Questions of the day

- What's the best way to store a graph in memory?
- How do we analyze algorithms using graphs?

Graph representation and analysis

William Hendrix

Lecture 10

Outline

- Review
  - Iterative dynamic programming
  - Dynamic programming practice
  - Greedy algorithms
- Graph theory
- Graph representations
- Graph analysis

Dynamic programming review

- Design strategy for recursive problems with repeated subproblems

1. Solve problem recursively
   - Recurrence and base cases
2. Determine data structure
   - Usually based on number of changing parameters
   - E.g., LCS: start index of each string => 2D array

<u>Memoization</u>

3. Determine sentinel value
   - Not a valid solution
4. Wrapper function, memo check, store before returning

<u>Iterative dynamic programming</u>

3. Determine iteration order
   - Start at base case, move in opposition to recursion
4. Decide if space complexity can be reduced
5. Allocate data structure, write loops, recursion and return => reads and writes, problem variables => loop variables, return answer

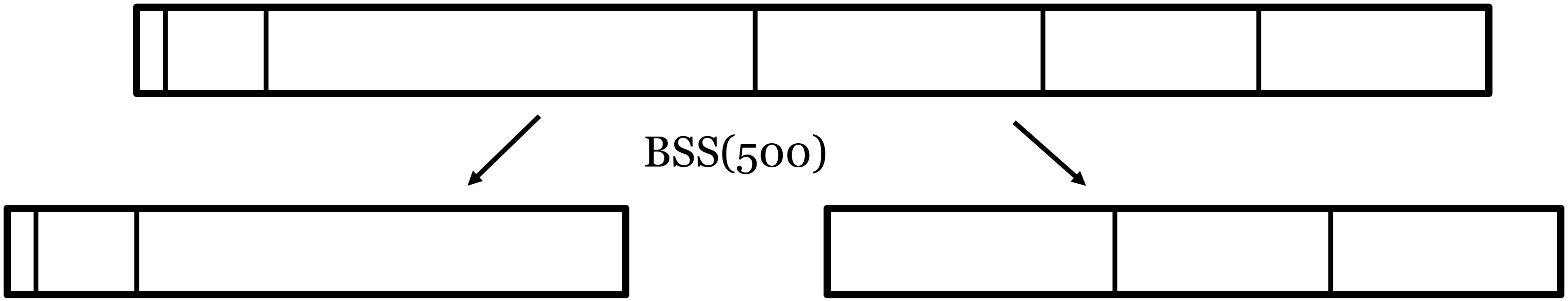
Dynamic programming example

- Binary string splitting
  - Given a string *a* and index *i*, return *a[1..i]* and *a[i+1..n]*
  - Trivial, linear time
- Multiway string splitting
  - Use BSS to split string in multiple places
  - **Output:** order of cuts that minimizes cost
  - **Example:** $n=1000$, $i = (10, 100, 500, 700, 850)$

- Left-to-right:
  - 10: cost = 1000
  - 100: cost = 990
  - 500: cost = 900
  - 700: cost = 500
  - 850: cost = 300
  - Total cost = 3690
- Binary splitting:
  - 500: cost = 1000
  - 100: cost = 500
  - 700: cost = 500
  - 850: cost = 300
  - 10: cost = 100
  - Total cost = 2400

Identifying optimal substructure

- Make a decision



- Can we use recursion to solve the rest?
  - Yes!
- How do we combine recursive solutions to solve overall problem?
  - For each decision, cost of cut + best cost(LHS) + best cost(RHS)
  - Best cost = min cost of all decisions
  - Base case
    - No cuts left to make
    - Cost = 0
- Do subproblems overlap?
  - Yes: cut at 500, then 700 yields same “pieces” as reverse order

