CS170 Final Review Session 1

Divide and Conquer

Problem 12. [Algorithm design] (11 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):

1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax(_____).
4. Set $y :=$ _____.
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ _____.
6. Return $\max(x, y, z)$.

Four Cases

1. Index i and j are both in left half of array
2. Index i and j are both in right half of array
3. Index i is in the left half and index j is in the right half
4. Index j is in the left half and index i is in the right half

Four Cases

1. Index i and j are both in left half of array
2. Index i and j are both in right half of array
3. Index i is in the left half and index j is in the right half
4. ~~Index j is in the left half and index i is in the right half~~

~~Four~~ Three Cases

1. Index i and j are both in left half of array
2. Index i and j are both in right half of array
3. Index i is in the left half and index j is in the right half
4. ~~Index j is in the left half and index i is in the right half~~

Problem 12. [Algorithm design] (11 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):

1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax(_____).
4. Set $y :=$ _____.
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ _____.
6. Return $\max(x, y, z)$.

Problem 12. [Algorithm design] (11 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):

1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax(`A[0,...,k-1]`).
4. Set $y :=$ _____.
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ _____.
6. Return $\max(x, y, z)$.

Problem 12. [Algorithm design] (11 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):

1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax(`A[0,...,k-1]`).
4. Set $y :=$ `FindMax(A[k,...,n-1])` .
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ _____ .
6. Return $\max(x, y, z)$.

Problem 12. [Algorithm design] (11 points)

We are given an array $A[0..n-1]$, where $n > 1$ and all array elements are non-negative integers. Our goal is to find the maximum value of $A[i] + A[j]^2$, where the indices i, j range over all values such that $0 \leq i < j < n$.

Fill in the blanks below to produce an efficient algorithm that correctly solves this problem.

FindMax($A[0..n-1]$):

1. If $n \leq 1$, return $-\infty$.
2. Let $k := \lfloor n/2 \rfloor$.
3. Set $x :=$ FindMax(`A[0,...,k-1]`).
4. Set $y :=$ `FindMax(A[k,...,n-1])` .
5. Set $z := \max(A[0], \ldots, A[k-1]) +$ `max(A[k]^2,...,A[n-1]^2)` .
6. Return $\max(x, y, z)$.

Fixed Points

Given a sorted array A[1,...,n] of distinct positive integers find a fixed point or return -1 if there are none. That is, find an index i such that A[i] = i.

Note: the naive solution is linear time - try to do better than this!

Ideas

- If A[i] > i, what can we say about A[i-1] and A[i+1]?
  - A[i+1] > i+1 A[i] increases by at least one
- If A[i] < i, what can we say about A[i-1] and A[i+1]?
  - A[i-1] < i-1 A[i] decreases by at least one
- What algorithm does this look like?
  - Binary Search!

Solution

```
procedure FixedPoint(A[1,...,n]):

    If n = 0: return -1

    Set k = floor(n/2)

    If A[k] = k: return k

    If A[k] < k: return FixedPoint(A[k+1,...,n])

    If A[k] > k: return FixedPoint(A[1,...,k])
```

What's the runtime?

Classic divide and conquer: 2 subproblems, each of half size

Linear time to combine solution

Master Theorem!

$$T(n) = 2T(n/2) + O(n)$$

What's the runtime?

Classic divide and conquer: 2 subproblems, each of half size

Linear time to combine solution

Master Theorem!

$$T(n) = 2T(n/2) + O(n)$$

$$\Theta(n \log n)$$

Hadamard Matrix Multiplication

The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $\begin{bmatrix} 1 \end{bmatrix}$
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Main Idea

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right] \qquad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

Split the problem in half:

$$H_k v = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right] \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$= \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

What's the runtime?

$$H_k v = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

Linear time to compute (v1 + v2) and (v1 - v2)

What's the runtime?

$$H_k v = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

Linear time to compute (v1 + v2) and (v1 - v2)

Classic divide and conquer: 2 subproblems, each of half size

What's the runtime?

$$H_k v = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$$

Linear time to compute (v1 + v2) and (v1 - v2)

Classic divide and conquer: 2 subproblems, each of half size

Master Theorem!

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

$$T(n) = O(n \log n)$$

Dynamic Programming

DP - Main Idea

- **Subproblem:**
  - Should resemble the overall problem
  - Think of the minimum # of the states to uniquely define your problem
  - Do your subproblems cover every possible case?
- **Recurrence Relation:**
  - Case-work! Think about the subproblems that the current one depends on, and how to combine them together (e.g. max/min/addition/any function)
  - Think recursion and use the recursive leap of faith
  - Establish order to solve subproblems (make sure that the subproblems you depend on are solved *before* your current one)
- **Base Case(s):**
  - Figure out what the first subproblems are (this should be easy once you have an order!)

DP - Pseudocode, Proof, Complexity

- **For pseudocode:**
  - Convert your subproblems into an n-dimensional array with each entry representing a subproblem (e.g. 2 variables in your subproblem corresponds to a 2D matrix)
  - Now fill in your n-dimensional array with their values according to your base case, determined order and recurrence relation
- **For proof of correctness:**
  - Use strong induction
  - Should follow easily from base cases and recurrence relation
- **For complexity:**
  - Runtime upper-bounded by # subproblems x max work per subproblem
  - Space upper-bounded by # subproblems and can be optimized by storing less array entries at once

A Sisyphean Task

(★★★ level) A Sisyphean Task

Suppose that you have n boulders, each with a positive integer weight w_i. You'd like to determine if there is any set of boulders that together weight exactly k pounds. You may want to review the solution to the Knapsack Problem for inspiration.

