4. Divide and conquer

1. Very fast algorithms
2. Vector problem solutions
3. Tree-based algorithms
4. Graph algorithms

Inquiry

- What is the common feature in the design of the
  - binary search
  - BST, and
  - heap?
- How tall is a tree?
Classroom exercise

• **Given:**
  – Nine (12; 16) small objects that look the same
  – All, being the same weight except one
• Using a *balance scale*, how many weighings does it take (minimum) to find the one that’s different?
• Solve as teams; report back

Topic objective

Explain the divide-and-conquer approach and apply it to a variety of algorithm-design problems, giving the appropriate time analysis for each.
1. Very fast algorithms

• What is a good strategy for finding a word in a paper dictionary?

• How long does a Google search take?
Subtopic objectives

4.1 Relate logarithmic time to very fast algorithms*
A fast numeric algorithm

- Consider finding the logarithm of $n$:
  \[
  \log_b(n) = \begin{cases} 
  0 & \text{if } n = 1 \\
  1 + \log_b(n/b) & \text{otherwise}
  \end{cases}
  \]

- *Iteratively*:
  \[
  y \leftarrow 0 \\
  \text{while } n > 0 \\
  \quad n \leftarrow n / b \\
  \quad y \leftarrow y + 1
  \]

- What is the running time of $\log$?

A way to describe binary search

- if middle element of a sorted array
  matches key, return true

- Otherwise, search the left or right half,
  depending on whether the middle element
  is greater than or less than the key

- How many times is a middle element
  checked, followed by a search of the left or
  half of the remaining array?
Decision trees for search

- The *decision tree* for a search of an array must examine each element.
- Searching an *unsorted* array, only one element is considered per step.
- Hence the tree depth (running time) is $O(n)$.
- Searching a *sorted* array, inspecting the middle element enables a decision to search only the right or left half of the array.
- Hence the decision tree depth is $O(\lg n)$.

Divide and conquer

- Break problem into *base* and *recursive* cases.
- We “conquer” by defining the recursive case as a simpler version of the original problem.
- *Examples:*
  - binary search
  - Quicksort
  - BST search, traversal
  - Mergesort
  - Heapsort
  - convex hull
  - closest pair
  - searches of graphs
  - topological sort
  - order statistic
  - matrix multiplication
**Binary-search** *(A, first, last)*

if `first > last`  
// (i.e., nothing to search)
return false
otherwise

```
middle ← (first + last) ÷ 2
if A[middle] matches key
    return true
otherwise
    if A[middle] > key
        return Bin-search(A, first, middle − 1, key)
    otherwise
        return Bin-search(A, middle + 1, last, key)
```  
> Post: returns true iff `key` is in A [first...last]

**Binary-search recurrences**

\[
\text{Bin-srch}(A, x) = \begin{cases} 
\text{false} & \text{if } |A| = 0 \\
\text{true} & \text{if } A\lfloor|A| ÷ 2\rfloor = x \\
\text{Bin-srch}(A[0..\lfloor|A| ÷ 2\rfloor]) & \text{if } A\lfloor|A| ÷ 2\rfloor > x \\
\text{Bin-srch}(A\lfloor|A| ÷ 2\rfloor+1..|A|) & \text{otherwise}
\end{cases}
\]

- Recursive call examines at most half the remaining elements
- Recurrence Solution:

\[
T_{\text{Bin-srch}}(n) = \Theta(\log_2 n) = \Theta(\lg n)
\]
Divide-and-conquer strategy

- Definition of divide and conquer: an algorithm in which
  - Problem is broken into parts
  - Each part is solved recursively
  - Partial solutions are combined into an overall solution

Solving recurrences: logarithmic time

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
\Theta(1) + T(n/2) & \text{otherwise}
\end{cases}
\]

Solution: \( \Theta(\lg n) \), because at each of \((n/2)\) recursive steps the remaining time is cut in half

Example: Binary search
Russian peasants’ algorithm (aka Al-Kwarizmi’s)

Product \((a, b)\)

\[
\text{result} \leftarrow 0 \\
\text{while } a > 0 \\
\quad \text{if } a \text{ is odd} \\
\quad \quad \text{add } b \text{ to result} \\
\quad a \leftarrow \left\lfloor a \div 2 \right\rfloor \\
\quad b \leftarrow \text{Product}(b, 2) \\
\text{Return } \text{result}
\]

Example: \(13 \times 5 = 65\)

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(\text{product})</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>65</td>
</tr>
<tr>
<td>0</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

