CS 483 - Data Structures and Algorithm Analysis

Lecture VI: Chapter 5, part 2; Chapter 6, part 1

R. Paul Wiegand

George Mason University, Department of Computer Science

March 8, 2006

Outline

DFS & BFS Edge-Types

tree edge — Edge encountered by the search that leads to an as-yet unvisited node (DFS & BFS)

back edge — Edge leading to a previously visited vertex other than its immediate predecessor (DFS)

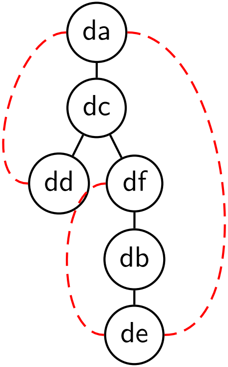
cross edge — Edge leading to a previously visited vertex other than its immediate predecessor (BFS)

DFS & BFS Edge-Types

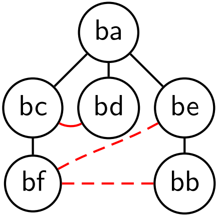
tree edge — Edge encountered by the search that leads to an as-yet unvisited node (DFS & BFS)

back edge — Edge leading to a previously visited vertex other than its immediate predecessor (DFS)

cross edge — Edge leading to a previously visited vertex other than its immediate predecessor (BFS)



DFS & BFS Edge-Types



tree edge — Edge encountered by the search that leads to an as-yet unvisited node (DFS & BFS)

back edge — Edge leading to a previously visited vertex other than its immediate predecessor (DFS)

cross edge — Edge leading to a previously visited vertex other than its immediate predecessor (BFS)

Directed Graphs: A Review

- A *directed graph* (digraph) is a graph with *directed edges*
- We can use the same representational constructs: adjacency matrices & adjacency lists
- But there are some differences from the undirected case:
  - Adjacency matrix need not be symmetric
  - An edge in the digraph has only one node in an adjacency list
- We can still use DFS & BFS to traverse such graphs, but the resulting search forest is often more complicated
- There are now four edge types

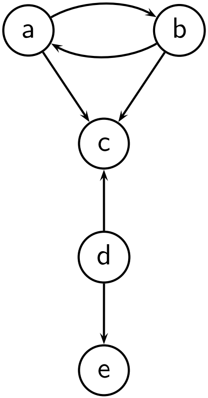
tree edge — Edge leading to an as-yet unvisited node

back edge — Edge leading from some vertex to a previously visited ancestor

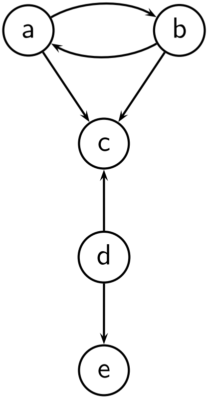
forward edge — Edge leading from a previously visited ancestor to some vertex

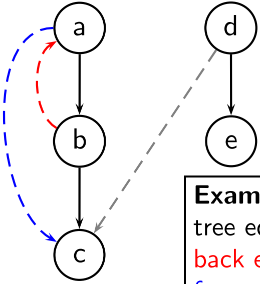
cross edge — Remaining edge types

Directed Graphs: More Review



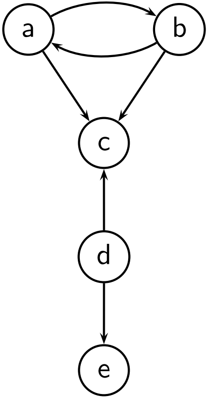
Directed Graphs: More Review

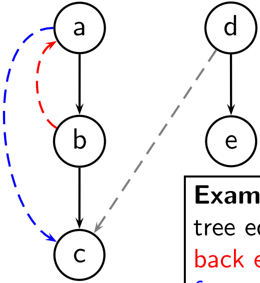




Example DFS forest:
tree edges
back edge
forward edge
cross edge

Directed Graphs: More Review



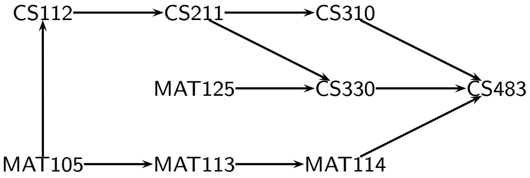


Example DFS forest:
tree edges
back edge
forward edge
cross edge

NOTE: A digraph with no back edges has no directed cycles. We call this a *directed acyclic graph* (DAG).