Let's just do subproblem, recurrence relation, base case(s), and runtime

A Sisyphean Task

- **Subproblem:**

A Sisyphean Task

- **Subproblem:**
  - Need to keep track of index into boulder array and the sum so far
  - Let S[i, j] = true if there is some subset of w_1 .. w_i that sums to j
  - Final result is S[n, k]
- **Recurrence Relation:**

A Sisyphean Task

- **Subproblem:**
  - Need to keep track of index into boulder array and the sum so far
  - Let S[i, j] = true if there is some subset of w_1 .. w_i that sums to j
  - Final result is S[n, k]
- **Recurrence Relation:**
  - Either you include the i-th boulder or not
  - S[i, j] = S[i - 1, j - w_i] or S[i - 1, j]
- **Base Cases:**

A Sisyphean Task

- **Subproblem:**
  - Need to keep track of index into boulder array and the sum so far
  - Let S[i, j] = true if there is some subset of w_1 .. w_i that sums to j
  - Final result is S[n, k]
- **Recurrence Relation:**
  - Either you include the i-th boulder or not
  - S[i, j] = S[i - 1, j - w_i] or S[i - 1, j]
- **Base Cases:**
  - Order should be increasing values of i and j
  - S[0, 0] = true, S[i, j] = false for j < 0, S[0, j] = false for j > 0
- **Runtime:**

A Sisyphean Task

- **Subproblem:**
  - Need to keep track of index into boulder array and the sum so far
  - Let S[i, j] = true if there is some subset of w_1 .. w_i that sums to j
  - Final result is S[n, k]
- **Recurrence Relation:**
  - Either you include the i-th boulder or not
  - S[i, j] = S[i - 1, j - w_i] or S[i - 1, j]
- **Base Cases:**
  - Order should be increasing values of i and j
  - S[0, 0] = true, S[i, j] = false for j < 0, S[0, j] = false for j > 0
- **Runtime:**
  - # subproblems = nk, work per subproblem = constant
  - Thus runtime is O(nk)

Solution

Main idea: This is very much like the knapsack problem, except we want our items to sum to *exactly* k instead of being less than or equal to k.

Let $S(i,k')$ be true if and only if there is some subset of $W[1..i]$ that sums to k'. Then, we have the following recurrence relation:

$$S(i,k') = S(i-1,k') \vee S(i-1,k'-w_i)$$

Intuitively, this comes from the observation that a subset of $W[1..i]$ summing to k' either includes a_i or it doesn't. If it includes a_i, then we are left looking for a subset of $W[1..i-1]$ summing to $k'-w_i$. If it doesn't include a_i, then this means we need a subset of $W[1..i-1]$ that sums to k'. Thus, $S(i,k')$ is true if and only if at least one of $S(i-1,k')$ and $S(i-1,k'-w_i)$ is true. Our base cases are:

- $S(0,0) = true$ because the empty set sums to 0.
- $S(0,k') = false$ for $k' > 0$.
- $S(i,k') = false$ for $k' < 0$.

The final answer we are looking for is $S(n,k)$. This gives us nk subproblems, each of which takes cosntant time to solve.

Longest Palindrome Subsequence

(★★★ level) Longest Palindrome Subsequence

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For example, "bob" and "racecar" are palindromes, but "cat" is not. Devise an algorithm that takes a sequence $x[1..n]$ and returns the length of the longest palindromic subsequence. Its running time should be $O(n^2)$. (Recall that a subsequence need not be consecutive, e.g., "aba" is the longest palindromic subsequence of "anbma".)

Let's just do subproblem, recurrence relation, base case(s), and runtime

Longest Palindrome Subsequence

- **Subproblem:**

Longest Palindrome Subsequence

- **Subproblem:**
  - Need two indices into the sequence because a palindromic subsequence can occur in any contiguous subset of the sequence
  - Let P[i, j] = length of longest palindromic subsequence of x[i … j]
  - Final result is P[1, n]
- **Recurrence Relation:**

Longest Palindrome Subsequence

- **Subproblem:**
  - Need two indices into the sequence because a palindromic subsequence can occur in any contiguous subset of the sequence
  - Let P[i, j] = length of longest palindromic subsequence of x[i … j]
  - Final result is P[1, n]
- **Recurrence Relation:**
  - Two cases: End characters match or don’t match (if don’t match, split into two small sub-cases)
  - P[i, j] = …
    - If x[i] == x[j], P[i + 1, j - 1] + 2
    - Else, max(P[i + 1, j], P[i, j - 1])

Longest Palindromic Subsequence

- **Base Case(s):**

Longest Palindromic Subsequence

- **Base Case(s):**
  - Order is to work outwards from single elements in x
  - P[i, i] = 1, P[i, i - 1] = 0
  - E.g. considering x = “aa”, P[1, 2] = 2 + P[2, 1] = 2 + 0 = 2
- **Runtime:**

Longest Palindromic Subsequence

- **Base Case(s):**
  - Order is to work outwards from single elements in x
  - P[i, i] = 1, P[i, i - 1] = 0
  - E.g. considering x = “aa”, P[1, 2] = 2 + P[2, 1] = 2 + 0 = 2
- **Runtime:**
  - # subproblems = n^2, work per subproblem = constant
  - Thus runtime is $O(n^2)$ as required

