Theory of Computations,
Final Exam for the course
CS 5315, Spring 2017

Name: ______________________________

You can use up to 5 handwritten pages of notes. Feel free to use extra sheets of paper if needed, but if you do, do not forget to put your name on each of these sheets.

1. *Primitive recursive and mu-recursive functions:*

10/10 1a. Why do we need to study primitive-recursive and mu-recursive functions in the first place? What programming concepts do they correspond to?

1b. Translate, step-by-step, the following double loop into a mu-recursive expression:

8/10
```
int s = 2;
while(s < a){
  for(int i = 1; i <= b; i++)
    {s = s + i;}
}
```

You can use the multiplication function is this description. What is the result of this program when a = b = 3?

10/10 1c. Prove, from scratch, that the remainder function a % b is primitive recursive. Provide a simpler proof that this function is mu-recursive.

10/10 1d. Is Kolmorogov complexity primitive recursive? mu-recursive? Explain your answers.

1 a) ⊙ We need to study primitive recursive functions in the first place because they are the important building blocks on the way to a full formalization of computability. Also, important in proof theory.

⊙ mu-recursive f's → computable in intuitive sense.
→ only functions computed by Turing Machine

⊙ primitive recursive → for-loops.

⊙ mu-recursive → while loops.

b)
```
int s = 2;
while (s<a) {
  for(int i = 1; i <= b; i++)
  {s = s+i; }
}
```

→ For-Loop
```
int s = 2;
for (int j = 1; j <= c; j++) {
  for (int i = 1; i <= b; i++) {
    s = s+i; }
}
```

Inner Loop: $G(s, b)$.

$s'(s_0, 0) = s_0$

$s'(s_0, m+1) = s'(s_0, m) + (m+1)$

$\therefore f(n_1, 0) = n_1;\ g(n_1) \to g = \pi^1_1$.

$f(n_1, m+1) = f(n_1, m) + (m+1)$

$= f(n_1, m) + (m+1)$

$\to h(n_1, m, f(n_1, m))$

$= \pi^3_3 + (\pi^3_2 + 1)$

$= \pi^3_3 + \sigma(\pi^3_2) = add(\pi^3_3, \sigma(\pi^3_2))$

$G(s, b) = PR(\pi^1_1, add(\pi^3_3, \sigma(\pi^3_2)))$

Outer loop: int s = 2; for(int j = 1; j<=c; j++) { s = G(s,b); }

$S(b,0) = 2;\ S(b, m+1) = G(S(b,m), b)$.

$f(n_1, 0) = 2 \to g(n_1) = 1+1 = \sigma(0) + \sigma(0) = \sigma(\sigma(0))$.

$f(n_1, m+1) = G(f(n_1, m), n_1) \to h(n_1, m, f(n_1, m)) = G(\pi^3_3, \pi^3_1)$

$\therefore f \equiv PR(\sigma(\sigma(0)), G(\pi^3_3, \pi^3_1))$ where $G \equiv PR(\pi^1_1, add(\pi^3_3, \sigma(\pi^3_2)))$. --- (I)

(#) Conversion to μ-recursive:

we stop at smallest 'm' for which $\mu m(!s < a)$

$\to \mu m(!f(m) < a)$.

where $f(m)$ is given by (I)

So what is the final expression?

$f(n_1, \mu m. !f(n_1, m) < a$

When:

a = b = 3;

```
int s = 2
  while (2<3){
  for(int i = 1; i <= 3; i++)
      {s = s + i}.
  }
```

i = 1;
s = 2 + 1 = 3
i = 2
s = 3 + 2 = 5
i = 3
s = 5 + 3 = 8

i = 4 out of for loop
S = 8 now
s > 3.
out of while loop.

S = 8 → Ans.

1 d) No. Kolmogorov complexity is not primitive recursive

No. Kolmogorov complexity is not mu-recursive.

Since, Kolmogorov complexity is <u>not computable</u>, It is neither primitive recursive nor it is mu-recursive.

1 c) Proof:

<u>Remainder</u>: <u>a%b</u> is <u>P.R.</u>

a%b ≡ rem(a, 0) = 0;
rem(a, b+1) = { rem(a, b) + 1 when rem(a,b)+1 < a; 0 otherwise

i.e. if rem(a,b) + 1 < a
then rem(a,b) + 1
else 0. → 0 is P.R. by definition

→ <u>Need to prove</u>:
i) If-then-else is P.R.
ii) Less than is P.R.
iii) add is P.R.