Recurrence for Product

\[
\text{Product}(a, b) = \begin{cases} 
\text{false} & \text{if } b = 0 \\
2 \text{ Product}(a, b/2) & \text{if } b \text{ even} \\
a + 2 \text{ Product}(a, \lfloor b/2 \rfloor) & \text{otherwise}
\end{cases}
\]

- Time recurrence:
\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 0 \\
\Theta(1) + T_{\text{Product}}(n/2) & \text{otherwise}
\end{cases}
\]
- Where \(n\) is size of representation of \(a, b\)
Exponential and logarithmic functions

• Many processes of growth and decay are described by exponential and logarithmic functions
• Function \( \log_b(x) \) is the inverse of function \( b^x \)
• These functions grow extremely slowly and extremely quickly, respectively
• These functions grow proportional to the base; i.e., the big-O analysis is independent of base

Logarithms and running time

• In algorithm analysis, we can ignore the base of a logarithm because \( \log_b n \) is proportional to \( \log_c n \) for all constants \( b \) and \( c \)
• Thus, \( \Theta(\lg n) = \Theta(\log_2 n) = \Theta(\log_b n) \) for any base \( b \)
• Logarithmic complexity is desirable because the log function grows slowly
Decrease by constant factor

- Looking for solutions where
  \[ T(n) = T(n/b) + f(n) \]
- If \( T(n) = T(n/2) + f(n) \), then depth of recursion is \( \log_2 n \)
- Examples:
  - Binary search, where \( f(n) = 1 \), hence solution is \( O(\lg n) \)
  - Quicksort, where \( f(n) = n \), hence solution is \( O(n \lg n) \)

Master Theorem ("Main Recurrence Theorem")

- Let \( T(n) = aT(\lfloor n/b \rfloor) + f(n) \)
- Let \( f(n) \in \theta(n^d), d \geq 0 \)
- Then \( T(n) \in \)
  - \( \theta(n^d) \), if \( a < b^d \)
  - \( \theta(n^d \lg n) \), if \( a = b^d \)
  - \( \theta(n \log_b a) \), if \( a > b^d \)
- This theorem simplifies solving some recurrence equations for divide-and-conquer algorithms
- Example: Binary search
Applying the Master Theorem

- Let \( T(n) = aT(n/b) + f(n), f(n) \in \Theta(n^d) \)
- Then by Master Thm., \( T(n) \in \Theta(n^d \lg n), \text{ if } a = b^d \)
- Examples:
  - (Binary search) \( T(n) = T(n/2) + O(1) \), so \( a = 1, b = 2, d = 0 \), hence \( a = b^d, f(n) = 1 \), hence \( T(n) \in \Theta(\lg n) \)
  - (Quicksort) \( T(n) = 2T(n/2) + O(n) \), so \( a = 2, b = 2, d = 1 \), hence \( a = b^d \)
- Master Theorem does not apply to linear search, quadratic sorts, or exponential-time algorithms

2. Vector problem solutions

- What do the binary search, Quicksort, merge, and merge sort algorithms have in common?
4. Divide and conquer

**Subtopic objectives**

4.2a Write a divide-and-conquer algorithm*

4.2b Write a recurrence for a divide-and-conquer algorithm*

4.2c Write and solve a time recurrence for a divide-and-conquer algorithm*

---

**QuickSort(A)**

If |A| > 1

\[(A, pvloc) \leftarrow Partition(A)\]

\[QuickSort(A[1.. pvloc - 1])\]

\[QuickSort(A[pvloc + 1 .. |A|])\]

Return A

- *Partition* rearranges A so that all elements less than A[pvloc] are to its left, and all elements greater are to its right
- Recursive calls sort arrays half as large at each recursive step
Quicksort top-down strategy

Unsorted

Partitioned

Partitioned

Sorted sequences of \leq 3

- **Time:** \( n \log_2 n \) because number of partition steps is \( \log_2 n \)

Partition code

Partitions subarray `num[first..last]`, returns index of pivot value

```c
int partition(int num[], int first, int last)
{
    int pivot = num[first];
    int left = first + 1, right = last;
    do {
        while (left <= right && num[left] <= pivot)
            ++left;
        while (left <= right && num[right] > pivot)
            --right;
        if (left < right)
            swap (num[left], num[right]);
    } while (left < = right);
    swap (num[first], num[right]);
    return right;
}
```
Correctness of Quicksort