Solution

Let $x = x_1 \dots x_n$ be the string, and let $P[i, j]$ be the length of the longest palindrome subsequence of $x_i \dots x_j$. Then we have the following recurrence:

$$P[i,j] = \begin{cases} P[i+1, j-1] + 2 & \text{if } x_i == x_j \\ \max\{P[i+1, j], P[i, j-1]\} & \text{otherwise} \end{cases}$$

With base cases:

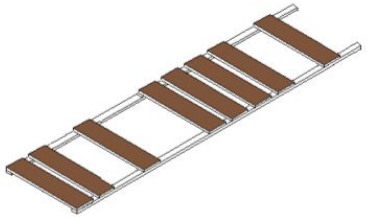
$$P[i,i] = 1 \qquad \forall i = 1 \dots n$$
$$P[i,i+1] = 2(x_i == x_{i+1}) \qquad \forall i = 1 \dots n-1$$

Once we have computed $P[i, j]$ for $i = 1 \dots n, j = i \dots n$, we can then output $P[1,n]$ – this is the length of the longest palindromic subsequence. One thing to be careful with is to make sure to compute $P[i, j]$ in the right order so that the entries $P[i+1, j-1], P[i+1, j], P[i, j-1]$ are computed before $P[i, j]$. They are several ways to do this; below we order based on decreasing order of i and increasing order of j respectively.

Bridge Hop

(★★★ level) Bridge Hop

You notice a bridge constructed of a single row of planks. Originally there had been n planks; unfortunately, some of them are now missing, and you're no longer sure if you can make it to the other side. For convenience, you define an array $V[1..n]$ so that $V[i]$ = TRUE iff the ith plank is present. You're at one side of the bridge, standing still; in other words, your *hop length* is 0 planks. Your bridge-hopping skills are as follows: with each hop, you can increase or decrease your hop length by 1, or keep it constant.



For example, the image above has planks at indices [1, 2, 4, 7, 8, 9, 10, 12], and you could get to the other side with the following hops: [0, 1, 2, 4, 7, 10, 12, 14].

You start at location 0, just before the first plank. Arriving at any location greater than n means you've successfully crossed. Due to your winged shoes, there is no maximum hop length. But you can only hop forward (hop length cannot be negative).

Devise an efficient algorithm to determine whether or not you can make it to the other side.

Let's just do subproblem, recurrence relation, base case(s), and runtime

Bridge Hop

- **Subproblem:**

Bridge Hop

- **Subproblem:**
  - Keep track of where we are on the bridge as well as our hop length
  - Let P[i, h] = true if we can arrive at plank i with our most recent hop length being h (hopped h planks to get to plank i)
  - Final result is ANY(P[n, i]) for all i in [0...n] (max hop length = n because we can only increase hop length by 1 each step)
- **Recurrence Relation:**

Bridge Hop

- **Subproblem:**
  - Keep track of where we are on the bridge as well as our hop length
  - Let P[i, h] = true if we can arrive at plank i with our most recent hop length being h (hopped h planks to get to plank i)
  - Final result is ANY(P[n, i]) for all i in [0...n] (max hop length = n because we can only increase hop length by 1 each step)
- **Recurrence Relation:**
  - Three cases: Increase, maintain, or decrease hop length for previous step
  - P[i, h] = V[i] and (P[i - h, h] or P[i - h, h - 1] or P[i - h, h + 1])
- **Base Case(s):**

Bridge Hop

- **Subproblem:**
  - Keep track of where we are on the bridge as well as our hop length
  - Let P[i, h] = true if we can arrive at plank i with our most recent hop length being h (hopped h planks to get to plank i)
  - Final result is ANY(P[n, i]) for all i in [0...n] (max hop length = n because we can only increase hop length by 1 each step)
- **Recurrence Relation:**
  - Three cases: Increase, maintain, or decrease hop length for previous step
  - P[i, h] = V[i] and (P[i - h, h] or P[i - h, h - 1] or P[i - h, h + 1])
- **Base Case(s):**
  - Order is in increasing order of i (any order for h)
  - P[0, 0] = true, P[i, h] = false for h > 0 and i <= 0, P[i, 0] = false for i > 0
- **Runtime:**

Bridge Hop

- **Subproblem:**
  - Keep track of where we are on the bridge as well as our hop length
  - Let P[i, h] = true if we can arrive at plank i with our most recent hop length being h (hopped h planks to get to plank i)
  - Final result is ANY(P[n, i]) for all i in [0...n] (max hop length = n because we can only increase hop length by 1 each step)
- **Recurrence Relation:**
  - Three cases: Increase, maintain, or decrease hop length for previous step
  - P[i, h] = V[i] and (P[i - h, h] or P[i - h, h - 1] or P[i - h, h + 1])
- **Base Case(s):**
  - Order is in increasing order of i (any order for h)
  - P[0, 0] = true, P[i, h] = false for h > 0 and i <= 0, P[i, 0] = false for i > 0
- **Runtime:**
  - # subproblems = n^2, work per subproblem = constant → $O(n^2)$ runtime

Solution

As in the previous problems, we have to decide what information to include in the subproblem definitions to make the recurrence easy to compute. We decide to define subproblems as $P(i,h)$, which is TRUE if we can get to location i with our last hop of length h and FALSE otherwise. Then, the recurrence is

$$P(i,h) = V[i] \wedge [P(i-h,h) \vee P(i-h,h-1) \vee P(i-h,h+1)]$$

because we can only land at location i with hop length h if there's a plank there, and our hop length on the previous step was $h-1$, h, or $h+1$.

Note that because our hop length can only increase by 1 at each step, the maximum hop length is n.