Representing Dependencies with DAGs

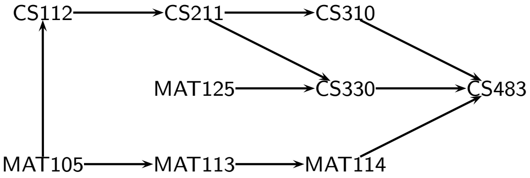
- Many real-world situations can be modeled with DAGs
- Consider problems involving dependencies (e.g., course pre-requisites):



Representing Dependencies with DAGs

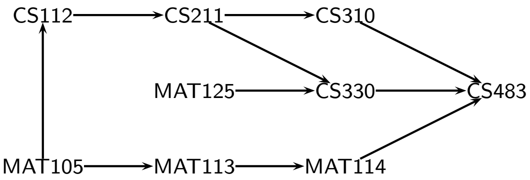
- Many real-world situations can be modeled with DAGs
- Consider problems involving dependencies (e.g., course pre-requisites):

- If you could take only one course at a time, what order would you choose?



Representing Dependencies with DAGs

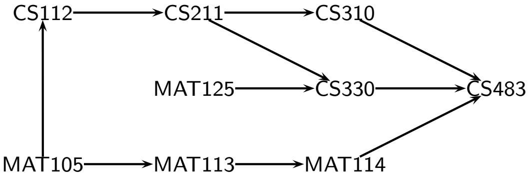
- Many real-world situations can be modeled with DAGs
- Consider problems involving dependencies (e.g., course pre-requisites):



- If you could take only one course at a time, what order would you choose?
- More generally: Order the vertices of a DAG such that for every edge, the vertex where the edge starts precedes the vertex where the edge ends?

Representing Dependencies with DAGs

- Many real-world situations can be modeled with DAGs
- Consider problems involving dependencies (e.g., course pre-requisites):



- If you could take only one course at a time, what order would you choose?
- More generally: Order the vertices of a DAG such that for every edge, the vertex where the edge starts precedes the vertex where the edge ends?
- This problem is called *topological sorting*

Two Algorithms to Sort Topologies

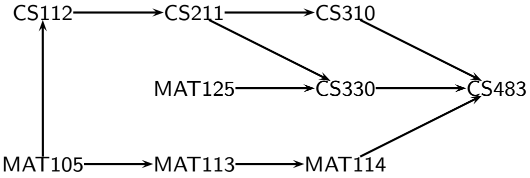
Algorithm 1

- Apply Depth-First Search
- Note the order in which the nodes become "*dead*" (popped off the traversal stack)
- Reverse the order; that is your answer
- Why does this work? When vertex v is popped off the stack, no vertex u with an edge (u, v) can be among the vertices popped of before v (otherwise (u, v) would be a back edge).

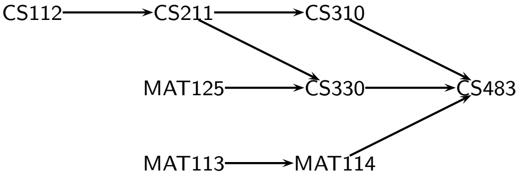
Algorithm 2

- Identify a *source* in the digraph (node with no in-coming edges)
- Break ties arbitrarily
- Record then delete the node, along with all edges from that node
- Repeat the process on the remaining subgraph
- When the graph is empty, you are done

Example On Algorithm 2

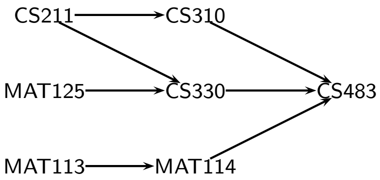


Example On Algorithm 2



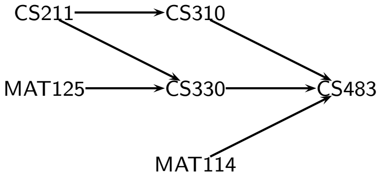
- MAT105

Example On Algorithm 2



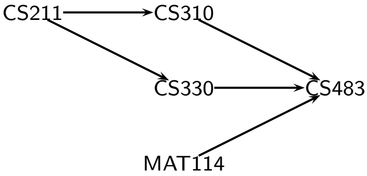
- MAT105
- CS112

Example On Algorithm 2



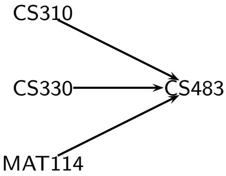
- MAT105
- CS112
- MAT113

Example On Algorithm 2



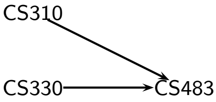
- MAT105
- CS112
- MAT113
- MAT125

Example On Algorithm 2



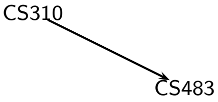
- MAT105
- CS112
- MAT113
- MAT125
- CS211