String splitting

- Write recurrence
  - Best cost = min{length + best cost(LHS) + best cost(RHS)}
  - $C(a) = n + \min_x \{C(a[1..x]) + C(a[x+1..n])\}$
  - $C(a[x..y]) = y - x + \min_i \{C(a[x..i]) + C(a[i..y])\}$
- Identify parameters of recursive function
  - String, cuts, *x*, *y*
- Data structure: $\text{cuts} \times \text{cuts}$
- Base case: no cuts between *x* and *y* (consecutive)

	10	100	500	700	850	1000
1						
10				?		
100						
500						
700						
850						

Final solution

Dynamic programming exercise

- Answer the following questions about a DP algorithm to calculate the cost of string splitting based on the following recurrence:

$$C(i,j) = \begin{cases} 0, & \text{if } j = i+1 \\ \ell[j] - \ell[i] + \min\limits_{i<x<j} \{C(i,x) + C(x,j])\}, & \text{otherwise} \end{cases}$$

where ℓ is the list of cut indexes and $\min\limits_{i<x<j} \{C(i,x) + C(x,j)\}$ is the min value of $C(i,x) + C(x,j)$ for all values of x between i and j (exclusive)

1. What is a reasonable sentinel value for a memoized algorithm?
2. Give pseudocode for a memoized algorithm.

1. -1
2.
Helper(n, l):
Append 0 and n to the beginning and end of l
numcuts = |l|
dp = Array(numcuts, numcuts)
Initialize dp to -1

Dynamic programming exercise

- Answer the following questions about a DP algorithm to calculate the cost of string splitting based on the following recurrence:

$$C(i,j) = \begin{cases} 0, & \text{if } j = i+1 \\ \ell[j] - \ell[i] + \min\limits_{i<x<j} \{C(i,x) + C(x,j])\}, & \text{otherwise} \end{cases}$$

where ℓ is the list of cut indexes and $\min\limits_{i<x<j} \{C(i,x) + C(x,j)\}$ is the min value of $C(i,x) + C(x,j)$ for all values of x between i and j (exclusive)

1. -1 (cannot be nonnegative)

2.

Input: a: string of length n
Input: ℓ: locations at which to split a
Input: k: length of ℓ
Output: Minimum cost of splitting a

Algorithm: MinSplit
$cost = \text{Array}(k+2, k+2)$
Initialize $cost$ to -1
Prepend 1 to ℓ and append n
return MemoSplit$(1,\ k+2)$

Algorithm: MemoSplit(x, y)
if $cost[x,y] \neq -1$ **then**
  return $cost[x,y]$
else if $y = x+1$ **then**
  $cost[x,y] = 0$
else
  $mincost = \infty$
  for $z = x+1$ to $y-1$ **do**
    $temp = \ell[y] - \ell[x] + \text{MemoSplit}(x,z) + \text{MemoSplit}(z,y)$
    $mincost = \min\{mincost, temp\}$
  end
  $cost[x,y] = mincost$
end
return $cost[x,y]$

Dynamic programming exercise

- Answer the following questions about a DP algorithm to calculate the cost of string splitting based on the following recurrence:

$$C(i,j) = \begin{cases} 0, & \text{if } j = i+1 \\ \ell[j] - \ell[i] + \min\limits_{i<x<j} \{C(i,x) + C(x,j])\}, & \text{otherwise} \end{cases}$$

where ℓ is the list of cut indexes and $\min\limits_{i<x<j} \{C(i,x) + C(x,j)\}$ is the min value of $C(i,x) + C(x,j)$ for all values of x between i and j (exclusive)

3. What are the recursive calls made by C(1, 6) when *n* = 6?
4. What is a valid iteration order for an iterative DP algorithm?