To prove: (i) If - then - else is P.R.:

If $Q(\bar{u})$ then $g(\bar{u})$ else $h(\bar{u})$ $\Rightarrow$ $Q(\bar{u})$ AND $g(\bar{u})$ OR $(1 - Q(\bar{u})))$ AND $h(\bar{u})$

AND is multiplication which is P.R.

OR is P.R. proved below

sub is P.R. proved below

OR is P.R. →

$a \vee b = 1 - (1-a) * (1-b)$
$= a + b - a * b$

→ add and mult is P.R. — proved below.

AND is multiplication which is P.R. proved below.

add is P.R.:

$add(a, 0) = 0$
$add(a, i+1) = add(a, i) + 1$
$f(n_1, 0) = g(n_1) = g = n_1 = \pi^1_1$
$f(n_1, m+1) = h(n_1, m, f(n_1, m))$
$= f(n_1, m) + 1$
$\rightarrow h = \pi^3_3 + 1 = \sigma(\pi^3_3)$.
$\therefore add = PR(\pi^1_1, \sigma \cdot \pi^3_3)$

sub is P.R. →

$(a - b) = a - 1 - 1 \ldots - 1$
$\rightarrow sub(a, 0) = a;\ sub(a, m+1) = prev(sub(a, m))$

prev is P.R. →

$prev(n) = \begin{cases} n-1 & \text{if } n \geq 1 \\ 0 & \text{otherwise} \end{cases}$

$prev(0) = 0 \rightarrow f(0) = 0 \rightarrow g() \rightarrow g = 0$
$prev(m+1) = m \rightarrow f(m+1) = m \rightarrow h(m, f(m)) = m$
$\rightarrow h = \pi^2_1$

$prev = PR(0, \pi^2_1)$

mult is P.R.:

$mult(a, i+1) = mult(a, i) + a$
$mult(a, 0) = 0$;
$f(n_1, 0) = 0; \rightarrow g = 0$
$f(n_1, m+1) = f(n_1, m) + n_1 \rightarrow h(n_1, m, f(n_1, m)) = \pi^3_3 + \pi^3_1$
$\rightarrow h = add(\pi^3_3, \pi^3_1)$
$\therefore mult = PR(0, add(\pi^3_3, \pi^3_1))$.

$\therefore$ (i) is P.R. proved.

(ii) # less than is P.R

$a < b \Leftrightarrow a \leq b\ \&\&\ !(a = b)$. —— (1)

$\leq$ is P.R. (below); $a = b$ is P.R. —— (2)

Again, $a = b \equiv a \leq b\ \&\&\ b \leq a$ → is P.R. —— (3)
Also, $a \neq b \equiv !(a = b)$ → is P.R —— (4)
Again, $a \leq b \rightarrow eqto0(a - b)$
$\rightarrow a - b = \begin{cases} a - b & \text{if } a > b \\ 0 & \text{otherwise} \end{cases}$

$eqto0(n) = \begin{cases} \text{true if } n = 0 \\ \text{false if otherwise} \end{cases}$

$eqto0(0) = 1$ $f(0) = 1$ $g() = 1$
$eqto0(m+1) = 0$.
$eqto0(m) = 0$.
$h(m, f(m)) = 0$
$eqto0(n) = P.R.(\sigma \cdot 0, 0)$

(4) is P.R. from which 3, 2, 1 is P.R. follows.

So, (i) — If - then - else is P.R
(ii) — less than is P.R.
(iii) - add is P.R

$\therefore$ a%b → remainder is P.R.

a%b is μ-recursive as it can be expressed in μ-recursive terms.

$a \% b \equiv r$; $r < a$, 0 otherwise

$\Rightarrow a \% b \equiv r;\ 0 \leq r < a$

$a / b = m + r$
$a / b = m + r;\ 0 \leq r \leq b$
$\Rightarrow a = bm + r;\ 0 \leq r \leq b$ (i)
$(b * m) \leq (b * m) + r < (b * m) + b$
$\therefore (b * m) \leq (b * m) + b$
$\Rightarrow (b * m) \leq a < (b * m) + b$
→ smallest m; for which $\mu m\,(a < b * (m+1))$.