Note: It's also possible to do this problem in the other direction, with the recurrence answering the question "can we get to the other side from this location at this speed?"

Linear Programs (LP)

Linear Program

Formulating LPs

- Decision Variables
  - Things you have control over.
- Objective Function
  - Optimize a goal. Minimize or maximize.
- Constraints
  - Optimize objective while satisfying restrictions to variables.

Max Flow

- Given a directed graph *G = (V, E)*, send as many units of flow from source *s* to sink *t*
- Satisfying positive edge capacity constraints.
- The number of units send from *s* must be received by *t*.

$$\textbf{max} \quad \text{size}(f) = \sum_{(s,u)\in E} f_{su}$$

$$\text{s.t.} \quad \sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz} \quad \text{for all } u \in V$$

Flow conservation constraints.

$$0 \le f_e \le c_e \textbf{ for all } e \in E$$

Capacity constraints.

Duality

- Suppose we have a maximization LP with an optimal value of z^*. (Primal)
- How do the constraints affect z^*?
- Suppose we have an upper bound of the LP in z_u, $z^* \leq z_u$.
- We can minimize z_u such that $z^* = z_u$.
- The smallest upper bound must be the optimal solution to the primal LP.
- We can weight the constraints in such a way to minimize the upper bound. (Dual)
- These weights will upper bound the coefficients

Duality

Find the dual of the following primal LP:

$$\max -2x_1 + x_2$$
$$x_2 \leq x_1 + 1$$
$$x_1 + x_2 \leq 5$$
$$x_1, x_2 \geq 0$$

Duality

Rewriting and multiplying by *y* multipliers:

$$\max -2x_1 + x_2$$

y_1 $x_2 - x_1 \leq 1$

y_2 $x_1 + x_2 \leq 5$

$$x_1, x_2 \geq 0$$

Duality

Rewriting the multipliers and constraints:

$$y_1 (x_2 - x_1) + y_2 (x_1 + x_2) \leq y_1 + 5y_2$$

$$\max -2x_1 + x_2$$

$$x_2 \leq x_1 + 1$$

$$x_1 + x_2 \leq 5$$

$$x_1, x_2 \geq 0$$

Duality

Rewriting the multipliers and constraints:

$$y_1 (x_2 - x_1) + y_2 (x_1 + x_2) \leq y_1 + 5y_2$$

$$(-y_1 + y_2) x_1 + (y_1 + y_2) x_2 \leq y_1 + 5y_2$$

$$\max -2x_1 + x_2$$

$$x_2 \leq x_1 + 1$$

$$x_1 + x_2 \leq 5$$

$$x_1, x_2 \geq 0$$

Duality

Adding upper bound to primal objective

$$\max -2x_1 + x_2$$
$$x_2 \leq x_1 + 1$$
$$x_1 + x_2 \leq 5$$
$$x_1, x_2 \geq 0$$

$y_1 (x_2 - x_1) + y_2 (x_1 + x_2) \leq y_1 + 5y_2$

$(-y_1 + y_2) x_1 + (y_1 + y_2) x_2 \leq y_1 + 5y_2$

$-2 x_1 + x_2 \leq (-y_1 + y_2) x_1 + (y_1 + y_2) x_2 \leq y_1 + 5y_2$

Duality

Writing the dual and nonnegativity constraints

$$\max -2x_1 + x_2$$
$$x_2 \leq x_1 + 1$$
$$x_1 + x_2 \leq 5$$
$$x_1, x_2 \geq 0$$

$$-2\,x_1 + x_2 \leq (-y_1 + y_2)\,x_1 + (y_1 + y_2)\,x_2 \leq y_1 + 5y_2$$

Dual

$$\min\ y_1 + 5y_2$$
$$\text{s.t. } -y_1 + y_2 \geq -2$$
$$y_1 + y_2 \geq 1$$
$$y_1,\ y_2 \geq 0$$

Duality

Writing the dual and nonnegativity constraints

$$-2x_1 + x_2 \leq (-y_1 + y_2)x_1 + (y_1 + y_2)x_2 \leq y_1 + 5y_2$$

$$\max -2x_1 + x_2$$
$$x_2 \leq x_1 + 1$$
$$x_1 + x_2 \leq 5$$
$$x_1, x_2 \geq 0$$

Dual

$$\min y_1 + 5y_2$$
$$\text{s.t. } -y_1 + y_2 \geq -2$$
$$y_1 + y_2 \geq 1$$
$$y_1, y_2 \geq 0$$

Nonnegativity constraints are added so that the sign in the primal constraints do not get flipped.

Zero Sum Games (Domination)

Let the zero sum game payoff matrix for the row player be as follows.

Column

Row	A	B	C
D	1	2	-3
E	3	2	-2
F	-1	-2	2

(a) If the row player plays optimally, can you find the probability that row picks **D** without directly solving for the optimal strategy?

Zero Sum Games (Domination)

Let the zero sum game payoff matrix for the row player be as follows.

		Column		
		A	B	C
Row	D	1	2	-3
	E	3	2	-2
	F	-1	-2	2

(a) If the row player plays optimally, can you find the probability that row picks **D** without directly solving for the optimal strategy?

Zero!

Regardless of what the column player chooses to do, the row player always gets a higher payoff by picking E over D

Strategy E “dominates” strategy D

Zero Sum Games (Domination)

		Column		
		A	B	C
Row	D	1	2	-3
	E	3	2	-2
	F	-1	-2	2

(b) Then, similarly, what is the probability that the column player picks strategy **A**?