	1	100	500	700	850	1000
1						
100						
500						
700						
850						
1000						

5. Can space be reduced?

Dynamic programming exercise

- Answer the following questions about a DP algorithm to calculate the cost of string splitting based on the following recurrence:

$$C(i,j) = \begin{cases} 0, & \text{if } j = i+1 \\ \ell[j] - \ell[i] + \min\limits_{i<x<j} \{C(i,x) + C(x,j])\}, & \text{otherwise} \end{cases}$$

where ℓ is the list of cut indexes and $\min\limits_{i<x<j} \{C(i,x) + C(x,j)\}$ is the min value of $C(i,x) + C(x,j)$ for all values of x between i and j (exclusive)

3. Everything to the left and everything below
4. Bottom-to-top, left-to-right or left-to-right, bottom-to-top

	1	100	500	700	850	1000
1						*
100						
500						
700						
850						
1000						

5. No!

Iterative pseudocode

```
Input: a: string of length n
Input: ℓ: locations in a to cut
Input: k: length of ℓ
Output: Minimum cost to split a at ℓ
Algorithm: IterSplit
Add 1 and n to the beginning and end of ℓ
cost = Array(k + 2, k + 2)
for x = 1 to k + 1 do
    for y = x − 1 down to 1 do
        if y = x − 1 then
            cost[x, y] = 0
        else
            mincost = ∞
            for z = x + 1 to y − 1 do
                temp = ℓ[y] − ℓ[x] + cost[x, z] + cost[z, y]
                mincost = min{mincost, temp}
            end
            cost[x, y] = mincost
        end
    end
end
return cost[1, k + 2]
```

Could also make separate loop for base cases

Recursive case

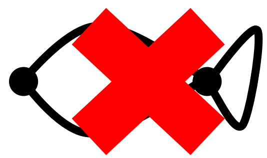
Complexity: $\Theta(k^3)$
Space: $\Theta(k^2)$
Not possible to reduce space complexity

Greedy algorithms review

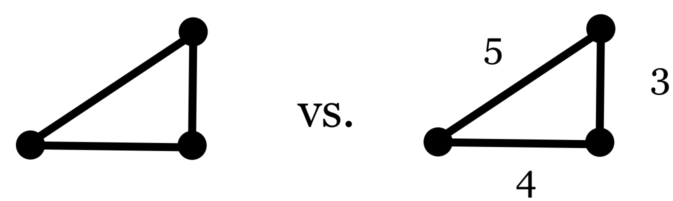
- **Strategy:**
  - Break down problem into sequence of decisions
  - Make "best" choice for each decision
- **Example:** find largest subset with sum below a threshold
  - Choose min element
    - *Greedy idea:* min element gives us more "room" to find other elements
  - Repeat until next element would exceed threshold
- Proof of correctness is tricky
  - *Intuition:* greedy is only correct when greedy choice always "as least as good" as any alternative
  - Many greedy algorithms are incorrect
- Natural choice for optimization problems
- Very efficient (heaps!)
- Can be used as a heuristic
- Often not correct

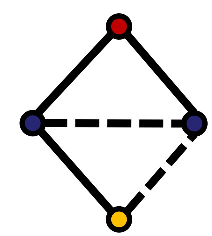
Graph theory review

- Mathematical abstraction for network of relationships
- **Vertices (nodes):** set of objects
  - Denoted as V
- **Edges:** set of relationships between vertices
  - Connect *two* vertices together
  - Denoted as E



- Graph variants
  - **Simple** graphs vs. multigraphs
    - No self loops
    - No edges between same pair of vertices
  - Directed vs. **undirected**
    - Symmetric or asymmetric relationships
  - Weighted vs. **unweighted**
    - Edges have "length" or "strength"
  - Labelled vs. **unlabelled**
    - Vertices and/or edges may have categories





Graph theory review

- **Order:** number of vertices in the graph
  - Usually denoted as n (or $|V|$)
- **Size:** number of edges in the graph
  - Usually represented as m (or $|E|$)
  - Graph complexity can depend on both n and m
    - $m = O(n^2)$
- **Adjacent:** two vertices with an edge between them
- **Incident:** vertex and an edge where edge connects to vertex
- **Degree:** number of vertices adjacent to a given vertex
  - Denoted $\deg(v)$ or $\deg_G(v)$
  - Max degree: denoted as Δ or $\Delta(G)$
  - Min degree: denoted as δ or $\delta(G)$
- **Neighborhood:** set of vertices adjacent to a given vertex
  - Denoted $N(v)$ or $N_G(v)$

