Practice Final
Autumn 2025

Name Answer Key

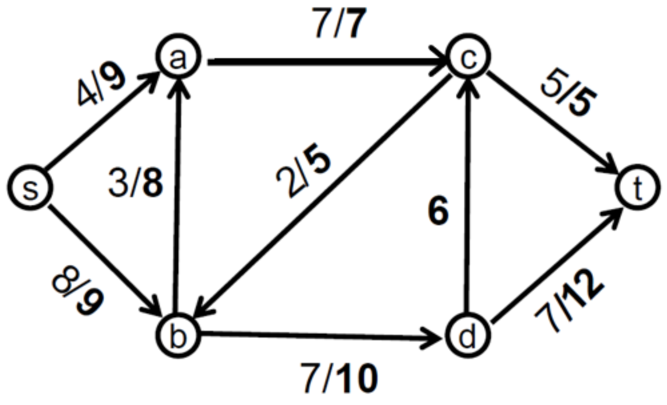
Net ID ____________________ (@uw.edu)

Academic Integrity: You may not use any resources on this quiz except for your one-page (front and back) reference sheet, writing instruments, your own brain, and the exam packet itself. This quiz is otherwise closed notes, closed neighbor, closed electronic devices, etc.

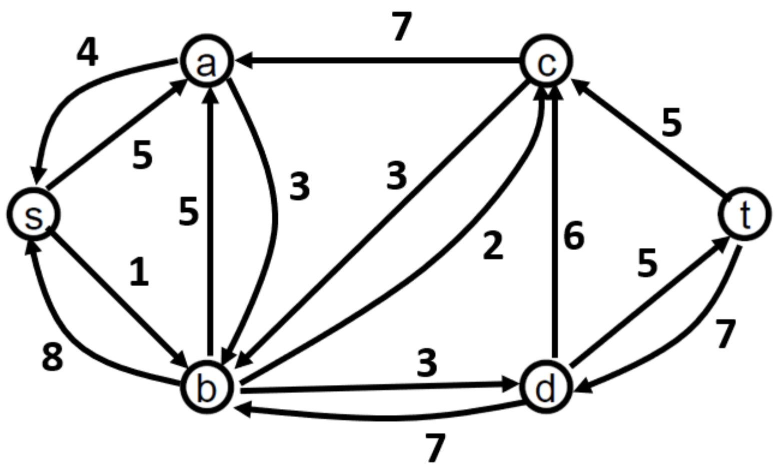
Instructions: Before you begin, **Put your name and UW Net ID at the top of this page.** Make sure that your name and ID are LEGIBLE. Please ensure that all of your answers appear within the boxed area provided.
The last page of this exam serves as a reference sheet as well as scratch paper. Please detach the last page from the exam packet. No markings on the last page will be graded.

(1 ESNU) **Question 1: Ford-Fulkerson**

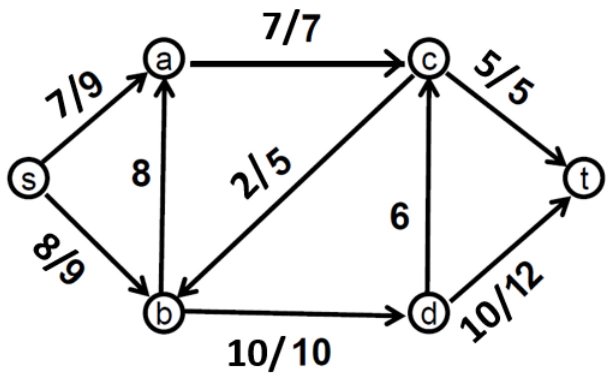
Consider the following flow network with a flow f shown. An edge labeled "b" means that it has capacity b and flow 0. An edge labelled "a/b" means that the flow on that edge is a and the capacity is b.



a) Draw the residual graph for f.



b) On the diagram below, indicate the new flow values for every edge resulting from augmenting f along the path s, a, b, d, t. (Blanks for all edges should be filled in.)