Zero Sum Games (Domination)

		Column		
		A	B	C
Row	D	1	2	3
	E	3	2	-2
	F	-1	-2	2

(b) Then, similarly, what is the probability that the column player picks strategy **A**?

We know that the row player will never play strategy **D**

Zero Sum Games (Domination)

		Column		
		A	B	C
Row	D	1	2	3
	E	3	2	-2
	F	-1	-2	2

(b) Then, similarly, what is the probability that the column player picks strategy **A**?

We know that the row player will never play strategy **D**

⇒ For column player, picking **B** is always better than picking **A** (Column player is trying to minimize payoff)

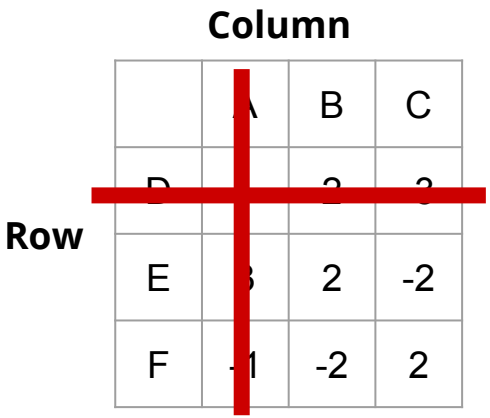
⇒ So column player will never pick **A**! (so probability is zero)

Zero Sum Games (Domination)

		Column		
		A	B	C
Row	D	1	2	-3
	E	3	2	-2
	F	-1	-2	2

(b) Given the answers to previous parts, what are the optimal strategies for both players?

Zero Sum Games (Domination)



(b) Given the answers to previous parts, what are the optimal strategies for both players and the optimal values?

The payoff matrix simplifies to a simple symmetric matrix. You can solve this directly using the LP formulation, but one can guess that playing 50/50 strategy is optimal.

Thus, optimal value of the game is zero (just like rock paper scissors!)

Breadth Requirements (Spring 2016 MT2)

UC Berkeley's newly formed College of Computer Science has several categories of breadth requirements. Each class is offered by a single department, and may match multiple categories. Students must take at least one class from each category, and at most k classes from each department. If a class matches multiple categories, you must choose which category you're using it to fulfill: no class can be used to fulfill more than one category.

Formally, there are m classes $c_1 ... c_m$, T departments $D_1 ... D_T$, and n categories of breadth requirements $G_1 ... G_n$. For some class c_i, let $H(c_i)$ be its department and $F(c_i)$ be the set of categories it can be used to fulfill. For example, you could have $H(c_{10}) = D_3$ and $F(c_{10}) = \{G_3, G_4, G_9\}$.

Given a subset of classes $c_1 ... c_p$, $G_1 ... G_n$, $D_1 ... D_T$, F, and H, you want to find whether the p classes satisfy all the breadth requirements subject to the above constraints.

(Formulate the problem above as an **LP**)

Breadth Requirements (Spring 2016 MT2)

- What are the decision variables?
- What is the objective function?
- What are the constraints?

Breadth Requirements (Spring 2016 MT2)

- What are the decision variables?
  - There are *m* classes, *T* departments, and *n* requirements.
  - Let x_{ij} be an indicator variable indicating if the *i*-th class is being used to satisfy the *j*-th requirement.
- What is the objective function?

- What are the constraints?

Breadth Requirements (Spring 2016 MT2)

- What are the decision variables?
  - There are *m* classes, *T* departments, and *n* requirements.
  - Let x_{ij} be an indicator variable indicating if the *i*-th class is being used to satisfy the *j*-th requirement.
- What is the objective function?
  - The question cares about feasibility rather than optimizing.
  - Obj Func: max 0
- What are the constraints?

Breadth Requirements (Spring 2016 MT2)

$$\forall j \in \{1..n\} : \sum_i x_{ij} \geq 1$$

For each breadth, at least one class is used to satisfy it.

Breadth Requirements (Spring 2016 MT2)

$$\forall j \in \{1..n\} : \sum_i x_{ij} \geq 1$$

For each breadth, at least one class is used to satisfy it.

$$\forall l \in \{1..T\} : \sum_{\{i|H(c_i)=D_l\}} \sum_j x_{ij} \leq k$$

For each department, at most *k* courses can be used to satisfy all requirements.

Breadth Requirements (Spring 2016 MT2)

$$\forall j \in \{1..n\} : \sum_{i} x_{ij} \geq 1$$

For each breadth, at least one class is used to satisfy it.

$$\forall l \in \{1..T\} : \sum_{\{i|H(c_i)=D_l\}} \sum_{j} x_{ij} \leq k$$

For each department, at most *k* courses can be used to satisfy any requirement.

$$\forall i \in \{1..m\} : \sum_{j} x_{ij} \leq 1$$

Each class can only satisfy at most one requirement.

Breadth Requirements (Spring 2016 MT2)

$$\forall j \in \{1..n\} : \sum_i x_{ij} \geq 1$$

For each breadth, at least one class is used to satisfy it.

$$\forall l \in \{1..T\} : \sum_{\{i|H(c_i)=D_l\}} \sum_j x_{ij} \leq k$$

For each department, at most *k* courses can be used to satisfy all requirements.

$$\forall i \in \{1..m\} : \sum_j x_{ij} \leq 1$$

Each class can only satisfy at most one requirement.

$$\forall i,j \in \{(i,j)|G_j \in F(c_i)\} : x_{ij} \leq 1$$
$$\forall i,j \in \{(i,j)|G_j \notin F(c_i)\} : x_{ij} \leq 0$$

We can only use classes to satisfy requirements that they can satisfy.