Handshaking Lemma

Theorem 1. *For any simple graph $G = (V, E)$,*

$$\sum_{v \in V} \deg(v) = 2|E|,$$

where $\deg(v)$ *is the degree of vertex v in graph G.*

Proof (informal). When $m = 0$, every vertex has degree 0, so
degree sum $= 0 = 2m$.

Suppose true for graphs with k edges, and
let G have $k + 1$ edges.

Remove an edge (u, v)
$\Rightarrow$ new graph has k edges
$\Rightarrow$ new graph has degree sum $= 2k$.

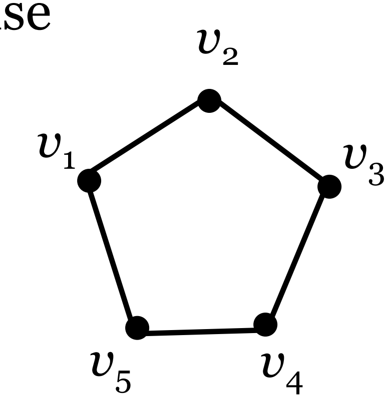
Degree of u and v one more in G, otherwise the same

G has degree sum $2k + 2 = 2m$.

Graph representation

- Three main representations of a graph in memory
  - C++: look up Boost Graph Library (www.boost.org)
- **Adjacency matrix**
  - n by n matrix
  - a_{ij} is 1 if v_i and v_j are adjacent, 0 otherwise

v_1	0	1	0	0	1
v_2	1	0	1	0	0
v_3	0	1	0	1	0
v_4	0	0	1	0	1
v_5	1	0	0	1	0



  - Symmetric for undirected graphs
    - *Optimization:* only store lower triangle
  - Uses numbers > 1 for multiple edges
  - Or: a_{ij} stores edge weight
    - Use sentinel value like 0 or Inf for missing edges
  - Edge/vertex labels stored separately

Common graph operations

- **Graph(n)**
  - Initializes a graph with n vertices and 0 edges
- **AddEdge(u, v)**
  - Adds an edge from u to v
- **RemoveEdge(u, v)**
  - Removes the edge (u, v) from the graph
- **IsAdjacent(u, v)**
  - Returns whether (u, v) is an edge of the graph
- **GetNeighbors(v)**
  - Returns set of neighbors of v

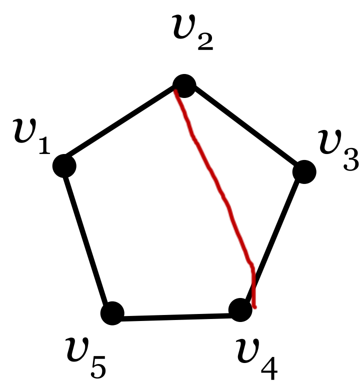
Adjacency matrix operations

- **Graph(*n*)**
  - Initialize n by n matrix
  - $\Theta(n^2)$ time
- **AddEdge(*u*, *v*)**
  - $a_{uv} = 1$
  - $\Theta(1)$ time
- **RemoveEdge(*u*, *v*)**
  - $a_{uv} = 0$
  - $\Theta(1)$ time
- **IsAdjacent(*u*, *v*)**
  - Return $a_{uv} = 1$
  - $\Theta(1)$ time
- **GetNeighbors(*v*)**
  - Scan row v of matrix and add u to list if $a_{vu} = 1$
  - $\Theta(n)$ time