(1 ESNU) **Question 2: Project Staffing - Flow Reduction**

A project management company is tasked with hiring job applicants. Their objective is to hire a set of applicants so that all of the projects, which each requires one staff member of every role, can be fully staffed. For each applicant, we have identified roles they are qualified for. When an applicant is hired into a role (which they must be qualified for), they will then be assigned to one project (therefore each applicant can only work on one project and can only play one role). A project, to be fully staffed, must have one applicant assigned to it for every role. Your task is to write an algorithm to determine whether it is possible to fully staff all projects with the applicants provided.

The input to this algorithm will be:

– A set of m roles $R = \{r_1, ..., r_m\}$

– A set of x projects $P = \{p_1, ..., p_x\}$

– A set of n applicants $A = \{a_1, ..., a_n\}$.

– A set of qualifications per each applicant. We say $Q_i \subseteq R$ is the set of roles that applicant a_i is qualified for.

Using this notation, we say that a project p_i is fully staffed if it has m applicants assigned to it (one per each role) where if applicant a_i has role r_j then $r_j \in Q_i$.

Your task is to solve this problem by reducing it to Max Flow. The following questions will relate to this solution.

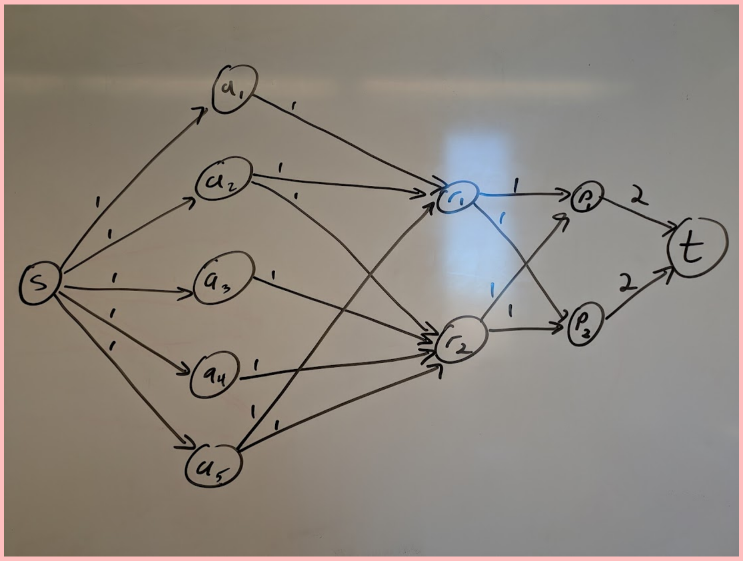
a) Draw an example of your flow network for the following sample input:

– $m = 2$

– $x = 2$

– $n = 5$

– $Q_1 = \{r_1\}$, $Q_2 = \{r_1, r_2\}$, $Q_3 = \{r_2\}$, $Q_4 = \{r_2\}$, $Q_5 = \{r_1, r_2\}$



b) Describe what the vertices will be in your flow network. Also mention the total number of vertices that will be in the graph.

We will have the following nodes:

– A source s

– A sink t

– One node per each applicant (n nodes)

– one node per each role (m nodes)

– one node per each project (x nodes)

This yields $n + m + x + 2$ total nodes.

c) Describe what your edges will be in your flow network (including their capacities).

The graph will have the following edges:

– An edge from s to each applicant node with capacity 1

– An edge from each a_i to each r_j node where $r_j \in q_i$ with capacity 1

– An edge from each r_i node to each project node p_j with capacity 1

– An edge from each project node p_i to t with capacity m.

d) Explain how you will use the max flow through the network to determine the solution to the problem (i.e. how will you decide when to return true vs. false?). You do not need to prove correctness.

To staff all projects we need to have one applicant per role per project, so we need $x \cdot m$ total flow through the network. So we will run Ford-Fulkerson, then check if the flow is $x \cdot m$.

(1 ESNU) **Question 3: True/False - SAT and NP-Completeness**

Suppose we have two problems that we call A and B. Suppose that we additionally have the following:

– A polynomial time algorithm to verify A

– A polynomial time algorithm to verify B

– A polynomial time reduction from A to SAT

– A polynomial time reduction from SAT to B

For each statement below, indicate whether it is true or false. "True" should only be selected if the statement is always true. "False" should be selected if the statement is either sometimes or always false. Give a 1–2 sentence explanation for each.

a) If there is a polynomial time algorithm to solve A then $P = NP$.