Breadth Requirements (Spring 2016 MT2)

$$\forall j \in \{1..n\} : \sum_i x_{ij} \geq 1$$

For each breadth, at least one class is used to satisfy it.

$$\forall l \in \{1..T\} : \sum_{\{i|H(c_i)=D_l\}} \sum_j x_{ij} \leq k$$

For each department, at most *k* courses can be used to satisfy all requirements.

$$\forall i \in \{1..m\} : \sum_j x_{ij} \leq 1$$

Each class can only satisfy at most one requirement.

$$\forall i, j \in \{(i,j)|G_j \in F(c_i)\} : x_{ij} \leq 1$$
$$\forall i, j \in \{(i,j)|G_j \notin F(c_i)\} : x_{ij} \leq 0$$

We can only use classes to satisfy requirements that they can satisfy.

$$\forall i, j : x_{ij} \geq 0$$

Lower bound for decision variables. Upper bound implicit from 3rd constraint.

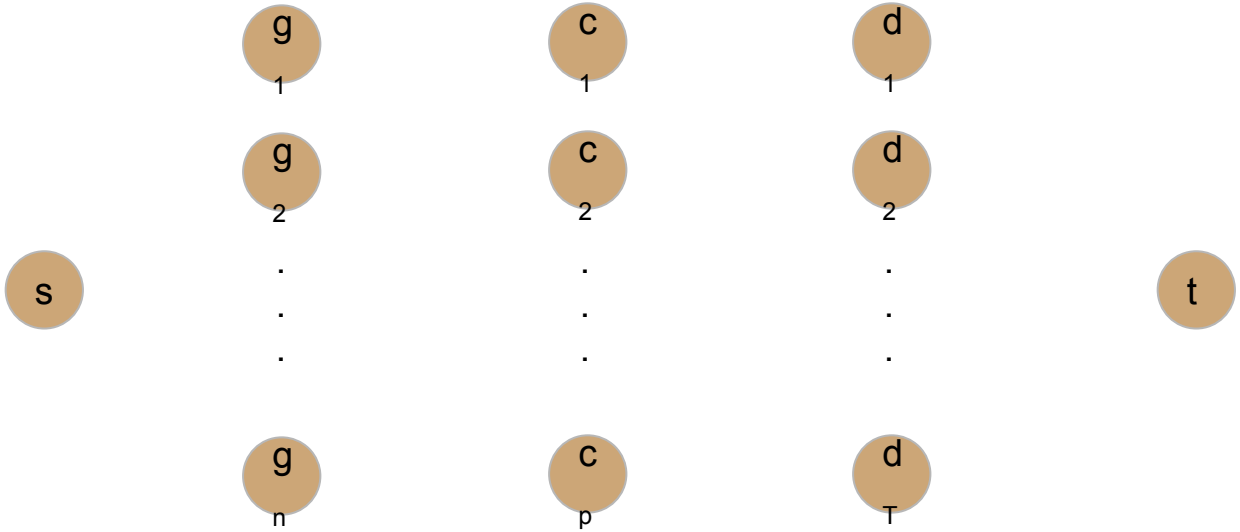
Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Reduce to max flow instance and solve in a way similar to bipartite matching.



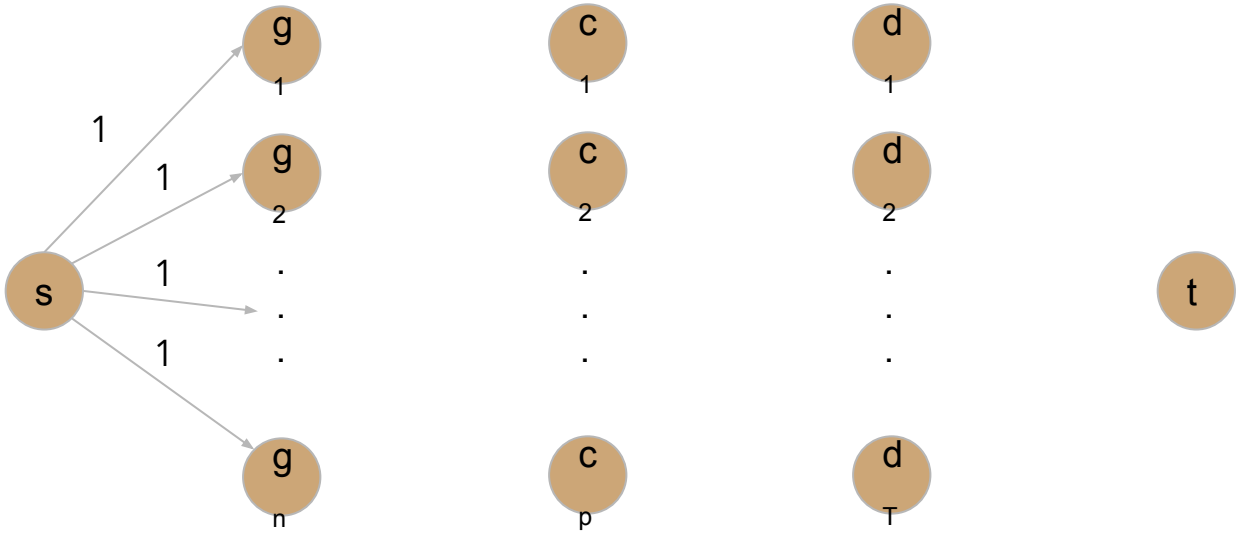
Graph consist of vertices of requirements, classes, and departments.