$\therefore a \% b \equiv \mu m\,(a < b * (m+1))$ → hence, μ-recursive.

35/40 = 8.8/10 10.0

2. *Computable sets, computable functions, and Turing machines:*

10/10 2a. Why do we need to study decidable and recursively enumerable (r.e.) sets?

10/10 2b. Is the intersection of two r.e. sets always r.e.? If yes, prove it, if no, provide a counterexample.

10/10 2c. Prove that it is not possible, given a program that always halts, to check whether this program always computes 2* n + 1.

10/10 2d. Design a Turing machine that computes 2 * n + 1 in binary code. Trace this machine on the example of n = 10_2.

2a) computations → math-wise complicated. but Maths has simple, basic concepts of "sets" to represent these computations in clear, concise terms, easy to understand. We need to study decidable set to have an algo to check whether n ∈ A at a moment of time. We need to study recursively enumerable set to prove that there is an algo. which eventually prints all of its elements.

2b) Yes intersection of 2 r.e. sets is always r.e.
A, B → r.e.
A, B → let each run for 1 hr
1hr. A → print x.
1hr B → print y
Assuming it is okay to repeat, the computer can print the elements from A for 1 hr, B for 1 hr search for elements in common, print them as A ∩ B. Repeat until finished. Hence, all elements gets eventually printed.
A ∩ B → r.e.

2c) By contradiction, let a program is possible which always checks whether this program computes 2 * n + 1.

$$2*n+1\text{-checker}(p) = \begin{cases} 1 & \text{if } \forall n\,(p(n) = 2*n+1) \\ 0 & \text{if } \exists n\,(p(n) \neq 2*n+1) \\ \text{whatever} & \text{if } p \text{ doesn't always halt.} \end{cases}$$

We build a zero-checker on this:

$$q \longrightarrow \forall n\,(q(n) = 0) \xrightarrow[\text{checker}]{2*n+1} \forall n\,(p(n) = 2*n+1)$$

$$\downarrow$$

$$p(n) = q(n) + 2*n+1$$

$$\text{zero-checker}(q) = [2*n+1\text{ checker}]\,[q(n) + 2*n+1]$$

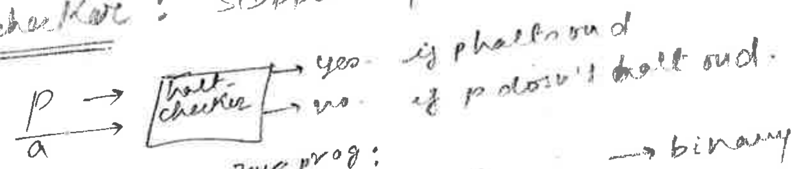
$$= \begin{cases} \text{yes} & \text{if } \forall n\,(q(n) = 0 \Leftrightarrow p(n) = q(n) + 2*n+1 = 2*n+1 \\ \text{no} & \text{if } \exists n\,(q(n) \neq 0 \Leftrightarrow p(n) = q(n) + 2*n+1 \neq 2*n+1 \\ \text{whatever} & \text{if } q \text{ sometimes doesn't halt.} \end{cases}$$

zero-checker $(f_{p,d})$ = true if $\forall t\ (f(p,d)(t) = 0)$
false if $\exists t\ (f(p,d)(t) > 0)$

halt-checker (p, d) = ! zero checker $f(p,d)$ → true if p halts on d.
→ false if p doesn't halt on d.

But halt-checker is not possible, ∴ zero-checker not possible →
hence, our assumption wrong and said algo. not possible.

halt-checker : suppose possible.