True [✓ False]

Explanation:

All problems in NP are known to reduce to SAT, including the ones that also belong to P. So we already know of algorithms with polynomial time solutions that reduce to SAT.

b) If there is a polynomial time algorithm to solve B then $P = NP$.

[✓ True] False

Explanation:

With the reduction, this creates a polynomial time solution to SAT, meaning P=NP

c) If $P \neq NP$ then there does not exist a polynomial time algorithm to solve A.

True [✓ False]

Explanation:

Same reasoning as above. We already know of some polynomial time algorithms which reduce to SAT.

d) If $P \neq NP$ then there does not exist a polynomial time algorithm to solve B.

[✓ True]

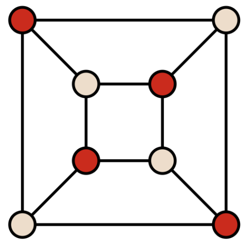
False

Explanation:

If there did exist a polynomial time algorithm to solve B then that would mean there is a polynomial time algorithm to solve SAT, meaning $P = NP$.

(4 ESNUs) **Question 4: Independent Set - SAT Encoding**

Given an undirected graph with vertices V and edges E, an *independent set* is a subset of the vertices so that no two vertices in the subset are connected by an edge. For example, the red vertices in the graph below form an independent set of size 4.



Given such a graph and a number k, does the graph have an independent set of size $\geq k$? In this problem, you will use SAT to solve the independent set question. Make sure that the number of variables and clauses that you use is polynomial in the size of the input graph, though you will not be asked to prove this.

1. Define the Boolean variables that you will use in your encoding. Say what each variable is supposed to represent.

   Let x_v represent whether or not vertex v is taken in the independent set, for all $v \in V$.

2. Using your variables, what clauses can you write that represent the selected vertices forming an independent set?

   For every edge $\{u, v\} \in E$, create the clause $(\neg x_u \vee \neg x_v)$.

3. Using your variables, describe how you would encode the fact that there are at least k vertices in your independent set. If you use Tseitin transformations and/or circuits in your encoding, you can simply state them without describing them in detail. (The attached reference sheet has a list of some circuits that may be useful.)

Let n be the number of vertices. We would create $n-1$ adders, each operating on numbers that are $\lceil \log_2(n) \rceil$ bits long. These adders would compute $S = \sum_{v \in V} x_v$. (The maximum sum is n, which fits in $\lceil \log_2(n) \rceil$ bits.)

Finally, we would use a $\lceil \log_2(n) \rceil$-bit comparator to determine if $S \geq k$. We would use Tseitin transformations to encode these circuits as clauses, and lastly add a clause containing just the variable associated to the output of the comparator, saying it must be true.

(Sufficient response for E: We would create adders to add $S = \sum_{v \in V} x_v$. We would create a comparator to determine if $S \geq k$. We would use Tseitin transformations to encode these circuits as clauses, and lastly add a clause containing just the variable associated to the output of the comparator, saying it must be true.)

(1 ESNU) **Question 5: Asymptotic Analysis**

Each subproblem below contains psuedocode for a different algorithm which solves the same problem. For each, give the worst case asymptotic running time of the algorithm. Provide a 1-2 sentence explanation for how you arrived at your conclusion.

In each case, the input to the algorithm is an array list of positive integers that has already been sorted in ascending order (so smallest to largest) and has length at least 2 elements. The output will be the sum of the largest 2 elements.

We assume the list is 0-indexes (meaning the first element appears at index 0)

a)

```
public int sum2Largest(list myList){
    int n = myList.length();
    int first = 0;
    int second = 0;
    for(int i = 0; i < n; i++){
        int x = myList[i];
        if (x >= largest){
            second = first;
            first = x;
        }
        else if (x >= second){
            second = x;
        }
    }
    return first + second;
}
```

Running Time: $O(n)$

Explanation:

there is a for loop over the list with constant work done per iteration.

b)
```
public int sum2Largest(list myList){
    int n = myList.length();
    return myList[n-1] + myList[n-2]
}
```

Running Time: $O(1)$

Explanation:

All that's required is indexing into the list. Since this is an array list this requires constant time.

c)
```
public int sum2Largest(list myList){
    if (myList.length() ==1){
    return myList[0];
    }
    if (myList.length() == 2){
        return myList[0] + myList[1];
    }
    int n = myList.length();
    list nextList = new ArrayList();
    for(int i = n/2; i < n; i++){
        nextList.add(myList[i]);
    }
    return sum3Largest(nextList);
}
```

Running Time: $O(n)$

Explanation:

At each stage of the recursion half of the list is copied. This gives the recurrence $T(n) = T(\frac{n}{2}) + O(n)$. By the Master Theorem this gives a running time of $O(n)$.