Add source and sink nodes.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Reduce to max flow instance and solve in a way similar to bipartite matching.



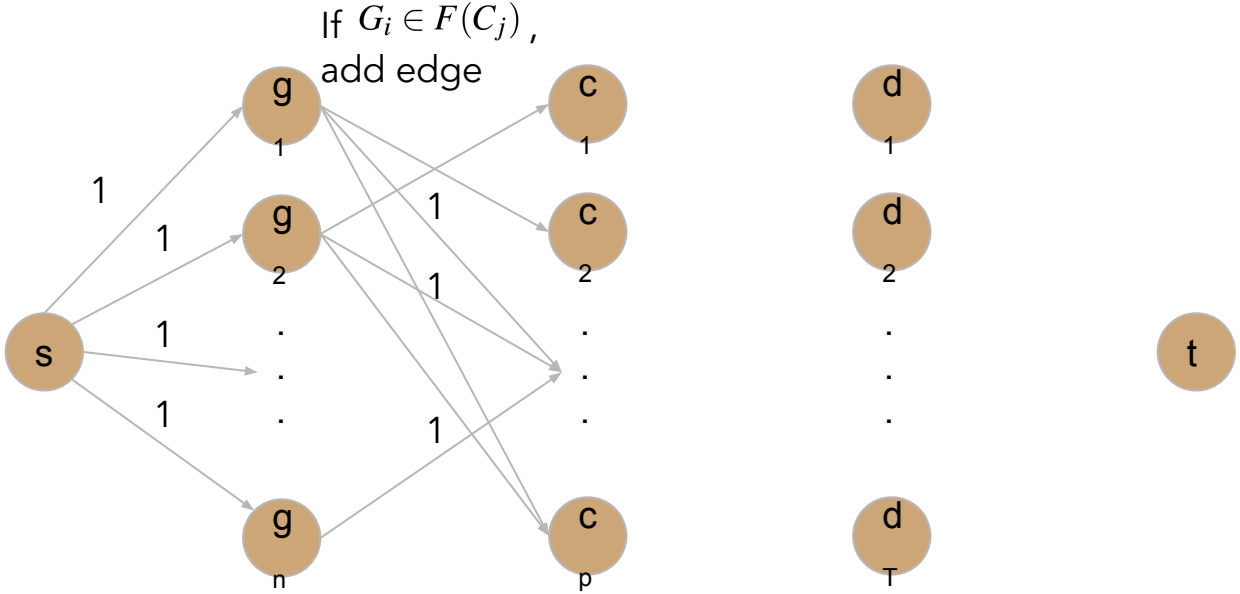
Need to satisfy each requirement.

Add edge (s, g_j) with edge weight 1.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Reduce to max flow instance and solve in a way similar to bipartite matching.



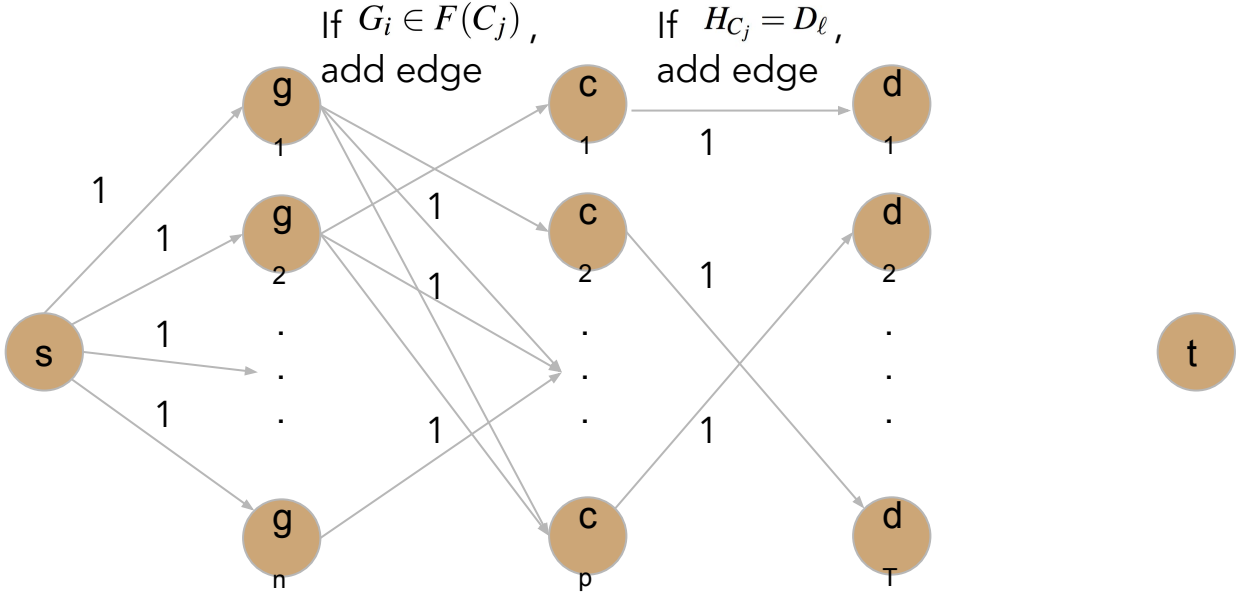
Add edge only if requirement can be satisfied with that class.

Each requirement is satisfied by a unique class.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Reduce to max flow instance and solve in a way similar to bipartite matching.