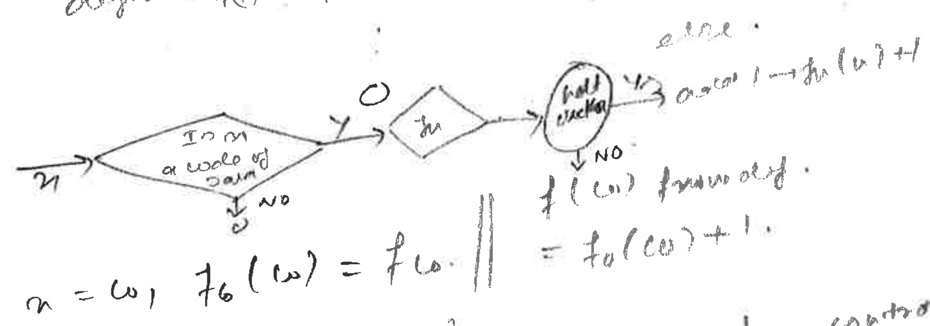


code of Java 'C' : Java prog → ASCII → binary '0's and 1s → append '1'

Lemma : ∃ an algo that given natural no. C, checks whether 'C' code of Java, returns one file, f_C.

define $f(n)$: $f_n(n) + 1$ n is a code of Java

else. ∴ f is computable.



$f(c_0)$ from def.
$= f_0(c_0) + 1$.

$n = c_0$, $f_0(c_0) = f_{c_0}$ ||

∴ $f_{c_0}(c_0) = f_0(c_0) + 1 \Rightarrow 0 = 1$. contradiction
halt-checker not possible.

[2*n+1]

f(g(n)) → Composition Turing Machine:

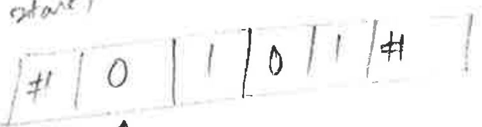
g(n) = 2*n = (n+n)

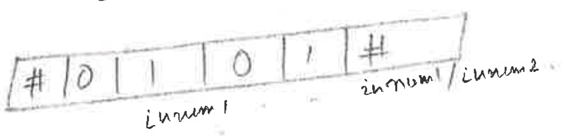
(start,#) → innum1, R
innum1, 1 → R; innum1, 0 → R
innum1, # → 1, R, innum2
innum2, 0 → R
innum2, 1 → R; innum2, 0 → R
innum2, # → L, ready1
ready1, 1 → L, #, rewind1; rewind1, 0 → L
rewind1, 1 → L; rewind1, # → start2

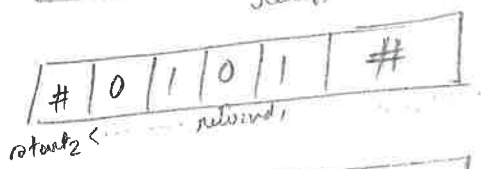
f(n) = n+1

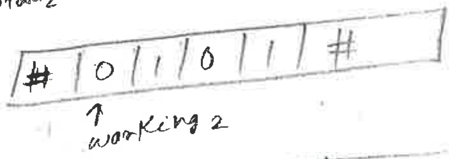
start2, # → working2, R
working2, 0 → 1, L, back2
working2, 1 → 0, R
working2, # → 1, back2, L
back2, 1 → L; back2, 0 → L
back2, # → halt