Example On Algorithm 2



- MAT105
- CS112
- MAT113
- MAT125
- CS211
- MAT114

Example On Algorithm 2



- MAT105
- CS112
- MAT113
- MAT125
- CS211
- MAT114
- CS330

Example On Algorithm 2

CS483

- MAT105
- CS112
- MAT113
- MAT125
- CS211
- MAT114
- CS330
- CS310

Example On Algorithm 2

- MAT105
- CS112
- MAT113
- MAT125
- CS211
- MAT114
- CS330
- CS310
- CS483

Reviewing Combinations & Permutations

What is the difference between a *combination* and a *permutation*?

combination —

permutation —

Reviewing Combinations & Permutations

What is the difference between a *combination* and a *permutation*?

combination — The number of ways of picking k *unordered* outcomes from n possibilities. We often write it as "*n choose k*". $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

permutation —

Reviewing Combinations & Permutations

What is the difference between a *combination* and a *permutation*?

combination — The number of ways of picking k *unordered* outcomes from n possibilities. We often write it as "*n choose k*". $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

permutation — A permutation is a rearrangement of the elements of an ordered list $\mathcal{S}$ into a one-to-one correspondence with $\mathcal{S}$ itself. ${}_nP_k = \frac{n!}{(n-k)!}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

```
for i ⟵ 0 to n − 2
   j ⟵ RandInt(i + 1, n − 1)
   Swap(A, i, j)
```

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{1, 2, 3, 4, 5, 6, 7, 8\}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

for $i \longleftarrow 0$ to $n-2$
  $j \longleftarrow$ RandInt$(i+1, n-1)$
  Swap(A, i, j)

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{4, 2, 3, 1, 5, 6, 7, 8\}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

for $i \longleftarrow 0$ to $n-2$
  $j \longleftarrow$ RandInt$(i+1, n-1)$
  Swap(A, i, j)

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{4, 7, 3, 1, 5, 6, 2, 8\}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

```
for i ⟵ 0 to n − 2
   j ⟵ RandInt(i + 1, n − 1)
   Swap(A, i, j)
```

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{4, 7, 1, 3, 5, 6, 2, 8\}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

for $i \longleftarrow 0$ to $n-2$
  $j \longleftarrow$ RandInt($i+1, n-1$)
  Swap(A, i, j)

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:
$\{4, 7, 1, 8, 5, 6, 2, 3\}$

Generating a Single Random Permutation

PermuteSet($A[0 \ldots n-1]$)

for $i \longleftarrow 0$ to $n-2$
  $j \longleftarrow \textsc{RandInt}(i+1, n-1)$
  $\textsc{Swap}(A, i, j)$

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{4, 7, 1, 8, 2, 6, 5, 3\}$

Generating a Single Random Permutation

PermuteSet$(A[0 \ldots n-1])$

```
for i ⟵ 0 to n − 2
   j ⟵ RandInt(i + 1, n − 1)
   Swap(A, i, j)
```

- Basic operation is RandInt
- The loop is $\Theta(n)$

- In general, items in a permutation can be anything
- We think of them as ordered *sets* $\{a_0, a_2, \ldots, a_{n-1}\}$
- But we'll talk about them as a lists of integers for simplicity

For example:

$\{4, 7, 1, 8, 2, 3, 5, 6\}$

Generating All Permutations

- Suppose we want to generate all permutations between 1 and n
- We can use Decrease-and-Conquer:
  - Given that the $n-1$ permutations are generated
  - We can generate the n^{th} permutations by inserting n at all possible n positions
  - We start adding right-to-left, then switch when a new perm is processed

Generating All Permutations

- Suppose we want to generate all permutations between 1 and n
- We can use Decrease-and-Conquer:
  - Given that the $n-1$ permutations are generated
  - We can generate the n^{th} permutations by inserting n at all possible n positions
  - We start adding right-to-left, then switch when a new perm is processed