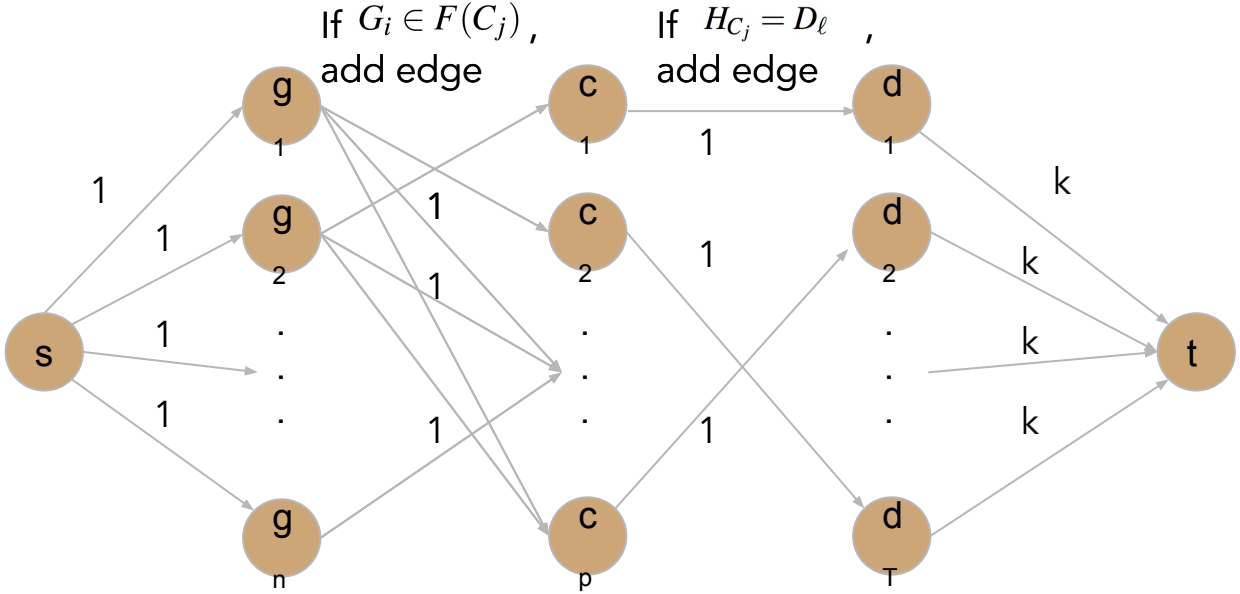
Add edge only if the class can satisfy the requirement.

Each class belongs to only one department. Thus edge capacity of 1.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

Reduce to max flow instance and solve in a way similar to bipartite matching.

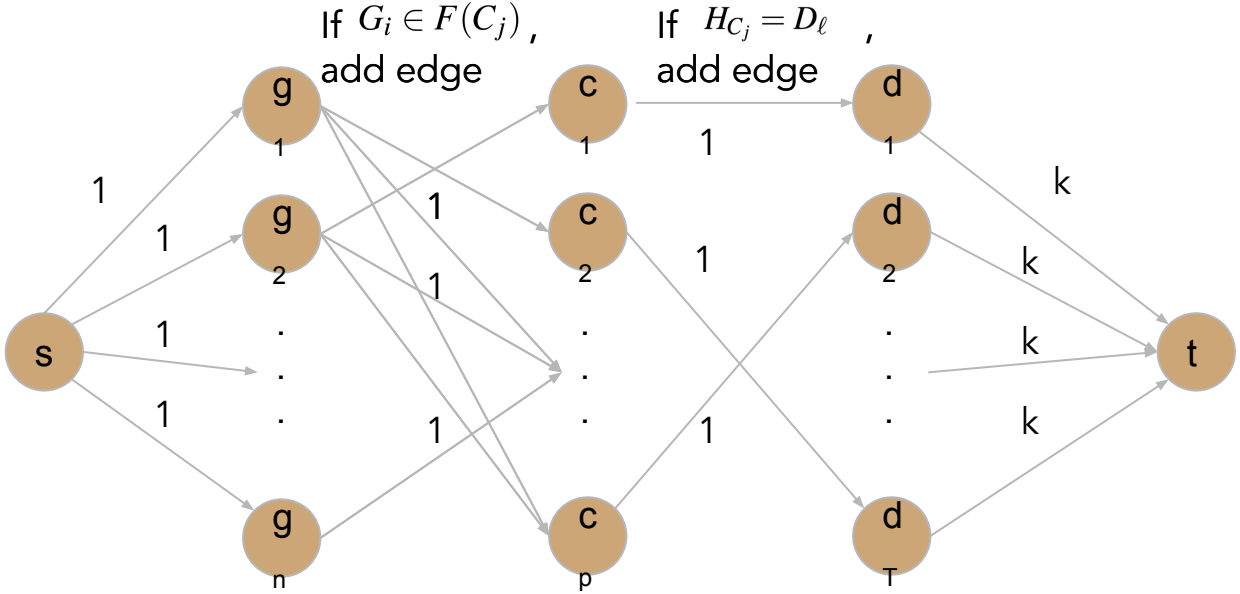


No more than k classes can be satisfy each requirement.

Breadth Requirements (Spring 2016 MT2)

(b) Find an efficient algorithm to determine whether the requirements are satisfied, and (if all requirements are satisfied) a mapping of which class contributes to which requirement.

There is a satisfying assignment if size of max flow = n (number of requirements).



The flow can go in the other direction.