigarrow u_k) + w(u_k, u_{k+1})$
So $dist(u_{k+1}=v) \leq w(s \rightsquigarrow u_{k+1}=v)$ holds for $k+1$ edges
Thus claim 2 holds for any path $s \rightsquigarrow v$ and when the algorithm halts (all edges are relaxed) $dist(v) \leq w(s \rightsquigarrow v)$

a)

5.) Take the $\log_{base>1}$ of c_i and make these the edges in a directed graph with the countries as the vertices. Multiply them all by -1. Then run Bellman Ford, if it halts before $|V|$th phase there is no Arbitrage. If it runs past $|V|$th phase, there is a negative cycle and therefore Arbitrage. The transformation runs in constant time $*|E|$ so this is absorbed in Bellman Ford's $O(|V||E|)$ = total runtime

Negative cycle: $-\log(R[i_1,i_2]) - \ldots - \log(R[i_{k-1},i_k]) - \log(R[i_k,i_1]) < 0$

negated: $\log(R[i_1,i_2]) + \ldots \log(R[i_{k-1},i_k]) + \log(R[i_k,i_1]) > 0$

delog: $R[i_1,i_2] * \ldots * R[i_{k-1},i_k] * R[i_k,i_1] > 1$

b) We build the graph from a) the same way. When running Bellman Ford and relaxing edges, keep the the parent of each vertex stored in an array. If BF halts before the $|V|$th phase, print "no sequence exists". If BF runs the $|V|$th phase, compare the edges from the $|V-1|$th phase, and take a vertex from any edge that is relaxed. The cycle will be it and its parents ending in it. Keeping track of the edges is constant time $*|E|$, The cycle is at most $|V|$ to traverse. Neither is greater than BF's $O(|V||E|)$ runtime

6. Go through all the edges and replace them with log(probability) where the base > 1, and then multiply by -1. Then run Dijkstras.

Converting to log space allows us to to sum the edges to get the weight of a path as $\log(ab) = \log(a) + \log(b)$. As the probabilities are $0 < p \leq 1$, the weight in log space is negative, with the lowest probability being the most negative. So multiplying by -1, the lowest probability becomes the highest edge weight. Thus the lowest weight path will correspond to the highest probability. Thus we can use Dijkstras, as probabilities can't be negative.

To run the conversion we must go through all the edges * constant time for log and $*(-1)$. $= O(|E|)$

Dijkstras runs in $O(|E| + |V| \log |V|)$, so the $O(|E|)$ is absorbed into that. So the runtime is $O(|E| + |V| \log |V|)$.

7. We can reduce the Graph Coloring problem to the Radio Towers problem to show the latter is NP-Hard.

Assume we have a graph $G = (V, E)$ to color using k colors

To transform a Radio towers problem:

S will contain the towers mapped to the vertices in G.

$S = \{v \mid v \in V\}$

T will contain the subsets of towers mapped to the edges in G. Every possible pair of towers in a subset of T will be an edge, this is done for all subsets in T, if they already haven't been mapped.

$T = \{\{u, v\} \mid \{u, v\} \in E\}$ (original T will be reconstructed from this T)

k will contain the number of frequencies mapped to the number of colors G will be colored with.

1: If a valid coloring of G exists, assigning the towers frequencies corresponding to its color makes sure that towers in connected vertices (edges), (and therefore within one of the subsets of T together), will have different frequencies, since connected vertices can't have the same color.

2: If there is a valid assignment of frequencies to towers, we can map the frequency's color to the towers vertex, ensuring that connected vertices have different colors, forming a valid coloring of G.

Transforming S depends on $|V|$ in G so it takes $O(|V|)$ time.

Transforming T depends on $|E|$ in G so it takes $O(E)$ time

K in radio towers in # of colors in G, so it does not depend on input size. $O(1)$ time.

So transformation takes $O(|V|+|E|+1) = O(|V|$ time

= polynomial → efficient.

so BC holds. ✓

1. BC: $n=1$ has degree 0, needs 1 color for the only node,
IH: Any graph with $<n$ nodes with max degree 3 can be colored with at most 4 colors.
IS: Take a node v with degree 3 From a graph G with n nodes and maximum degree 3, and remove it. This creates a graph G' with $n-1$. Using the IH, we can color the graph with at most 4 colors. Now add v back in to G' to get G. Because its 3 neighbors will have at most 3 different colors, we can use the 4th color on v.
$\therefore$ Any graph with maximum degree 3 can be colored with at most 4 colors.

2. Subgraph Isomorphism can be checked by seeing if each edge in G1 is present in G2 and whether the vertices in G1 are mapped to corresponding vertices in G2. The vertices can take $O(|V|^2)$ time and edges $O(|E|)$ time, so the checking is polynomial time.

To show that Subgraph Isomorphism is NP-Complete we will reduce it from the clique problem. To find a clique of size k, make G1 a graph with k vertices, with each one connected to the rest (a k-size clique). Set G2 to be the graph we are trying to find the clique in (G). If theres a subgraph isomorphism between G1 and G2, it implies that there is a mapping of vertices from G1 to G, and there is thus a clique of size k in G. Going the other way, if there's a clique of size k in G, it's implied that there is a subgraph with k vertices in G, completely connected. So if we have a solution to subgraph Isomorphism we also have a solution to clique. So Subgraph Isomorphism is NPC.

