Name:
Student ID:

Please read all of the following information before starting the exam.

• You have three hours (180 minutes) to work the exam. For courtesy to other students, no one may leave during the last 10 minutes of the exam.
• To receive full credit, show all work, clearly and in order. Points may be taken off if we cannot see how you arrived at your answer (even if your final answer is correct).
• Circle or otherwise indicate your final answers.
• Be clear and to the point. Points will be taken off for rambling and for incorrect or irrelevant statements.
• Algorithm questions do not require a complexity analysis or proof except where explicitly stated.
• The test has 5 problems and is worth 90 points. There are 12 pages, including one scratch page, and it is your responsibility to ensure you have all of them.
• You may use two double-sided 8.5” x 11” sheets of handwritten notes during the exam. Except for this, the exam is closed-book. Any breach of this policy will be regarded as a breach of the UCSD Policy on Integrity of Scholarship which will result in an automatic zero grade and further disciplinary action.

Good luck!

POINTS: #1 /15
#2 /15
#3 /15
#4 /15
#5 /30

TOTAL: /90
1. (15 points) **Divide-and-Conquer.**

Design a divide-and-conquer algorithm to determine whether two elements in an array of \( n \) integers add up to a given integer, \( x \).

**Example:** For the array of \( n = 8 \) integers \([5, -3, 18, 99, -9, 3, 1, 50]\) your algorithm should return **YES** if \( x = 47 \) (since \( 50 + (-3) = 47 \)), and **NO** if \( x = 200 \).

(a) (4 points) Give a short description of your algorithm.

(b) (7 points) Write out the pseudocode.

(c) (4 points) Analyze the divide-and-conquer recurrence to obtain the time complexity.

Note: Maximum credit requires a D/Q algorithm that runs in \( O(n \log n) \) time.

At most 50% credit will be given for a D/Q algorithm that runs in \( O(n^2) \) time.

Zero credit will be given for a non-D/Q algorithm.

(a) Given a list of values \( L \) and a target value \( x \). We sort \( V \) using merge/quicksort. Divide values of list \( V \) into two lists \( L \) and \( R \). We notice that \( L \) and \( R \) are still sorted. We pointer march from either end trying to find a pair which sums to \( X \). If no such value is left. We recursively attempt to do this on each of the sublists.

(b) Pseudocode:

\[
\text{dqFindSum(V,x)} \\
V = \text{mergesort}(V) \\
\text{dqSum(V,x)} \\
\text{if } |V| == 1: \\
\quad \text{return False} \\
L = V[0...|v|/2] \\
R = V[|v|/2 + 1 ... end] \\
i=0 \\
j=end \\
\text{while } (i < j): \\
\quad \text{if } (L[i] + L[j] > sum): \\
\quad \quad j -= 1 \\
\quad \text{else if } (L[i] + L[j] < sum): \\
\quad \quad i += 1 \\
\quad \text{else:} \\
\quad \quad \text{return True} \\
\text{return } (\text{dqSum}(L,x) \text{ || dqSum}(R,x)):
\]

(c) Complexity analysis:
We denote the length of our input array as $N$. Sorting a list using mergesort has a complexity bound of $O(N \log N)$. Dividing array and pointer marching once on a sorted array is $O(N)$. This is done recursively on each sub-array following the recursions $T(N) = 2T(N/2) + O(N)$, which has the complexity bound $O(N \log N)$ (for example from the Master method). Overall the algorithm is bound by $O(N \log N)$.

**Grading:**

*Below are the main ways points were lost:*

- Many other variations were accepted for full credit. For example: Sensible implementations based on Mergesort and explicit binary search, mergesort based implementations.

- Mergesort/quicksort methods are D/Q, however by themselves they do not form a D/Q solution to this problem. Do to this ambiguity we decided to award partial credit with -6 penalty.

- $O(n^2)$ working implementation -6

- $O(n^2)$ working implementation and non-D/Q -9

- Flawed or non-working pseudo code or idea -11

- 2 error in some boundary conditions.

- 1 lack of detail

- 1 to -3 unclear/overly long code (was rarely applied, mostly in cases of absurdly bad pencilmenship)

- -3 code non general code mostly solving for the example(!?!)

2. (15 points) **Dynamic Programming.**

You are given an $n \times n$ array $A$ of integers. The integers can be positive, negative or zero. The rows of $A$ are indexed by $i = 1, 2, \ldots, n$, and the columns are indexed by $j = 1, 2, \ldots, n$.