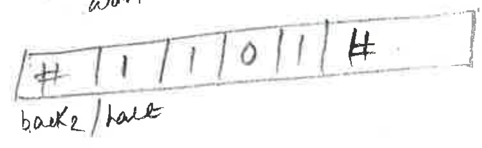
f(g(n)) → 10₂ = n











Turing Machine $2*n+1$

Given $n = 10_2$

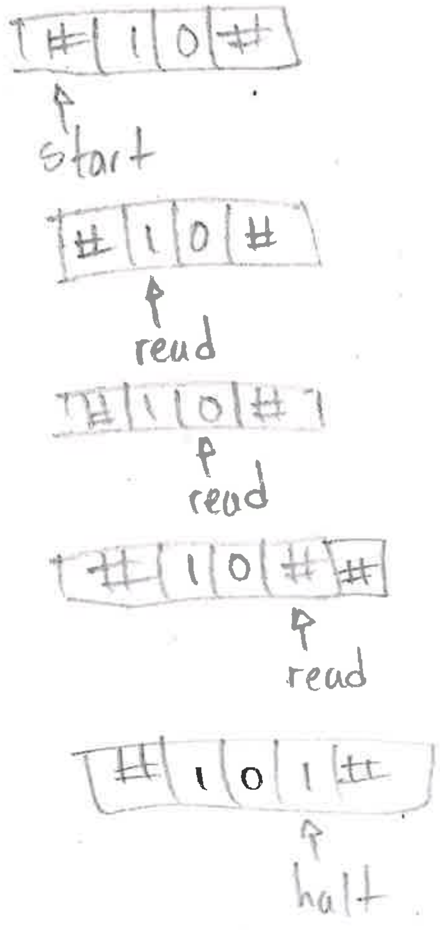
Want 101_2

start, # → R, read

read, 1 → R

read, Ø → R

read, # → 1, halt



2a. So we can understand other concepts such as when will the intersections of two sets print a number. If they are not r.e. that (printing number) can't be done.

67/70 = 9.6/10

3. *NP-hardness:*

3a-d. Reduce the satisfiability problem for the formula (a ∨ ~c)& (~b ∨ c) & (a ∨ b ∨ c) to:

a) 3-coloring, 10/10
b) clique, 10/10
c) subset sum problem, and 10/10
d) interval computations. 10/10

10/10 3e. In proofs of what results are these reductions used? What do we gain by proving that a problem is NP-hard?

7/10 3f. Use the definitions of the classes P, NP, NP-hard, and NP-complete to describe, for each of these four classes, whether this class contains 3-coloring, 2-SAT, sorting, and subset sum. Explain your answers.

3g. Give two examples of why the current definition of a feasible algorithm is not perfect:

10/10

- an example when an algorithm is feasible according to this definition but not practically feasible, and
- an example when an algorithm is not feasible according to this definition but is practically feasible.

3.g) Eg ① practically not feasible by defⁿ feasible:

$t_A(n) = 10^{200}\,\text{len}(n)$.

② practically feasible, by definition not feasible:

$t_A(n) = \exp(10^{-80}\,\text{len}(n))$.

3.e) These reductions are used in the proof of NP hardness.

Gain by proving a problem in NP-hard:

i) If a problem is NP-hard, unless P = NP, no feasible algo. is possible for solving all particular cases of this problem. Thus, we should not waste time looking for a solution.

ii) By defⁿ, a problem in NP hard, every prob. of class NP can be reduced to i.e. Thus, we have a good heuristic feasible algo. for solving some particular cases of this problem.

3.b)

3-colouring : cannot be solved in poly. time, not P.

: 3-SAT is reduced to it → NP-hard

: each node has 3 colours to choose from → NP.

∴ NP, NP-hard → NP-complete

2-SAT : solve in polynomial time → P.

SAT ~~reduced to it~~ → ~~NP hard~~

P ≠ NP; ∴ ~~not NP~~

∴ not NP-complete.

2-SAT is polynomial time it is in NP

Sorting : → polynomial time sol^n → P.

→ represented as a guess of correct order → NP

∵ P ≠ NP ∴ not NP hard, not NP-complete

Subset Sum : polynomial time solution → P.

3-CNF reduced to it → NP hard.

not a single solution → not NP.

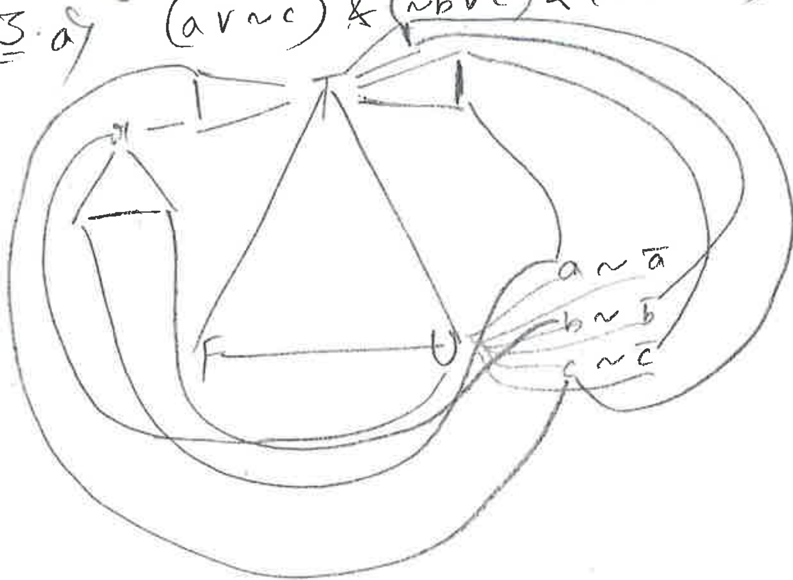
∴ not NP complete.