- **To prove:** For all $i$ and $j$ less than the size of array $A$, $i < j$ implies $A[i] \leq A[j]$
- **Base step:** Property to be proven holds vacuously for array of size 1, since $i = j$
- **Induction step (summary):** This step must show that if Quicksort leaves each pair of array elements in correct order for all arrays of size $n$, then the same is true for all arrays of size $(n + 1)$.
- Our argument uses the fact that each partition is of size $n$ or smaller, so each gets sorted

Quicksort recurrences

\[
\text{Qsort}(A) = \begin{cases} 
A & \text{if } |A| \leq 1 \\
\text{Qsort}(\text{Right-partition}(A, A[1])) & \text{otherwise}
\end{cases}
\]

\[
\text{Left-partition}(A, x) = \begin{cases} 
\lambda & \text{if } A = \lambda \\
\text{Left-partition}(A[2..|A|], x) + A[1] & \text{otherwise}
\end{cases}
\]

\text{Right-partition mirrors Left-partition}
### The partition step

1. **pivot**
   
   \[
   \begin{array}{cccccccc}
   5 & 1 & 7 & 10 & 2 & 4 & 8 & 11 & 3 & 6 \\
   \end{array}
   \]

   *Repeatedly finds leftmost element in left partition that should be in right and rightmost that should be in left, swaps them*

2. **pivot**
   
   \[
   \begin{array}{cccccccc}
   5 & 1 & 3 & 4 & 2 & 10 & 8 & 11 & 7 & 6 \\
   \end{array}
   \]

3. **pivot**
   
   \[
   \begin{array}{cccccccc}
   2 & 1 & 3 & 4 & 5 & 10 & 8 & 11 & 7 & 6 \\
   \end{array}
   \]

   to partition

Here, 3 swaps with 7, 10 swaps with 4, and pivot 5 swaps into place

---

### Complexity of Quicksort (avg. case)

\[
T_{\text{Quick}}(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
O(n) + 2T_{\text{Quick}}(\lceil (n - 1) / 2 \rceil) & \text{otherwise}
\end{cases}
\]

- There are \((\lg n)\) levels of recursion (average case) because \((n - 1)\) is repeatedly divided by 2
- There are \(n\) steps at each level of recursion to perform partition
- We can toss out term 1 and factor 2
- Solution: \(T(n) = O(n \lg n)\)
- This is *much* better than \(O(n^2)\)
How Quicksort uses divide and conquer

• Sorting problem is divided into
  – partition
  – recursively sort left half
  – recursively sort right half
• Two sorted versions of half the array each are concatenated together into a sorted version of the entire array

Merging two sorted arrays

Repeat until A and B are exhausted:
  Append the lesser of \{A_{ai}, B_{bi}\} to C, incrementing the indexes \(ai, bi,\) and \(ci\) as appropriate

• \(C\) should be as large as \(A, B,\) together
Intuition for merge

To merge two sorted arrays:

- Return the lower of the two first elements, plus a merge between the rest of that array and the entire other array

Merge algorithm

$A$ and $B$ are sorted arrays; $\text{Merge}$ returns a sorted array containing all elements of $A$, $B$

$\text{Merge}(A, B) =$

\[
\begin{cases}
  A & \text{if } B = \lambda \\
  B & \text{if } A = \lambda \\
  B[1] + \text{Merge}(A, B[2..|B|]) & \text{otherwise}
\end{cases}
\]

Spec: If $A$, $B$ are each sorted arrays, then $\text{Merge}(A, B)$ is a sorted array

Problem: Write $T_{\text{Merge}}(n)$ recurrence
**Mergesort**

Recursively divide array in two, sort each half, merge the results

\[
\text{Mergesort}(A) \\
\text{If } |A| \leq 1 \\
\quad \text{return } A \\
\text{else} \\
\quad \text{return Merge(Mergesort(A [1 .. |A| / 2]), Mergesort(A [|A| / 2 +1.. |A|]))}
\]

**Problem:** Write \( T_{\text{Mergesort}}(n) \) recurrence
Mergesort exercise

• Take 7 or 15 students to sort array of 7 or 15 cards
• One student divides the deck in two and gives each half to two others
• Each does the same, each recipient does same
• When two students get 0 or one card each, they merge them and pass back
• Each student who gets back two piles merges them

Order statistics

• Problem: Find $k$th-highest value in an unsorted list; e.g., median is order statistic where $k = \lceil n / 2 \rceil$

• Brute-force solution (equivalent to sorting then finding $A[k]$):

  $Kth$-highest $(A, k) = \begin{cases} 
  A[1] & \text{if } k = 1 \text{ and } \text{size}(A) = 1 \\
  \text{min-elt}(A) & \text{if } \text{size}(A) = k \\
  Kth$-highest $(A - \text{min-elt}(A), k - 1) & \text{otherwise}
  \end{cases}$
Divide-and-conquer order statistic algorithm