Start			1			
Insert 2			12	21		
			right to left			
Insert 3	123	132	312	321	231	213
		right to left			left to right	

Generating All Permutations

- Suppose we want to generate all permutations between 1 and n
- We can use Decrease-and-Conquer:
  - Given that the $n-1$ permutations are generated
  - We can generate the n^{th} permutations by inserting n at all possible n positions
  - We start adding right-to-left, then switch when a new perm is processed

Start			1			
Insert 2			12	21		
			right to left			
Insert 3	123	132	312	321	231	213
		right to left		left to right		

Satisfies *minimal-change*: each permutation can be obtained from its predecessor by exchanging just two elements

Generating n^{th} Permutation

- We can get the same ordering of permutations of n elements without generating the smaller permutations
  - We associate a direction with each element in the permutation: $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$
  - A *mobile* component is one in which the arrow points to a smaller adjacent value (3 & 4 above, but not 1 & 2)

JohnsonTrotter(n)

Initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \cdots \overleftarrow{n}$
`while` there exists a mobile k `do`
  Find the largest mobile integer k
  Swap k and the adjacent integer its arrow points to
  Reverse the direction of all integers larger than k

Generating n^{th} Permutation

- We can get the same ordering of permutations of n elements without generating the smaller permutations
  - We associate a direction with each element in the permutation: $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$
  - A *mobile* component is one in which the arrow points to a smaller adjacent value (3 & 4 above, but not 1 & 2)

JohnsonTrotter(n)

Initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2}\ \cdots\ \overleftarrow{n}$
`while` there exists a mobile k `do`
  Find the largest mobile integer k
  Swap k and the adjacent integer its arrow points to
  Reverse the direction of all integers larger than k

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$

$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$

$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$

$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$

$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

Generating Subsets

- Given some universal set: $U = \{a_1, a_2, \cdots, a_n\}$, generate all possible subsets
- The set of all subsets is called a *power set*; there are 2^n of them

n	**subsets**							
0	$\emptyset$							
1	$\emptyset$	$\{a_1\}$						
2	$\emptyset$	$\{a_1\}$	$\{a_2\}$		$\{a_1, a_2\}$			
3	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

Generating Subsets

- Given some universal set: $U = \{a_1, a_2, \cdots, a_n\}$, generate all possible subsets
- The set of all subsets is called a *power set*; there are 2^n of them

n	**subsets**			
0	$\emptyset$			
1	$\emptyset$	$\{a_1\}$		
2	$\emptyset$	$\{a_1\}$ $\{a_2\}$	$\{a_1, a_2\}$	
3	$\emptyset$	$\{a_1\}$ $\{a_2\}$ $\{a_3\}$	$\{a_1, a_2\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

- Can represent a set as a binary string:

000	001	010	011	100	101	
$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	...

- Is there an equivalent to minimal-change algorithm here?

Generating Subsets

- Given some universal set: $U = \{a_1, a_2, \cdots, a_n\}$, generate all possible subsets
- The set of all subsets is called a *power set*; there are 2^n of them

n	**subsets**							
0	$\emptyset$							
1	$\emptyset$	$\{a_1\}$						
2	$\emptyset$	$\{a_1\}$	$\{a_2\}$		$\{a_1, a_2\}$			
3	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

- Can represent a set as a binary string:

000	001	010	011	100	101	
$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	...

- Is there an equivalent to minimal-change algorithm here?
  Yes: 000 001 011 010 110 111 101 100 (Gray code)

Fake-Coin Problem

- You are given n coins, one of which is fake (you don't know which)
- You are provided a balance scale to compare sets of coins
- What is an efficient method for identifying the coin?

Fake-Coin Problem

- You are given n coins, one of which is fake (you don't know which)
- You are provided a balance scale to compare sets of coins
- What is an efficient method for identifying the coin?

1. Hold one (or two) coins aside
2. Divide the remainder into two equal halves
3. If they balance, the fake has been set aside
4. Otherwise examine the lighter pile in the same way

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1,\ W(1) = 0 \qquad \in \Theta(\lg n)$$

Fake-Coin Problem

- You are given n coins, one of which is fake (you don't know which)
- You are provided a balance scale to compare sets of coins
- What is an efficient method for identifying the coin?

1. Hold one (or two) coins aside
2. Divide the remainder into two equal halves
3. If they balance, the fake has been set aside
4. Otherwise examine the lighter pile in the same way

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1,\ W(1) = 0 \qquad \in \Theta(\lg n)$$

But wait! This is not the most efficient way. What if you divided into *three* equal piles?