NP it is in NP

so it is NP-complete

3.a) (a ∨ ~c) & (~b ∨ c) & (a ∨ b ∨ c)

3-colouring.



or solving one particular case of this problem, then reduction automatically generates algorithm for solving particular cases of all NP-hard problems.

3f)

	3-Coloring	Clique	2-SAT	Interval Computation
P	No, unless P = NP	No, unless P = NP	yes	No, unless P = NP
NP	yes	yes	yes	no one knows
NP-hard	yes	yes	No, unless P = NP	yes
NP-complete	yes	yes	~~No one knows~~ No, unless P=NP	no one knows.

3g) Some algorithm are feasible, they are usually polynomial time, so, an algorithm is feasible if it takes polynomial time. There exist a polynomial $p(n)$ such that $\forall$ input x

$t_A(x) \le p(len(x))$

* example of when algorithm is feasible according to the above definition but not practically feasible

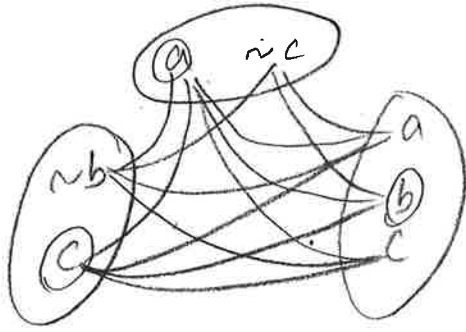
$t_A(x) = 10^{40} \cdot len(x)$ This is feasible in the view point of the definition but not practically

* $t_A(x) = exp(10^{-50} \cdot len(x))$: Feasible in practical

3 b) clique:

$(a \vee \sim c) \wedge (\sim b \vee c) \wedge (a \vee b \vee c)$

→ connect all nodes other than variable & negation



solution:

a = T
b = T
c = T
clique: connected.

3 c) Subset problem.

	a	b	c	c1	c2	c3
a	1	0	0	1	0	1
$\bar{a}$	1	0	0	0	0	0
b	0	1	0	0	0	1
$\bar{b}$	0	1	0	0	1	0
c	0	0	1	0	1	1
$\bar{c}$	0	0	1	1	0	0
h1				1	0	0
g1				1	0	0
f2				0	1	0
g2				0	1	0
f3				0	0	0
g3				0	0	0
	1	1	1	3	3	3

a = T
b = T
c = T.

Internal computations:

$(a \vee \sim c) \;\&\; (\sim b \vee c) \;\&\; (a \vee b \vee c)$

$\equiv [1-(1-a)(1-(1-c))]\,[1-(1-1-b)(1-c)]\,[1-(1-a)(1-b)(1-c)]$

$= [1-(1-a)(c)]\,[1-(b)(1-c)]\,[1-(1-a)(1-b)(1-c)]$

$\therefore F \equiv 1.$

$\Rightarrow 1-(1-a)(c) = 1$; $1-(b)(1-c) = 1.$ $1-(1-a)(1-b)(1-c) = 1$

$\Rightarrow 1-a = 0$ or $c = 0$; $\Rightarrow b = 0$ or $c = 1.$; $\Rightarrow a = 1$ or $b = 1$ or $c = 1.$

$a = 1$ or $c = 0$

Hence, $a = 1, b = 1, c = 1$ satisfies the problem.
(chosen solution)

36/40 = 9.0/10

4. *Parallelization:*

10/10 4a. matrices (2-D arrays) are added component-wise. Does the problem of adding n matrices belong to the class NC? Explain your answer.

8/10 4b. If we parallelize and take into account communication time, what is the fastest that we can compute the sum of two matrices? Explain your answer.

8/10 4c. Where are similar arguments used in the proof that satisfiability is NP-hard?

10/10 4d. What are the physical assumptions behind these arguments? Give two examples of how non-standard physics -- in which these assumptions are not satisfied -- can be used to solve NP-hard problems in polynomial time.