- Partition array $A$, finding pivot position $p$
- If $p = k$, return $A[p]$
- If $p < k$, recurse to find $(k-p)$th order statistic of right partition, otherwise recurse for left partition
- Average-case analysis: $T_{Ordstat}(n) = T_{Ostat}(n/2) + T_{Ostat}(n/4) + T_{Ostat}(n/8) \ldots$
- $\ldots = \Theta(n)$

Order statistic recurrence

\[
OS(k, A) =
\begin{cases}
A[pivloc(A)] & \text{if } k = pivloc(A) \\
OS(pivloc(A) - k, A[1..pivloc(A)-1]) & \text{if } k < pivloc(A) \\
OS(k - pivloc(A), A[pivloc(A)+1, |A|]) & \text{otherwise}
\end{cases}
\]

Pivloc is computed by using the Partition from Quicksort
Convex hull problem

- The Convex hull of a set of points is a subset of ordered pairs (on a Cartesian plane) that define a polygon with no concave series of edges ("dents")
- Intuition: convex hull are locations of the posts of the shortest fence that would enclose all the points
- I.e., find smallest convex polygon that contains all of a set of points

Convex hull divide-and-conquer solution

- Find leftmost and rightmost points in set, \( P_1 \) and \( P_2 \) (these are in the convex hull)
- Define upper and lower areas as separated by the line segment \( (P_1, P_2) \)
- In each area, find farthest point from segment, recursively find triangles defined by farthest points -- EXPLAIN
- Combine upper and lower hulls
- Average case: \( T(n) = \theta(n \ lg \ n) \), worst: \( \theta(n^2) \)
### Closest pair (of a set of points)

- **Given a set of ordered pairs (points on a Cartesian plane) find the two points that are closest together.**
- **Algorithm:**
  1. Divide set into two vertical areas
  2. Find closest pair in each
  3. Choose closest of these two
  4. See if any pairs crossing boundary are closer

- \( T(n) = 2T(n/2) + M(n) \), where \( M(n) \) is time to merge the solutions to subproblems
- \( T(n) \in \Theta(n \lg n) \)
- Brute force approach yields \( \Theta(n^2) \)

### Counting inversions for string distances

- Count inversions in left, right halves
- Sort left and right halves
- Recurse for each half of a half
- Similar to *Merge-Sort* but also counts inversions along the way
- **Time:** \( O(n \lg n) \)
3. Tree-based algorithms

- What is it about trees that makes them useful for performance in binary search trees and heaps?
- Can you describe the inorder, preorder, and postorder tree traversals?
Some algorithms with trees

- *Tree construction* (from array)
- *Tree traversals*: Recursively visit subtrees until subtrees are leaves
- *Binary search tree search*: Makes use of BST ordering property and average height of binary tree
- *Heap algorithms*: Make use of height of complete binary tree

Building a tree

- *Example*: expression tree
- *Input*: arithmetic expression
- *Output*: tree, with literals and leaves, operations as internal nodes
Inorder traversal

Inorder traversal (node, op)
1. If node has left child
   Inorder-traverse (Left(node), op)
2. Apply operation op to node
3. If node has right child
   Inorder-traverse (Right(node), op)

• Order for tree above: 1, 3, 5, 6, 7
• Application: Display BST in ascending order

Preorder traversal

A node is operated on as soon as it is first visited.

Preorder-traverse (node, op)
1. Apply operation op to node
2. If node has left child
   Preorder-traverse (Left(node), op)
3. If node has right child
   Preorder-traverse (Right(node), op)

• Order for tree above: 3, 1, 6, 5, 7
• Application: Copy tree
**Postorder traversal**

A node is operated on after its descendants

Postorder-traverse \((node, op)\)

1. If \(node\) has left child
   
   \(Postorder\)-traverse \((Left(node), op)\)

2. If \(node\) has right child
   
   \(Postorder\)-traverse \((Right(node), op)\)

3. Apply operation \(op\) to \(node\)

- Order for tree above: 1, 5, 7, 6, 3
- Application: Deallocate tree

**BST search of a subtree**

\(BST\)-search \((root, key)\)

- If \(root\) is null, return \(false\)
- If \(root\)’s data matches \(key\)
  
  return \(true\)
- Otherwise
  
  if \(root\)’s data > \(key\)
    
    return \(BST\)-search \((left (root), key)\))
  
  otherwise
    
    return \(BST\)-search \((right (root), key)\))

- Initial value of \(root\) is root of entire tree
### Complexity of BST-search

\[ T_{Srch}(n) = \begin{cases} \Theta(1) & \text{if } n < 2 \\ \Theta(1) + T_{Srch}(n/2) & \text{otherwise} \end{cases} = \Theta(\log n) \]

- What are the assumptions of the above?
- Does it apply to best, worst, average cases?
- Compare with binary search
- How does BST-insert compare?

