I have completed this exam in a manner consistent with
the Honor code. I have completed it entirely on my
own and have not used any resources beyond
those listed above (textbook, notes/homework/projects,
class recordings/whiteboard notes). I have not and will
not discuss or share my part of this exam with other
students

Signed:

Sam Hopkins

Samuel Hopkins

CS 512 Final Exam

1. Linear Programming

A) Linear Regions:

$A \leq 20$

$A - 2B \leq 0$

$A + B \leq 24$

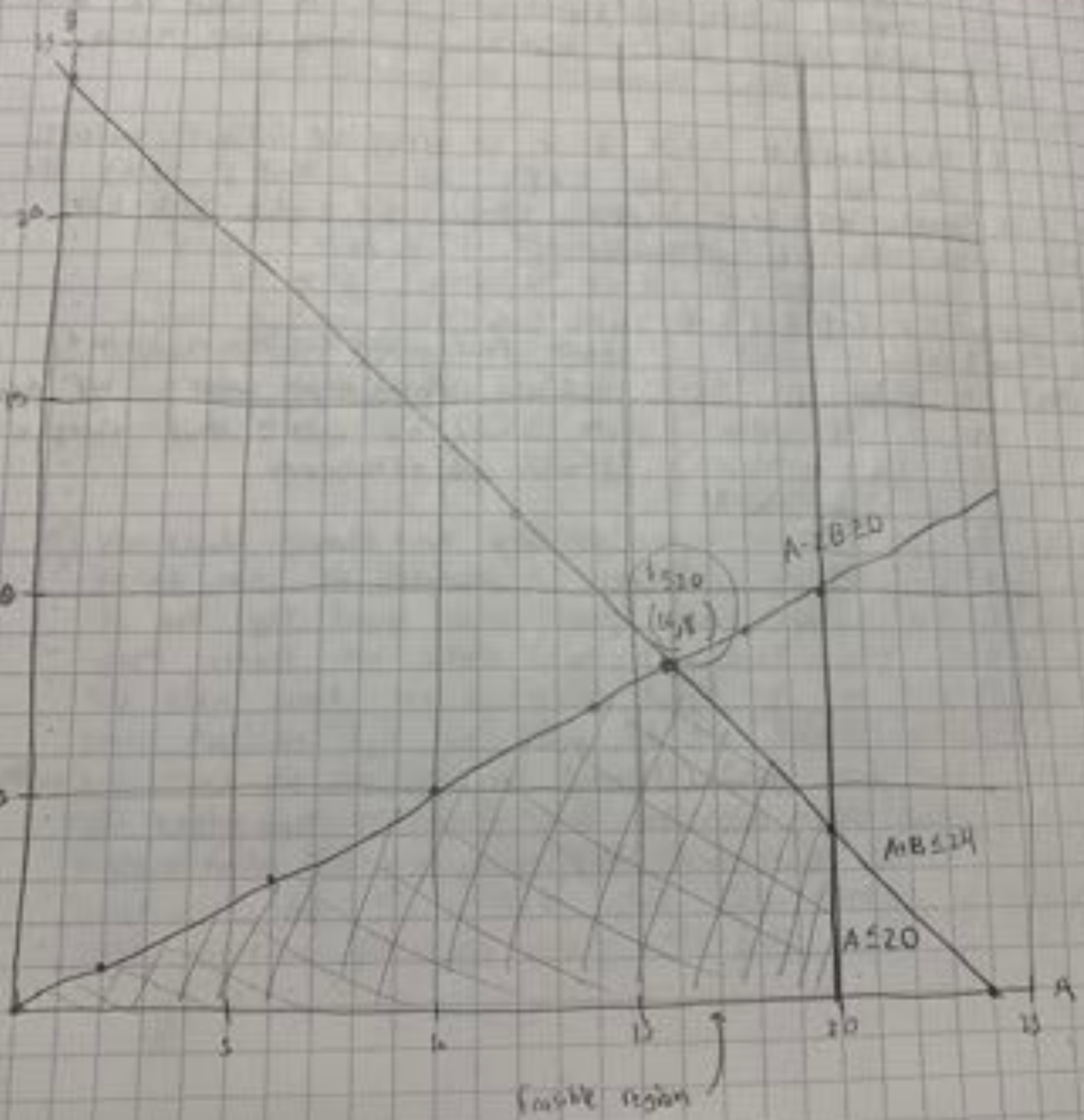
$\max(20A + 25B)$

Optimal solution:

A = 16

B = 8

Money = $520



2. Dynamic Programming

A) Basically, the solution for $T(i) = \sum T(i-1)$. The recursive relationship would rely on the current state/solution of the iteration before, like we did with Gene alignment. We can show the structure in Part B, but basically in each recursive step we rely upon the current score for the solution before us. We make values for matches, indels, and substitutions and use those to find the best possible solution before recursing down again.