Multiplication á la Russe

- We want to compute the product of n and m, two positive integers
- But we only know how to add and multiple & divide by two
- If n is even, we can re-write: $n \cdot m = \frac{n}{2} \cdot 2m$
- If n is odd, we can re-write: $n \cdot m = \frac{n-1}{2} \cdot 2m + m$
- We can apply this method iteratively until $n = 1$

Multiplication á la Russe

- We want to compute the product of n and m, two positive integers
- But we only know how to add and multiple & divide by two
- If n is even, we can re-write: $n \cdot m = \frac{n}{2} \cdot 2m$
- If n is odd, we can re-write: $n \cdot m = \frac{n-1}{2} \cdot 2m + m$
- We can apply this method iteratively until $n = 1$

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3,250

Median and Selection

- The *selection problem*: Find the k^{th} smallest element in a list of n numbers (the k^{th} order statistics)
- Finding the *median* is a special case: $k = \lceil n/2 \rceil$
- Brute force: Sort, then select the k^{th} value in the list: $O(n \lg n)$
- But we can do better: Use the partitioning logic from QuickSort

Median and Selection

- The *selection problem*: Find the k^{th} smallest element in a list of n numbers (the k^{th} order statistics)
- Finding the *median* is a special case: $k = \lceil n/2 \rceil$
- Brute force: Sort, then select the k^{th} value in the list: $O(n \lg n)$
- But we can do better: Use the partitioning logic from QuickSort

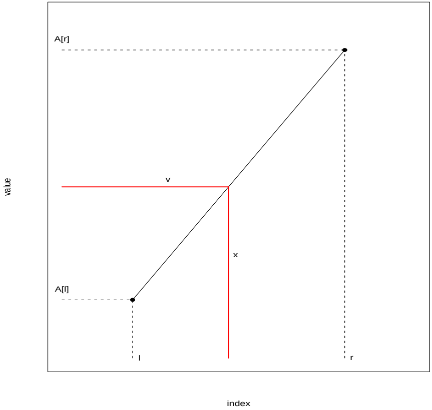
$$\boxed{\begin{array}{ccc} a_1 \cdots a_s & p & a_{s+1} \cdots a_n \\ \leq p & & \geq p \end{array}}$$

  - If $s = k$ then p solves the problem
  - If $s > k$ then the k^{th} smallest element in whole list is the k^{th} smallest element left-side sublist
  - If $s < k$ then the k^{th} smallest element in whole list is the $(k-s)^{th}$ smallest element right-side sublist
  - Average: $C(n) = C(n/2) + (n-1) \qquad \in \Theta(n)$

Interpolation Search

- Like BinarySearch, but more like a telephone book
- Rather than split the list in half, we interpolate the position based on the key value

  - $v :=$ search key value
  - $l :=$ left index
  - $r :=$ right index
  - $y = mx + b \Longrightarrow x = \frac{y-b}{m}$
  - $x = l + \left\lfloor \frac{(v-A[l])(r-l)}{A[r]-A[l]} \right\rfloor$
  - Average case: $O(\lg \lg n)$



Introduction to Transform & Conquer

- In many cases, one can transform a problem instance and solve the transformed problem
- Three variations of this idea are as follows:

instance simplicfication — Transform problem instance into a simpler or more convenient istance

representation change — Transform problem instance representations

problem reduction — Transform problem instance into an instance of different problem

Presorting

- Many questions about lists can be answered more easily when the list is already sorted
- The cost of the sort itself should be warranted

- Example: Element uniqueness

Presorting

- Many questions about lists can be answered more easily when the list is already sorted
- The cost of the sort itself should be warranted

- Example: Element uniqueness
  - Brute force: compare every element against every other element, $\Theta(n^2)$
  - Presort & scan: $T(n) = T_{\text{sort}}(n) + T_{\text{scan}} = \Theta(n \lg n) + \Theta(n) \in \Theta(n \lg n)$
- Example: Search

Presorting

- Many questions about lists can be answered more easily when the list is already sorted
- The cost of the sort itself should be warranted

- Example: Element uniqueness
  - Brute force: compare every element against every other element, $\Theta(n^2)$
  - Presort & scan: $T(n) = T_{\text{sort}}(n) + T_{\text{scan}} = \Theta(n \lg n) + \Theta(n) \in \Theta(n \lg n)$