Graph representations, cont'd

- **Adjacency list**
  - Stores list (or set) of neighbors for each vertex

v_1:	2	5	
v_2:	1	3	4
v_3:	2	4	
v_4: 2	3	5	
v_5:	1	4	



  - Neighbors are often sorted
    - Sometimes use hash table for large graphs
  - Can store edge weights or labels in `struct`/object
  - Can also store edges using a linked structure
    - Improves performance of adding/deleting vertices

Hash-based adjacency list

- **Main idea**
  - Store neighbors in hash table
- Graph: make empty hash tables
- Add neighbors: insert
- Remove: delete
- IsAdjacent: search
- GetNeighbors: iterate

Operation	Adjacency list
Graph(n)	
AddEdge(u, v)	
RemoveEdge(u, v)	
IsAdjacent(u, v)	
GetNeighbors(v)	

* Expected case
† Amortized

Hash-based adjacency list

- **Main idea**
  - Store neighbors in hash table
- Graph: make empty hash tables
- Add neighbors: insert
- Remove: delete
- IsAdjacent: search
- GetNeighbors: iterate

- Reduces expected complexity
- GetNeighbors still linear
- More complex
- Higher coefficients
- Poor worst-case performance

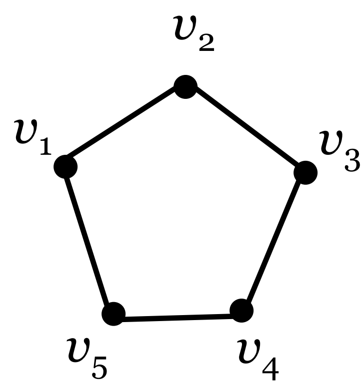
Operation	Adjacency list
Graph(n)	$\Theta(n)$
AddEdge(u, v)	$\Theta(1)$*†
RemoveEdge(u, v)	$\Theta(1)$*
IsAdjacent(u, v)	$\Theta(1)$*
GetNeighbors(v)	$\Theta(\deg(v))$

* Expected case
† Amortized

Graph representations, cont'd

- **Edge list**
  - List of all edges in graph
    - Usually sorted
    - Undirected: may or may not include "reverse" edges

1	2
1	5
2	3
3	4
4	5



  - Edge weights or labels appear after vertex IDs
  - Order and vertex labels represented separately

Graph representation analysis

Operation	Adjacency matrix	Adjacency list	Edge list
Graph(n)	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
AddEdge(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(m)$
RemoveEdge(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(m)$
IsAdjacent(u, v)	$\Theta(1)$	$\Theta(1)$*	$\Theta(\lg(m))$
GetNeighbors(v)	$\Theta(n)$	$\Theta(\deg(v))$	$\Theta(\mathrm{m})$
Convert from:	**Adjacency matrix**	**Adjacency list**	**Edge list**
Adj. matrix	n/a	$\Theta(n^2)$	$\Theta(n^2)$
Adj. list	$\Theta(n^2)$	n/a	$\Theta(m)$
Edge list	$\Theta(n^2)$	$\Theta(m)$	n/a
Space:	$\Theta(n^2)$	$\Theta(m+n)$	$\Theta(m)$

- Matrix: good for small graphs or dense graphs
  - *Dense*: significant fraction of edges exist, $m = \Theta(n^2)$
- Adj. list: good for sparse graphs
  - Most "real world" graphs are sparse
- Edge list: commonly used to store on disk

Facebook friend graph statistics

- As of May 2011:
  - n: ~721 million
  - m: ~68.7 billion
  - Δ: 5000 (hard cap)
  - Avg degree: ~191
  - Median degree: 99
  - 99.91% are connected together
  - 99.6% are within 6 “hops” of one another
  - 92% are within 5 “hops”
- *Source*: “The Anatomy of the Facebook Social Graph,” J. Ugander et al., arXiv.org, Nov 2011.

Very sparse!