**Example** ($n = 3$): $a_{11} = -5, a_{12} = -10, a_{13} = 1000, a_{21} = 0, a_{22} = 100, a_{23} = -500, a_{31} = 30, a_{32} = 500, a_{33} = 20$

```
-5  -10  1000
 0   100  -500
 30  500   20
```

(a) (8 pts) Your task is to find a path of array elements $a_{ij}$, starting from the top-left element $a_{11}$ and ending at the bottom-right element $a_{nn}$, such that the sum of the integers along the path is maximum. The path can only move downward or rightward to adjacent squares. In the above example, the optimal path is -5, 0, 100, 500, 20 with sum = 615.
In an $O(n^2)$ dynamic programming solution, you will fill up a path cost table $C$ with entries $c_{ij}$.

Write down pseudocode for the dynamic programming algorithm. Base case (initialization) is worth 4 points, and recurrence is worth 4 points.

**Answer:** We can solve it by dynamic programming. We compute a table $m$ that contains the maximum sum for optimal path.

**Base Case:**
- $m_{11} = a_{11}$
- $m_{ij} = m_{i(i-1)} + a_{ii}, 2 \leq i \leq n$
- $m_{i1} = m_{(i-1)1} + a_{i1}, 2 \leq i \leq n$

**Recurrence Relation:**
- $m_{ij} = \max(m_{i-1}j + m_{i(j-1)}) + a_{ij}, 2 \leq i, j \leq n$

**Pseudocode:**

```plaintext
m_{11} = a_{11}
pn_{11} = NULL
for i = 2 to n
  m_{ii} = m_{i(i-1)} + a_{ii}; pn_{ii} = RIGHT;
  m_{ii} = m_{(i-1)i} + a_{ii}; pn_{ii} = DOWN;
for i = 2 to n
  for j = 2 to n
    if m_{i-1}j > m_{i(j-1)}
      m_{ij} = m_{i-1}j + a_{ij}
      pn_{ij} = DOWN
    else
      m_{ij} = m_{i(j-1)} + a_{ij}
      pn_{ij} = RIGHT
return m_{nn}
i = j = n
while pn_{ij} != NULL
  print pn_{ij}
  if pn_{ij} = RIGHT
    j = j - 1
  else
    i = i - 1
```

**Complexity:** We fill a table of $n^2$ entries and it takes constant computation for each entry. So, time complexity as well as space complexity is $O(n^2)$.

**Grading Scheme:**
- **Correct base case:** +2
- **Correct Recurrence:** +4
- **Correct Path:** +2
- **Only correct + complete recurrence but no pseudo code:** −4 points out of 8
No path tracking but keeps track of ‘prev’ – 1
No points for non-DP solution
1/2 points for solution that gives the intuition of recurrence (i-1, j), (i, j-1)
(b) (7 pts) A modified
task is to find a path of array elements $a_{ij}$, starting from any element $a_{p1}$ in the first column and
ending at any element $a_{qn}$ in the last column, such that the sum of the integers along the path is
maximum. Again, the path can only move downward or rightward to adjacent squares, so $p \leq q$.
In the above example, the optimal path is -5, 10, 1000 with sum = 615.

Write down the dynamic programming algorithm for this modified task. Base case
(initialization) is worth 2 points, recurrence is worth 3 points, how to obtain the optimum sum is
worth 2 points.

Answer:
We can simply change the base cases –
Base Case:
\[ m_{11} = a_{11} \]
\[ m_{1i} = m_{1(i-1)} + a_{1i}, \quad 2 \leq i \leq n \]
\[ m_{i1} = \max(a_{i1}, m_{(i-1)1} + a_{i1}), \quad 2 \leq i \leq n \]

Recurrence Relation:
\[ m_{ij} = \max(m_{(i-1)j} + m_{i(j-1)} + a_{ij}), \quad 2 \leq i, j \leq n \]

Pseudocode:
Same as part (a)

To get the optimum sum, we take the maximum of all entries in the last (right most) column.