(1 ESNU) **Question 6: Unique Minimum Spanning Trees - Graphs**

Suppose we have a weighted and undirected graph $G = (V, E)$ with the property that all edges have distinct positive weights (i.e. no two edges have equal weight, all edges have positive weights).

Show that the following is always true of G:

The minimum spanning tree of G is unique (i.e. there are no two different minimum spanning trees with the same total cost).

Justification:

Suppose, towards contradiction, that there were two different MSTs called A and B of equal cost. Consider some edge e_a which belongs to A but not B. Define a cut so that e_a is the only crossing edge in tree A. Because e_a does not belong to B, it must have a different edge (or edges) crossing this cut. This means that it cannot be that both of A and B use the smallest crossing edge, and so at least one of them violates the cut theorem. This contradicts the assumption that both of A and B are MSTs.

(1 ESNU) **Question 7: Valley Finder - Divide and Conquer**

Note: This question was copied from Practice Quiz 1 to indicate we intend to put one of a similar style on the final.

For this problem, you will write an algorithm that takes as input an array of doubles that's length n where $n > 4$, where this array has the following properties:

– index 0 of the array contains the value 0

– index 1 of the array contains the value -1

– index $n-1$ of the array contains the value 0

– index $n-2$ of the array contains the value -1

– no two consecutive indices of the array contain the same value.

Your goal is to write an algorithm FindValley that outputs an index i such that i is a local minimum (meaning that the value at index $i-1$ is larger than the value at index i, which is smaller than the value at index $i+1$).

Note that the properties of the input array guarantee that there is a local minimum at some index.

a) Design an algorithm for FindValley that performs only $O(\log n)$ comparison operations (i.e. a $>, <, \geq, \leq$ operation on elements in the array).

Input: an array of doubles that we call arr, an index i (initially 0), and an index j (initially $n-1$) where $arr[i] > arr[i+1]$ and $arr[j-1] < arr[j]$.
Goal: find a local minimum between index i and index j.
Base case: if the range from i to j has 5 or fewer elements, then one of those 5 must be a local minimum. For each element, check if it is less than both of its neighbors. Return the first element seen with this property.
Divide: Query the middle index of the range (call this $mid = \frac{i+j}{2}$), and look up $arr[mid-1], arr[mid], arr[mid+1]$. If $arr[mid-1] > arr[mid] < arr[mid+1]$ then $arr[mid]$ is a local minimum, so return mid. Otherwise we will be in one of three cases:
If $arr[mid-1] > arr[mid] > arr[mid+1]$ (i.e. the middle three indices are decreasing), then there is guaranteed to be a local minimum between mid and j.
If $arr[mid-1] < arr[mid] < arr[mid+1]$ (i.e. the middle three indices are increasing), then there is guaranteed to be a local minimum between i and mid.
If $arr[mid-1] < arr[mid] > arr[mid+1]$ (i.e. the middle the middle index is a local maximum), then there is guaranteed to be a local minimum between both i and mid as well as mid and j, so we can just pick the subproblem from i to mid.
Conquer: Recursively solve the problem on the input identified above
Combine: return the result of the recursive call.

b) Write out the recurrence that describes its number of comparisons and indicate how that yields the required bound on the number of tests.

$T(n) = T(\frac{n}{2}) + O(1)$
we can apply the master theorem with $a = 1, b = 2, k = 0$. This means $log_b(a) = 0 = k$ giving us a running time $n^k \log(n) = n^0 \log(n) = \log(n)$.

(1 ESNU) **Question 8: No Parents - Dynamic Programming**
Note: This question was copied from Practice Quiz 2 to indicate we intend to put one of a similar style on the final.