4a) $\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} + \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & & \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$

$c_{ij} = a_{ij} b_{1j} + a_{12} b_{2j} + \cdot \; a_{in} b_{nj}$.

|| elements in matrix $= n^2$. For each matrix, n processors. $\therefore n^2 \times n \to n^3$ no. of processors.

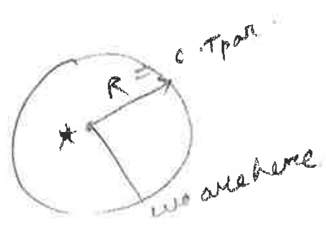
Adding 'n' matrices: $O(\log(n))$ time

Yes, it belongs to the class NC. Since, it is solvable in polynomial time. F, by polynomial processors and NC is the class of problems solved in polylog time.

4b) Parallelize, processing time $\to T(par(n))$ with communication.

Sequential processing time $\to T(seq(n))$.

communication done is almost as fast as speed of light, the diagram.



Assuming Volume of sphere, $V = 4/3 \pi R^3 = 4/3 \pi c^3 T_{par}(n)^3$.

$\Rightarrow$ Sphere can fit 'n' processors.

$\therefore N_{processors} \leq V/\Delta V = \frac{4/3 \pi c^3}{\Delta V} \cdot T_{par}(n)^3$

$\Rightarrow T(seq(n)) \leq N_{processors} \leq c_1 T^3_{par} \cdot T_{par} \leq c \cdot T^4_{par}(n)$

$\Rightarrow T^4 par(n) \geq c \cdot T_{seq}(n)$

$\Rightarrow T\, par(n) \geq c \cdot [T_{seq}(n)]^{1/4}$.

This is the fastest assuming Euclidean physics.

In this case what is it? $T_{seq}(n) = n$ so $\geq c \cdot n^{1/4}$

4c) Similar arguments used in the proof that satisfiability is NP-hard. Argument of using speed of light as the maximum transfer of speed is used in the proof of NP-hardness of SAT to show that at any given cell depends on a finite no. of neighbours & they only depend on finite polynomial no. of states which can be computed in binary way

As # is polynomial the total # of processors is polynomial

4d) Physical assumptions based on Euclidean arguments:
we state that $u \leq c$ & $v = \frac{4}{3}\pi r^3$, & because info. can't travel faster, we can only use polynomial no. of processors which are not enough to solve exponential time problems.

Non-Eucleadean physics can help solve NP-hard problems in polynomial time:

1) Let a computer run & send signal back in time when done.

2) Use of Lobachevsky space: allows us to fit an exponential no. of processors in the sphere which would allow to compute exponential time algo in polynomial time

3) Exploit causality by having random no. generators try to find a sol^n & trying to provoke very low probability events (much lower than finding the correct sol^n) by travelling back in time & trying to disrupt causality.

30/20 = 15/10

5. *How to solve NP-hard problems?*

10/10 5a. Suppose that a probabilistic algorithm for checking the program correctness misses the program's mistake 20% of the time. How many times do we need to repeat this algorithm to achieve reliability of 99%? And why do we need probabilistic algorithms in the first place?

10/10 5b. Apply both greedy algorithms to solve the following particular case of the knapsack problem: we have 4 objects with weights 20, 30, 40, and 50, values 1000, 1200, 1200, and 1500, and the overall weight of 50. Why do we need to use these algorithms, if they do not produce optimal results?

10/10 5c. (For extra credit) Give an example of how quantum computing can speed up computations.

5 a) prob. of missing a mistake → 20% → $\frac{20}{100} = 0.2$ $p_0 = 0.2$.
Reliability → 99% means probability of missing does not exceed (100 − 99)%
= 1%. ∴ $\varepsilon = \frac{1}{100} = 0.01$.

K → no. of times algo is to be repeated.

Now, $p_0^K \le \varepsilon$.

$\Rightarrow K \ln p_0 \le \ln \varepsilon$

$\Rightarrow K \ge \frac{\ln \varepsilon}{\ln p_0}$ [∵ $p_0 \le 1$]

$\Rightarrow K = \frac{\lceil \ln \varepsilon \rceil}{\ln p_0}$

$= \frac{\lceil \ln 0.01 \rceil}{\ln 0.2}$

≈ 2.86.