Grading Scheme:
Correct base case: +2
Recurrence: +3
How to obtain the optimum sum: +2
No point for base case if it is partially correct
3. (15 points) **Network Flow.**

You are given an $n \times n$ grid. Each grid cell may have a coin on it (indicated by a “C” in the 4 $\times$ 4 example below), and there are $0 \leq k \leq n^2$ coins in total. If a given row has one or more coins in it, you can pick up to one coin from that row (i.e., to “represent” the row). Similarly, if a given column has one or more coins in it, you can pick up to one coin from that column (to “represent” the column). So, it is possible to pick up to $n$ coins for rows, and $n$ coins for columns.

Your task is to pick the maximum possible number of coins.

In the example, you can pick 8 coins – e.g., (1, 1), (4, 2), (3, 3), (4, 4) to represent the 4 columns and (1, 2), (2, 3), (3, 4), (4, 1) to represent the 4 rows.

![Grid Example](image)

(a) (6 pts) Formulate this as a maximum flow problem.

Give a short description of the network flow formulation (in terms of $n$, $k$, grid coordinates, etc.). Then, draw the flow network for the above example. Make sure to clearly label nodes and edge capacities.

**Solution:**

We can solve this problem using the same idea of bipartite matching. We need to find the maximum matching between $k$-coins and $n$-rows, columns. So, the flow network has 2 (for source, sink) + $k$ (for coins) + $2n$ (for rows, columns) nodes. All edges have capacity 1.

Here is the flow network for the above example –
All edges have capacity 1. We run max flow from ‘S’ to ‘T’, and the value of maximum flows denotes the maximum amount of coin that can be picked.

**Grading scheme:**
- Edges correct – +2
- Capacities correct -- +2
- Correct idea -- +2
- Edge direction missing – (-1/2)
- No point for a non-network flow solution
- Partial credit (maximum 2) for a network that does not solve the problem
(b) (4 pts) Write out the pseudocode that will convert the input grid and coin locations into the flow network.

**Pseudocode:**
1. Add a source node, ‘S’ and a sink node ‘T’
2. For each of the k-coins add a node C\_i (1<=i<=k) and add an edge from ‘S’ to ‘C\_i’ with capacity 1
3. For n-rows, columns add 2n nodes. Add an edge with capacity 1 from each row/column node to the sink node
4. If coin, has position (row\_i, column\_i) then add one edge from C\_i to row\_i node with capacity 1. Do the same for column position as well

**Grading scheme:**
*Maximum points for correct network construction*
*Partial credit for understanding that there will be edges from coin to corresponding row, column location*
*1 point for being consistent with the flow graph, when the graph itself is wrong*

---

(c) (5 pts) Your task is modified such that you can now pick at most 2 coins from any given row. So, picking 3 coins from the same row, as we did from Row 4 in the above example, is no longer allowed. Explain (and optionally show) how to modify your flow network in (a) to maximize the number of coins picked given this new restriction.

We can add an extra layer of nodes that will impose the constraint. The idea is we will add one node for each row. There will be an edge from source to this special row node with capacity 2. From each of this special node, we will have outgoing edges to corresponding coins in that row with capacity 1. The rest of the graph will be unchanged.

Here we show the new graph.
The unmarked edges have capacity 1 like part – (a)

**Grading Scheme:**
Correct idea of using capacity 2 and one extra node for each row – +2
Correct network formulation -- +3
No point deduction for not giving the flow graph
Both textual description/flow graph earns maximum points, if correct
Partial credit for extra nodes but with wrong capacity
However, if part-‘a’ is not correct, part ‘c’ is unlikely to get maximum credit
4. (15 points) NP-completeness.

**Definition:** Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset of vertices $V' \subseteq V$ such that for any edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$.

**Definition:** Given a graph $G = (V, E)$, an independent set in $G$ is a subset of vertices $V' \subseteq V$ such that no two vertices in $V'$ are adjacent in $G$.

**Vertex Cover Problem** $VC(G, k)$: Given a graph $G = (V, E)$ and a number $k$, does $G$ contain a vertex cover of size $k$?

**Independent Set Problem** $IS(G, k)$: Given a graph $G = (V, E)$ and a number $k$, does $G$ contain an independent set of size $k$?

[Hint: It may be helpful to draw minimum vertex covers and maximum independent sets in small example graphs such as that given below.]

---

Given that VC is NP-complete, prove that IS is NP-complete.

(a) (3 pts) Prove that IS is in NP.