Suppose you are given an integer array of length n which represents a complete binary tree T ("complete" meaning every node either has 2 children or it is a leaf) such that the root of the tree is at index 1, it's left child is at index 2, its right child is at index 3, etc. Let v_i represent the value of the node found at index i. Your goal is to find a subset U of vertices from T such that:

– U does not contain any node and its parent

– The sum of the values of all nodes in U is maximized.

For convenience, you may assume you have access to constant-time subroutines $L(i)$ and $R(i)$ which return the index of the left child of node i and the right child of node i (respectively) if it exists, and -1 otherwise (and so $L(-1) = R(-1) = -1$).

The questions below work towards dynamic programming algorithm to solve this problem.

a) Write a recursive definition for $\mathsf{OPT}(i)$, which should represent the maximum sum of U you can get for the subtree rooted at node i.

$$\mathsf{OPT}(i) = \begin{cases} 0 \text{ if } i = -1 \\ \max \begin{cases} \mathsf{OPT}(L(i)) + \mathsf{OPT}(R(i)) \\ v_i + \mathsf{OPT}(L(L(i))) + \mathsf{OPT}(L(R(i))) + \mathsf{OPT}(R(L(i))) + \mathsf{OPT}(R(R(i))) \end{cases} \end{cases}$$

idea: Either the solution contains the root or it doesn't if it contains the root then the solution is the value of the root (v_i) plus the sum of the maximum solutions for each of the root's granchildren (ensuring we skip adjacent nodes). If it does not contain the root then it will be sum of the maximum solutions for each of the root's children.

b) How would you store the solutions to the subproblems that you encounter? How large is your storage?

in a 1-dimensional array of length n. Or equivalently, from leaves up to the root.

c) For an efficient iterative (i.e. bottom-up) dynamic programming solution, in what order would you evaluate these subproblems?

In descending order of index (so from n down to 1).

d) What would be the running time of your iterative algorithm?

$\Theta(n)$

Reference

Nothing written on this page will be graded.

Logs

$$x^{\log_x(n)} = n$$
$$\log_a(b^c) = c\log_a(b)$$
$$a^{\log_b(c)} = c^{log_b(a)}$$
$$\log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Asymptotic Notation

$f(n)$ is $O(g(n))$ provided that after some input size n_0, $f(n) \leq c \cdot g(n)$ for some constant c.

$f(n)$ is $\Omega(g(n))$ provided that after some input size n_0, $f(n) \geq c \cdot g(n)$ for some constant c.

$f(n)$ is $\Theta(g(n))$ provided that $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Master Theorem

Suppose that $T(n) = aT(\frac{n}{b}) + O(n^k)$ for $n > b$. Then:

– if $a < b^k$ then $T(n)$ is $O(n^k)$

– if $a = b^k$ then $T(n)$ is $O(n^k \log n)$

– if $a > b^k$ then $T(n)$ is $O(n^{\log_b a})$

Proposer Optimality / Receiver Pessimality

A pair (p, r) is a valid pair if there is some stable matching where they are matched together

Proposer Optimality: Every proposer is matched with their most preferred valid pair.

Receiver Pessimality: Every receiver is matched with their least preferred valid pair.

Graph Algorithm Running Times

n is the number of nodes, m is the number of edges

Breadth-first search: $O(n + m)$

Depth-first search: $O(n + m)$

Kruskal's algorithm: $O(n \log n)$

Prim's algorithm: $O(n^2)$ or $O(m + n\log n)$.

Dijkstra's algorithm: $O(n^2)$ or $O(m + n\log n)$

Building blocks for SAT encodings

Tseitin transformation: Any expression of the form $a \iff (b\, R\, c)$ can be equivalently rewritten as $O(1)$ clauses, where R is any binary Boolean operation (like AND, OR, XOR, etc.).

Circuits: Given integers encoded in binary, there are compact logical circuits that can perform mathematical operations. (The size is polynomial in the number of bits of the input integers.) Tseitin transformations let you use these in SAT encodings.

– Adder: adds two numbers

– Subtractor: subtracts two numbers

– Multiplier: multiplies two numbers

– Comparator: determines if one number is $\geq$ another number

Scratch Work

Nothing written on this page will be graded.