B) This structure would be a 2d array of integer values. Its size would be $L \times R$ with L and R being the number of books in each stack. It might look like this for the example:

		Y	G	G	B	G	Y	B	G
	B								
	Y								
	B								
	G								
	Y								
	G								
	B								
	Y								
	G								
	Y								

Filled with optimal/best number for priority

Each place would contain a value to show optimization where we get an Indel, we can show where a book should be removed

c) Create 2d array arr with [illegible] value

for match: +1
for [illegible]: 0

```
for i in len(L):   # [illegible]
    for j in len(R):   # [illegible]
        at arr[i][j], look at top, left, and left diagonal values
        check that L[i] and R[j] are equal colors
            if so, arr[i][j] = max(dia, top, left) + 1
                # only increment max height if book colors are =
        else if max(dia, top, left) = diag   # [illegible] book from both stacks
        else if max(dia, top, left) = left   # [illegible] book from L
        else if max(dia, top, left) = top    # [illegible] book from R
return arr[L][R]   # bottom right corner for max height
```

d) Both time and space are O(mn) with m being length of L and n being length of R. Space complexity comes with the two 2 array, [illegible] to be mxn space, changing values which does not alter the space as changes are O(1). Time is also O(mn) because we iterate through each element in arr, each iteration being O(1), so double for loops used give us O(mn) time.

e) You could create two boolean arrays (Larray and Rarray) of size n (L) and m (R). As we run the program, we maintain pointers to which positions in arr we have come from, which produces an optimal path when we start from the lower right corner (max height). We then back track, keeping track of the max height at each position i, j we visit. If the height doesn't change, (and we came from a diagonal) we mark the position at Larray[i] = false and Rarray[j] = false. If height doesn't change and we came from below, Rarray[j] is marked false. If we come from right index, Larray[i] = false. This leaves us with the arrays where false means removing the book at that position from the pile.

3.

a)

∞	5	6	4	3
9	∞	5	6	5
5	9	∞	7	5
8	10	5	∞	5

→

∞	0	2	1	3
4	∞	0	1	5
0	3	∞	2	5
3	5	0	∞	5
0	0	0	1	

→

∞	0	2	0	3
4	∞	0	0	5
0	3	∞	1	5
3	5	0	∞	5
0	0	0	1	

	A	B	C	D	
A	∞	0	2	0	
B	4	∞	0	0	
C	0	3	∞	1	
D	3	5	0	∞	13 19

3+5+1+5+1 = 14

b)

	A	B	C	D	
A	∞	0	2	0	
B	4	∞	0	0	
C	0	3	∞	1	
D	3	5	0	∞	19

2nd

	A	B	C	D	
A	∞	∞	∞	∞	
B	4	∞	0	0	
C	0	∞	∞	1	
D	3	∞	0	∞	19

	A	B	C	D	
A	∞	∞	∞	∞	
B	4	∞	∞	0	
C	0	1	∞	1	
D	0	0	∞	∞	37

	A	B	C	D	
A	∞	∞	∞	∞	
B	4	∞	0	0	
C	4	0	∞	∞	
D	3	7	0	∞	22

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	∞	
C	0	∞	∞	0	
D	0	∞	∞	∞	23

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	∞	
C	0	∞	∞	∞	
D	3	∞	0	∞	19

4. local search

a) 1. we need an initial state, so assign the tiles in order to a coefficient
2. we need to define our neighborhood
3. compare and prioritize solutions/heuristic

Initial state →

assign randomly each tile to a position
while total != y:
if the total is > y:
swap the highest [illegible] tile with the tile-1 value

how to neighbor ←

if total < y
swap the lowest [illegible] tile with next value

prioritize values that are closer to y

↙ heuristic

b) no, a local search can only provide a local optimum solution, which is not the same as a global optimal solution. It might be optimal for its given neighborhoods, but not necessarily for the whole solution. if every way it tries next is worse it will claim to have the optimal, which isn't always the case

5. The goal of this problem is to reach as many people (or everyone) as quickly as possible. This is a BFS solution

```
for each position to place the truck in the neighborhood
#(This is assuming the graph is made for each house with a
# distance of 1 to each other node)

    run Dijkstra's algorithm on the graph, with starting position
    being the truck position
        if max distance to another node is ≤ previous truck
            position distances
                That is the new best truck location
```

The distances computed by Dijkstra's are the number of minutes it would take to vaccinate that house, find the optimal position by running it on every possible position. We prefer the max distances to be lower, so we want the position that gives us the shortest tree from Dijkstra's Algorithm.

You also could use Kruskal's algorithm and create a minimum spanning tree to run Dijkstra's on, but I think using Dijkstra's is good enough

6.

a) Use Dijkstra's, with a graph only going right and down, with end node being cost, start node being [illegible] [illegible]

b) Use Dijkstra's [illegible] BFS

c) Use Dynamic Programming

D) This is DFS

```
grid = [illegible] of [illegible] [illegible] [illegible] spaces
end = grid[R-1][C-1]

recurse (position, [illegible])
    if (position = end)
        return 1
    if can go down and right
        return recurse(position[j-1]) + recurse(position[i+1])

    if only can go down
        return recurse(position[j-1])

    if only can go right
        return recurse(position[i+1])
```

Time complexity: O(V+E), this is because V is number of vertices (possible squares) E is paths from those squares. Each path has at most 2 possible paths, + is a simple DFS algorithm

Space complexity: O(R·C), this is for the initial grid, nothing else is taking up space besides O(1) returns

One thing I could improve is by using some Dynamic Programming algorithms, it might improve the time complexity a bit.