Facebook graph representation

Average case timing (approx.):

Best choice! ↓ (Adjacency list)

Operation	Adjacency matrix	Adjacency list	Edge list
Graph(n)	~ 2 years	721 ms	1 ns
AddEdge(u, v)	1 ns	2 ns	68.7 s
RemoveEdge(u, v)	1 ns	2 ns	68.7 s
IsAdjacent(u, v)	1 ns	2 ns	36 ns
GetNeighbors(v)	721 ms	191 ns	68.7 s
Convert from:	**Adjacency matrix**	**Adjacency list**	**Edge list**
Adj. matrix	n/a	~ 2 years	~ 2 years
Adj. list	~ 2 years	n/a	68.7 s
Edge list	~ 2 years	68.7 s	n/a
Space:	~ 58 PB (million GB)	~517 GB	~512 GB

Graph algorithm analysis

AL: $\Theta(\deg v)$

- Analysis similar to general algorithm analysis
- Complexity given in terms of graph features
  - n: order (vertices)
  - m: size (edges)
  - $\deg(v)$: all neighbors of a given vertex
  - Δ: max degree
- Also depends on graph representation
- Sometimes useful to add up function calls separately
- **Example**

$\sum \deg v = \Theta(m)$

Dominance relationships

$$\deg(v) = O(\Delta),\ O(n),\ O(m)$$
$$\Delta = O(n),\ O(m)$$
$$m = O(n^2)$$

No direct relationship b/w n and m

AM: $\Theta(n)$

$\cdot\, O(n)$

$\cdot\, \Theta(n^2)$

$\sum_{v \in V} \deg v = O(m)$

Input: $G = (V, E)$: a graph
Input: n: the number of vertices in G
Input: m: the number of edges in G
Output: the average degree of the vertices in G
Algorithm: AvgDegree
$sum = 0$
for v in V **do**
  for u in $N(v)$ **do**
    $sum = sum + 1$
  end
end
return sum/n

n iters

deg v iters

iters @ $\Theta(1)$

order $|V|$

Graph algorithm analysis

- Analysis similar to general algorithm analysis
- Complexity given in terms of graph features
  - n: order (vertices)
  - m: size (edges)
  - deg(v): all neighbors of a given vertex
  - Δ: max degree
- Also depends on graph representation
- Sometimes useful to add up function calls separately
- **Example**

Dominance relationships

$$\deg(v) = O(\Delta),\ O(n),\ O(m)$$
$$\Delta = O(n),\ O(m)$$
$$m = O(n^2)$$

No direct relationship b/w n and m

Input: $G = (V, E)$: a graph
Input: n: the number of vertices in G
Input: m: the number of edges in G
Output: the average degree of the vertices in G
Algorithm: AvgDegree
$sum = 0$ — $\Theta(1)$
for v in V **do** — n iterations
  for u in $N(v)$ **do** — $\Theta(\deg(v))$ iterations
    $sum = sum + 1$ — $\Theta(1)$
  end
end
return sum/n — $\Theta(1)$

Cost to calculate $N(v)$:
$\Theta(\deg(v))$ for list
$\Theta(n)$ for matrix

Total:
$\sum_{v \in V} \Theta(\deg(v)) = \Theta(m)$ for list
$\Theta(n^2)$ for matrix

Graph theory example

- What is the worst-case complexity for the following algorithm to compute the average degree in a graph?

```
Input: G = (V, E): a graph
Input: n: the number of vertices in G
Input: m: the number of edges in G
Output: the average degree of the vertices in G
Algorithm: FastAverage
return 2m/n
```

Graph theory example

- What is the worst-case complexity for the following algorithm to compute the average degree in a graph?

```
Input: G = (V, E): a graph
Input: n: the number of vertices in G
Input: m: the number of edges in G
Output: the average degree of the vertices in G
Algorithm: FastAverage
return 2m/n
```

- **Moral of the story:** knowledge is power

Coming up

- Traversal-based algorithms
- Weighted graph algorithms

- **Recommended readings:** Sections 13.1-13.4
- *Practice problems:* R-13.1, R-13.2, R-13.5, R-13.7, C-13.4