- Example: Search
  - Brute force: linear search: $\Theta(n)$
  - Presort & binary search: $\Theta(n \lg n) + \Theta(\lg n) \in \Theta(n \lg n)$
  - Presorting does not help with one search, though perhaps it will with *many* searches on the same list

- Example: Computing a *mode* (most frequent value)

Presorting

- Many questions about lists can be answered more easily when the list is already sorted
- The cost of the sort itself should be warranted

- Example: Element uniqueness
  - Brute force: compare every element against every other element, $\Theta(n^2)$
  - Presort & scan: $T(n) = T_{\text{sort}}(n) + T_{\text{scan}} = \Theta(n \lg n) + \Theta(n) \in \Theta(n \lg n)$

- Example: Search
  - Brute force: linear search: $\Theta(n)$
  - Presort & binary search: $\Theta(n \lg n) + \Theta(\lg n) \in \Theta(n \lg n)$
  - Presorting does not help with one search, though perhaps it will with *many* searches on the same list

- Example: Computing a *mode* (most frequent value)
  - Brute force: scan list an store count in auxiliary list then scan auxiliary list for highest frequency, worst case time $\Theta(n^2)$
  - Presort, Longest-run: sort then scan through list looking for the longest run of a value, $\Theta(n \lg n)$

Gaussian Elimination

- Problem: Solve a system of n linear equations with n unknowns

$$\begin{array}{ccc}
a_{11}x_1 + a_{12}x_2 + & \cdots & + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + & \cdots & + a_{2n}x_n = b_2 \\
 & \vdots & \\
a_{n1}x_1 + a_{n2}x_2 + & \cdots & + a_{nn}x_n = b_n
\end{array}$$

- This can be written as $A\vec{x} = \vec{b}$
- Gaussian Elimination first asks us to transform the problem to a different one, one that has the same solution: $A'\vec{x} = \vec{b}'$
- The transformation yields a matrix with all zeros below its main diagonal:

$$A' = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & \ddots & & \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix}, \quad \vec{b}' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ v'_n \end{bmatrix}$$

Gaussian Elimination

- Problem: Solve a system of n linear equations with n unknowns

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + &\cdots + a_{1n}x_n = b_1\\
a_{21}x_1 + a_{22}x_2 + &\cdots + a_{2n}x_n = b_2\\
&\vdots\\
a_{n1}x_1 + a_{n2}x_2 + &\cdots + a_{nn}x_n = b_n
\end{aligned}$$

- This can be written as $A\vec{x} = \vec{b}$
- Gaussian Elimination first asks us to transform the problem to a different one, one that has the same solution: $A'\vec{x} = \vec{b}'$
- The transformation yields a matrix with all zeros below its main diagonal:

$$A' = \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix}, \quad \vec{b}' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ v'_n \end{bmatrix}$$

A simple backward substitution method can be used to obtain the solution now!

Gaussian Elimination: Obtaining An Upper-Triangle Coefficient Matrix

- Solutions to the system are invariant to three *elementary operations*:
  - Exchange two equations of the system
  - Replace an equation with its nonzero multiple
  - Replace an equation with a sum or difference of this equation and some multiple of another
- Consider the following example:

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned} \qquad \left[\begin{array}{rrr|r} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right] \qquad \begin{array}{l} \text{row } 2 \leftarrow row2 - row1 * \frac{4}{2} \\ \text{row3} \leftarrow row3 - row1 * \frac{1}{2} \end{array}$$

Gaussian Elimination: Obtaining An Upper-Triangle Coefficient Matrix

- Solutions to the system are invariant to three *elementary operations*:
  - Exchange two equations of the system
  - Replace an equation with its nonzero multiple
  - Replace an equation with a sum or difference of this equation and some multiple of another
- Consider the following example:

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned} \qquad \left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{array}\right] \text{row3} \leftarrow row3 - row2 * \frac{1}{2}$$

Gaussian Elimination: Obtaining An Upper-Triangle Coefficient Matrix

- Solutions to the system are invariant to three *elementary operations*:
  - Exchange two equations of the system
  - Replace an equation with its nonzero multiple
  - Replace an equation with a sum or difference of this equation and some multiple of another
- Consider the following example:

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned} \qquad \left[\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right]$$

Upper-triangle form!