**IS** is in NP as it is easily verifiable in polynomial time. Given $G(V, E)$ an input graph and $V'$ a solution to IS of size $k$. We iterate over all edges in $E$ whose vertices are both a part of $V'$. This may be done in $O(|E|)$ time, which is clearly polynomial in the input.

**Grading:**
- 0 Reduction proof
- -1/-2 lacking explanation of validation scheme.
(b) (4 pts) Describe a polynomial-time reduction from VC to IS.

This is a very straightforward reduction. Given an input to VC, a graph \( G(V, E) \) and some positive integer \( k \). Solve IS(\( G, |V| - k \)) and return the resulting output.

Also acceptable here is a reduction which feeds to IS the complement of \( E \) input edges with the same \( k \).

**Grading:**
- (-2/-3) Wrong reduction/non-working reduction/the trivial reduction.
- -1 stating the complementarity of the solutions insight but not in the form of a reduction awarded partial credit as well in some cases.
- -2 Correct but reversed reduction i.e. IS \( \rightarrow \) VC.

(c) (7 pts) Prove that your polynomial-time reduction in part (b) is correct.

**Reduction proof:**
We will show the concordance of solutions across our reduction.

(\( \Rightarrow \)) If VC(G,k) = true, it follows that IS(G,|V|-k) = true.
Let \( V' \) be a vertex cover with size \( k \). As \( V' \) is a VC of size \( k \) it follows that all edges in the graph are covered by \( V' \) (i.e. at least one of their adjacent edges is in \( V' \)). Now define \( C = V \setminus V' \) and notice \( |C| = |V| - k \). We also note that by the definition for a VC, \( C \) cannot contain an edge between two vertices in \( C \), otherwise this edges would not be covered by the valid VC \( V' \). Thus, this collection of vertices meets the requirements for an IS of size \( |V| - k \).

(\( \Leftarrow \)) If IS(G,|V|-k) = true, it follows that VC(G,k) = true.
Let \( C \) be an independent set of size \( |V| - k \). From the definition of IS \( C \) contains no pair of vertices which is connected by an edge. We define \( V' = V \setminus C \) and notice that it is of size \( k \). Now, assume for the sake of contradiction that \( V' \) is not a VC. That means that there exist some edge \( e=(u,v) \) in \( E \) which is not covered by \( V' \). Thus it follows that both \( u \) and \( v \) are part of \( C \), a contradiction to our assumption that \( C \) is an independent set. Thus it follows that \( V' \) is a VC of size \( k \), as was needed.

As both directions hold our reduction is correct. The reduction is done in constant (i.e. O(1)) time and is thus trivially polynomial.

**Grading:**
- A by directional proof along similar veins was awarded full credit.
- 1-2 points where awarded for a careful justification of a wrong/non-working reduction.
- -3/-4 Half proof single directional proof of the correct reduction.
- 2-3 Non-proof of the correct reduction. Points were mainly lost for people stating the same thing as in (b) in different words.
- -1 No mention of the polynomial complexity aspect of the reduction.
- 0 points just discussing the polynomial complexity.
(d) (1 pt) If a new algorithm for IS were to be invented tomorrow with polynomial worst-case time complexity, how would this affect our understanding of computational complexity?

Anything which conveyed the idea that this would result in \( P = NP \).
Credit was lost if the idea suggested that any problem would be immediately solvable.

5. (30 points) **Short Answers.**

(a) (6 pts) Your roommate eats only Burritos or Burgers.

One Burrito costs $8, and gives 2250 calories, with 20 grams of protein.
One Burger costs $12, and gives 1200 calories, with 60 grams of protein.

What is the minimum-cost diet that allows your roommate to consume at least 45000 calories and 1200 grams of protein?

Write this down as a linear program, being clear to specify the objective and all constraints.

**For 5(a):**

Minimize \( 8x + 12y \)

Subject to:

\[
2250x + 1200y \geq 45000 \\
20x + 60y \geq 1200 \\
x, y \geq 0
\]

5 points were awarded for an otherwise correct solution that omitted the nonnegativity constraints.
2 points were awarded for having any part of a correct solution.
(b) (4 pts) Which of these problems are known to be NP-complete? (Circle your answers.)

   i.   5-SAT
   ii.  2-SAT
   iii. Euler Path (finding a path that goes through every edge of a graph exactly once)
   iv.  0-1 Integer Linear Programming (variables can only take on values in \{0,1\})
   v.   Minimum Vertex Cover in a bipartite graph
   vi.  Hamiltonian (Rudrata) Path in a DAG
   vii. Longest Simple Path (no repeated vertices) in a graph
   viii. Maximum Spanning Tree

For 5(b):

5-SAT, 0-1 ILP and Longest Simple Path in a graph are NP-complete.