$= 3$.

∴ No. of times algo. is to be repeated → 3.

Need of Probabilistic Algo.: i) Many practical problems are NP-hard.
ii) This means that unless P = NP, no feasible algo is possible that solves all instances of this problem
iii) In some cases, it is possible to solve some instances.

5 c) Quantum computing can speedup computations through superpositions.

Digital computers: everything 0 & 1

Quantum ": every 2 states 0 and 1 can also have a superimposition as.

$c_0|0\rangle + c_1|1\rangle$; c_0, c_1 are complex numbers.

$|c_0|^2 + |c_1|^2 = 1$; → measure this qubit (quantum bit) ; states

prob. of obtaining 0 → prob. of obtaining 1.

faster computations

→

→ $O(n^{1/2})$. time complexity.

5 b) 4 objects:

Max W = 50

	x_1	x_2	x_3	x_4
w_i	20	30	40	50
v_i	1000	1200	1200	1500

1st greedy algo: pick highest value first.

$x_4 = 1$; $v_i = 1500$; $w_i = 50$ maximum weight is reached.

∴ $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1$ in the solution.

Gain is 1500/-

2nd greedy algo.: pick highest ratio first

	x_1	x_2	x_3	x_4
w_i	20	30	40	50
v_i	1000	1200	1200	1500
v_i/w_i	50	40	30	30

W = 20

∴ $x_1 = 1$; $v_1 = 1000$, $w_1 = 20$.

then $x_2 = 1$; $v_1 = 1200$, $w_2 = 30$ W = 20 + 30 = 50. Maximum weight is reached.

∴ Solⁿ is $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$

∴ Gain is 1000 + 1200 = 2200/-.

→ Need to use these algos., because progress in 1 problem immediately enhances progress in all others.

30/30 = 10/10

6. *Beyond NP: polynomial hierarchy:*

10/10 6a. Which class of the polynomial hierarchy contains optimization problems? The problem of winning in 3 moves? Explain your answers.

10/10 6b. Which class of polynomial hierarchy contains $\Sigma_4^{\Pi_1}$?

10/10 6c. Give an example of an oracle A for which $P^A = NP^A$. Explain your answer.

6) (a) Optimization problems:

$\exists p \forall q C(p, q)$

$\rightarrow$ belongs to class $\Sigma_2 P$.

[NP: $\exists y\, c(x, y)$
CO-NP: $\forall y\, c(x, y)$]

Winning in 3-moves:

$\exists p_1 \forall p_2 \exists p_3 \forall p_4 \exists p_5$

$\rightarrow$ belongs to class $\Sigma_5 P$

(b) $\Sigma_4^{\Pi_1 P} \equiv \exists x_1 \forall x_2 \exists x_3 \forall x_4\, P(x_1, \ldots x_4)$

$\equiv \exists x_1 \forall x_2 \exists x_3 \forall x_4 \forall x_5\, Q(x_1, \ldots x_5)$

$\equiv \exists x_1 \forall x_2 \exists x_3 \forall x_4'\, Q(x_1, \ldots, x_4')$

$\equiv \Sigma_4 P$

6 (c) $P^A = NP^A$ for oracle A = PSPACE

$P^A \rightarrow$ relative to an oracle class.

$NP^A \longrightarrow NP \in P^{SAT} \rightarrow$ we can reduce every NP – problem to satisfiability.

NP: $\exists y\, c(x, y)$: $\Sigma_1 P$

CO-NP: $\forall y\, c(x, y)$: $\Pi_1 P$

P(SPACE) $\equiv \exists x_1 \forall x_2 \ldots P$ ... polynomial no. of quantifiers.

$P^{PSPACE} \equiv \exists x_1 \forall x_2 \ldots \equiv$ [PSPACE]
(# poly quantifiers)

$NP^{PSPACE} \equiv \exists x_1\, P^{PSPACE}(x_1, x_2) \equiv \exists x_1 \exists x_2 \forall x_3 \ldots$ (combining) $\rightarrow \exists y\, c(x, y)$

$\cong$ PSPACE

$\therefore P^{PSPACE} = NP^{PSPACE}$.

$\therefore P^A = NP^A$ for A = PSPACE.