LU Decomposition

- If we track the row multiples used during Gaussian elimination, we can construct a lower-triagonal matrix (with one's on the diagonal)
$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$
- We can also consider the upper-triangular matrix produced by Gaussian elimination, leaving off the $\vec{b}'$ vector:
$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$
- It turns out that $A = LU$, so we can re-write our original system as $LU\vec{x} = \vec{b}$
- We can split this into two steps, and solve each with back substitution: $L\vec{y} = \vec{b}$ then $U\vec{x} = \vec{y}$
- Advantage: We can solve many systems with different $\vec{b}$ vectors in the same way, with minimal additional effort

LU Decomposition

- If we track the row multiples used during Gaussian elimination, we can construct a lower-triagonal matrix (with one's on the diagonal)
$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$
- We can also consider the upper-triangular matrix produced by Gaussian elimination, leaving off the $\vec{b}'$ vector:
$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$

NOTE: We can actually store L and U in the *same matrix* to save space.

- It turns out that $A = LU$, so we can re-write our original system as $LU\vec{x} = \vec{b}$
- We can split this into two steps, and solve each with back substitution: $L\vec{y} = \vec{b}$ then $U\vec{x} = \vec{y}$
- Advantage: We can solve many systems with different $\vec{b}$ vectors in the same way, with minimal additional effort

Matrix Inversion

- The inverse of a matrix, denoted A', is defined as $AA' = I$, where I is the identity matrix

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & 0 \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

- But this can be written as a series of systems of linear equations, $A\vec{x}^j = \vec{e}^j$ where:

$$A' = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \ddots & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix}$$

  - $\vec{x}^j$ is the j^{th} column of the inverse matrix
  - $\vec{e}^j$ is the j^{th} column of the identity matrix
- We can compute the LU decomposition of A, then systematically attempt to solve for each column of the inverse
- If compute a U with zeros on the diagonal, there is no inverse and A is said to be *singular*

Matrix Inversion

- The inverse of a matrix, denoted A', is defined as $AA' = I$, where I is the identity matrix
- But this can be written as a series of systems of linear equations, $A\vec{x}^j = \vec{e}^j$ where:
  - $\vec{x}^j$ is the j^{th} column of the inverse matrix
  - $\vec{e}^j$ is the j^{th} column of the identity matrix

$$I = \begin{array}{c} \vec{e}^1 \\ \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & 0 \\ 0 & 0 & \dots & 1 \end{bmatrix} \end{array}$$

$$A' = \begin{array}{c} \vec{x}^1 \\ \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \ddots & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix} \end{array}$$

- We can compute the LU decomposition of A, then systematically attempt to solve for each column of the inverse
- If compute a U with zeros on the diagonal, there is no inverse and A is said to be *singular*

Matrix Inversion

- The inverse of a matrix, denoted A', is defined as $AA' = I$, where I is the identity matrix

$$I = \begin{array}{cc} & \vec{e}^2 \\ & \left[\begin{array}{cccc} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{array}\right] \end{array}$$

- But this can be written as a series of systems of linear equations, $A\vec{x}^j = \vec{e}^j$ where:
  - $\vec{x}^j$ is the j^{th} column of the inverse matrix
  - $\vec{e}^j$ is the j^{th} column of the identity matrix

$$A' = \begin{array}{c} \vec{x}^2 \\ \left[\begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \ddots & & \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{array}\right] \end{array}$$

- We can compute the LU decomposition of A, then systematically attempt to solve for each column of the inverse
- If compute a U with zeros on the diagonal, there is no inverse and A is said to be *singular*

Book Topics Skipped in Lecture

- In section 5.5:
  - *Josephus Problem* (pp. 182–184)
- In section 5.6:
  - *Search and Insertion in a Binary Search Tree* (pp. 188–189)
- In section 6.2:
  - The GAUSSELIMNATION and BETTERGAUSSELIMINATION algorithms in detail (pp. 202–203)
  - *Computing a Determinant* (pp. 206–207)

Assignments

- This week's assignments:
  - Section 5.3: Problems 1, 2, & 5
  - Section 5.4: Problems 1, 2, & 5
  - Section 5.5: Problems 2 & 4
  - Section 5.6: Problems 2 & 6
  - Section 6.1: Problems 1, 5, & 6
  - Section 6.2: Problems 1, 2, & 7

Project II: Balanced Trees

See project description at:
`http://www.cs.gmu.edu/~pwiegand/cs483/assignments.htm`

The project will be **due by midnight April 7**.