2-SAT, Euler Path, Minimum Vertex Cover in a bipartite graph, Hamiltonian (Rudrata) Path in a DAG, and Maximum Spanning Tree all have polynomial-time solutions.

0.5 points were awarded for each problem that was classified correctly.

(c) (20 pts) Mark each of the following statements as True or False. Give a brief justification. (No credit will be awarded without a justification.)

   i.  Dijkstra’s algorithm is an example of a greedy algorithm.

      Dijkstra’s algorithm is an example of a greedy algorithm. TRUE. 2 points awarded if next-shortest path length (not edge length!) was mentioned; 1 point was given for mentioning priority queue.

   ii. Given a flow network, in a maximum flow at least one edge will have flow equal to capacity.

      Given a flow network, in a maximum flow at least one edge will have flow equal to capacity. TRUE. 2 points awarded for mentioning duality or max-flow-min-cut or existence of an s-t path in the residual graph.

   iii. The Bellman-Ford algorithm can detect negative cycles that are not reachable from the source.

      The Bellman-Ford algorithm can detect negative cycles that are not reachable from the source. FALSE. 2 points awarded for noting that “not reachable” matters: B-F cannot detect a cycle that it does not reach.
iv. Depending on the input, the asymptotic (big-O) runtime of Dijkstra’s algorithm using a binary heap PQ implementation can be greater than the runtime using an array-based PQ implementation. TRUE. 1 point awarded for giving the $O(V^2)$ vs. $O((V + E) \log V)$ comparison from Section 4.5.1 of the textbook (“Which heap is best?”), and full credit (2 points) awarded for noting that the heap implementation is preferred for sparse inputs.

v. A straightforward D/Q implementation of matrix multiplication satisfies the recurrence $T(n) = 8T(n/2) + O(n^2)$. However, a D/Q implementation is also possible with recurrence $T(n) = 7T(n/2) + O(n^2)$. TRUE. 2 points awarded for mentioned Strassen or for otherwise recalling that a multiplication can be saved by computing intermediate terms or ‘trading multiplication for extra additions’.

vi. If $G = (V,E)$ has a cycle with a unique heaviest edge $e$, then $e$ cannot be in any MST of $G$. TRUE. 2 points awarded for any reasonable argument, or for mentioning the cycle property.

vii. Given that problem A reduces to problem B in polynomial time, and given that there exists a polynomial-time algorithm for A, it follows that there exists a polynomial-time algorithm for B. FALSE. 2 points awarded for noting (essentially) that if A reduces to B, and A is easy, we know nothing about B (from class slides).

viii. The worst case complexity of Bellman-Ford algorithm is $O(n^3)$, where $n$ is the number of nodes. TRUE. 2 points awarded for getting to this complexity in any way – e.g., $O(V \cdot E)$ when $E$ is $\Theta(V^2)$ (= dense input).

ix. The Floyd-Warshall algorithm always computes all shortest paths with 4 or fewer edges before it computes all shortest paths with 5 or fewer edges. FALSE. 2 points awarded for explaining that Floyd-Warshall relaxes to the correct APSP solution by growing the set of allowed intermediate
vertices on any i-j shortest path. 1 point awarded for saying “no, this is what Bellman-Ford does”.

x. Given an edge-weighted complete graph with eight nodes \{A, B, C, D, E, F, G, H\}. We compute an MST over the four nodes A-D, and another MST over the four nodes E-H. Then, we can always obtain an MST over the entire graph by joining the two MSTs using the minimum-cost edge between \{A, B, C, D\} and \{E, F, G, H\}.

FALSE. 2 points awarded for showing or explaining a counterexample. E.g., suppose all six edges AB, AC, AD, BC, BD, CD have cost 10, and all six edges EF, EG, EH, FG, FH, GH have cost 10, with the remaining C(8,2) – 12 = 16 edges having cost 1. Then joining the two MSTs results in a tree of cost 61, while the true MST (e.g., edges AE, EB, BF, FC, CG, GD, DH) has cost 7.
Scratch Page
(Please do not remove this page from the test packet.)