3. Reduction of Vertex Cover ≤ Subset Weights: Set S as the set of vertices in G, set T as the set of edges in G, where each edge $e=(u,v)$ is transformed to a subset containing its two vertices $\{u,v\}$, Set the weight of each vertex in S to be 1 and finally set $W=k$. This take $O(|V|+|E|)$ time = polynomial

If there is a vertex cover of size k in G, this means there are k vertices that cover all edges in G. This means those vertices will all be in the individual subsets in T and their weight will be no more than W.

Conversely, if there exists a subset of vertices of total weight at most $W=k$, that are all in the individual subsets in T, these vertices will correspond to vertices in G that cover all edges, forming a vertex cover of size k.

∴ there is a polynomial time reduction from Vertex cover to Subset Weights. As Vertex Cover is NP-Hard, this implies Subset Weights is NP-Hard too.

4. EX

$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$

(m_1, m_2, m_3)

$$A = \begin{bmatrix} -1 & -1 & 1 & 0 \\ 0 & 1 & -1 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix} \quad (m \times n) \qquad x = \begin{bmatrix} = \\ = \\ = \end{bmatrix} \quad (n \times 1) \qquad \leq \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad (m \times 1)$$

$1+1=0$, $2-1=1$, $1-1=0$

sum of non negatives

at least 1 literal satisfied in clause

x is our certificate. To check if $Ax=b$, we have to multiply A and x and check the result against b. There are mn entries in A and each one is multiplied by an entry in b, so this takes $O(mn)$ time. The additions take $O(mn)$ time and then we are left with comparisons of two $m \times 1$ vectors which take $O(m)$ time. So the checking is $O(mn)$ = polynomial time.

Reduction of 3CNF $\leq$ 0-1 Integer Programming. Each column in A represents a literal, and each row represents a clause. Within each clause for each row, set the value to be -1 if the the literal appears with no negation, 0 if the literal does not appear, and 1 if the literal appears with negation. We are inverting the clauses because b is an upper bound, we want clauses with more true values to be further within the bound. Set b to be the summation of all non-negative values in the corresponding row of A and subtract 1. We subtract 1 to make sure at least 1 literal evaluates to true in the clause. If all literals in the clause evaluate to false, Ax will have at least 1 value greater than its corresponding value in b, and thus the assignment given by x will evaluate to false. The order left-right of the literals is the order of the truth values top to bottom in X. In x, 0 is false and 1 is true.

If the 3CNF formula is satisfiable, the satisfying assignment with literals ordered top to bottom and true is 1 and false is 0 for x will satisfy the transformed $Ax=b$.

Conversely if there is an x that satisfies the 0-1 integer programming problem, we may convert the corresponding 0s to false assignments for the corresponding literals as well as 1s for true.

So if we have a solution for 0-1 Integer Programming, we also have a solution to 3CNF. So 0-1 Integer Programming is NP-Complete.

5. a) Opponent: rock scissors paper

$W = \frac{1}{3}(0p_1 - 10p_2 + 1p_3) + \frac{1}{3}(10p_1 + 0p_2 - 1p_3) + \frac{1}{3}(-1p_1 + 1p_2 + 0p_3)$

$= \frac{1}{3}(9p_1 - 9p_2 + 0p_3)$

$= 3p_1 - 3p_2$

Objective function: Maximize $3p_1 - 3p_2$

Subject to: $p_1 + p_2 + p_3 = 1$ and $0 \le p_1, p_2, p_3 \le 1$

This means we would make $p_1 = 1$ to maximize W.

b) Objective function: Maximize W

Subject to: $0 \le p_1, p_2, p_3 \le 1$ and $p_1 + p_2 + p_3 = 1$

$W \le p_3 - 10p_2$ Rock

$W \le 10p_1 - p_3$ Scissors

$W \le p_2 - p_1$ Paper

We would make $p_3 = \frac{10}{12}$, $p_1 = \frac{1}{12}$ and $p_2 = \frac{1}{12}$

Given that the opponent will choose 1 with $p = 1$,

opp $p_1 = 1$ would result in $\frac{1}{12}$ smashes $= \frac{10}{12}$, $\frac{-10}{12}$ losses to paper and $\frac{1}{12}$ ties (0) so $W = 0$

opp $p_2 = 1$ would result in $-\frac{1}{12}$ smashes against $= -\frac{10}{12}$, $\frac{10}{12}$ wins over paper and $\frac{1}{12}$ ties (0) so $W = 0$

opp $p_3 = 1$ would result in $\frac{1}{12}$ wins over rock, $\frac{10}{12}$ ties (0), and $-\frac{1}{12}$ losses to scissors so $W = 0$

6. a) we minimize $(x_i - v_i)^2$

b) $\frac{df_i(x_i)}{dx_i} = \nabla f_i(x_i) = 2(x_i - v_i)$

c) for k, The space [0,1] is convex, meaning its second derivative is always non-negative, and for D, the diameter is 1, the distance between endpoints of [0,1]

d) $G = \max_{x,v \in k} |\nabla f_i(x_i)| = \max_{x,v \in k} |2(x_i - v_i)| = |2(1-0)| = 2$

e) $\sum_{i=1}^{T} f_i(x_i) \leq \sum_{i=1}^{T} f_i(x^*) + GD\sqrt{T}$

$= OPT + 2\sqrt{T}$

f) the negative sign maximizes the squared difference which doesn't help us minimize the cost function.

g) We take the -log of the squared difference

$f_i(x_i) = -\frac{1}{4}\log((x_i - v_i)^2)$