### A heap is a kind of complete binary tree

- Solves priority queue problem efficiently
- A complete binary tree has all levels full except possibly the bottom one, which is filled from the left
- Which are complete binary trees?
Array implementation of complete binary tree

- Root is $A[1]$
- Parent of $A[i]$ is $A[\lfloor i / 2 \rfloor]$ $A[(i-1)/2]$
- If $n$ is size, height is $\lceil \log_2 n \rceil$

Java

```
A[0]  // Root
A[(i-1)/2]  // Parent
```

Height of a complete binary tree with $n$ nodes is $O(\log_2 n)$

- …because size $n$ approximately doubles with each level added
- Or, $2^{height-1} \leq n \leq 2^{height} - 1$
### A maximum heap

- Heap property: for all $i$ up to array size, $A[i] \leq A[i/2]$

```
[10 8 9 4 7 5 2 2 3 1]
```

- The **Heapify** operation
  - **Heapify**($A, i$) applies to one node, with subscript $i$, of a complete binary tree stored in array $A$
  - It assumes that subtrees with roots **Left-child**($i$) and **Right-child**($i$) already have the heap property
  - Node $A[i]$ may meet or violate heap property
  - **Heapify**($A, i$) causes the subtree with root subscript $i$ to have the heap property
Intuition for **Heapify**

- The algorithm drags the value at a selected node down to its proper level where heap property applies to it.
- It does this by recursively exchanging the value in the root node with the higher-priority child of its two children.
- **Complexity**: Depth of complete binary tree with \( n \) nodes, i.e., \( \Theta(\lg n) \)

**Heapify\((A,i)\)** for minimum heap

\[
\begin{align*}
L & \leftarrow \text{Left}(i) \\
R & \leftarrow \text{Right}(i) \\
\text{If } L = \text{null} \text{ and } R = \text{null} \text{ then return} \\
\text{if } L \leq |A| \text{ and } A[L] < A[i] \\
& \quad \text{smallest} \leftarrow L \\
\text{else} \\
& \quad \text{smallest} \leftarrow i \\
\text{if } R \leq |A| \text{ and } A[R] < A[\text{smallest}] \\
& \quad \text{smallest} \leftarrow R \\
\text{if } \text{smallest} \neq i \\
& \quad \text{exchange } A[i] \text{ with } A[\text{smallest}] \\
\text{Heapify}(A, \text{smallest})
\end{align*}
\]

**Precondition:** Left\([i]\), Right\([i]\) are roots of heaps
To convert an array to a heap

Overview: Heapify at every non-leaf node, starting at the bottom

Build-heap(A)
for i ← ⌊|A| / 2⌋ down to 1
Heapify(A, i)

Use version for min or max heap

Because leaves don’t need to be heapified

[SHOW WHY]

Analysis: Θ(n lg n)

Extract value from heap

Extract-min(A)
If |A| < 1
    throw exception
min ← A[1]
|A| ← |A| – 1
Heapify(A, 1)
Return min

• Returns minimum value from a min-heap
• Deletes that value from heap
• Analysis: Θ(lg n)
4. Divide and conquer

Insertion into a min-heap

**Heap-insert** \((A, key)\)

\[
\begin{align*}
\text{Heap-size}(A) & \leftarrow \text{Heap-size}(A) + 1 \\
i & \leftarrow \text{Heap-size}(A) \\
\text{while } i > 1 \text{ and } A[\text{Parent}(i)] > key & \\
& \quad A[i] \leftarrow A[\text{Parent}(i)] \\
& \quad i \leftarrow \text{Parent}(i) \\
A[i] & \leftarrow key
\end{align*}
\]

*Analysis: \(\Theta(\lg n)\)*

because Parent[i] halves \(i\)

To sort an array using a max-heap

**Heapsort** \((A)\)

**Build-heap** \((A)\)

for \(i \leftarrow \text{length}(A)\) downto 2

exchange \(A[1]\) with \(A[i]\)

\[
\begin{align*}
\text{Heap-size}(A) & \leftarrow \text{Heap-size}(A) - 1 \\
\text{Heapify}(A,1)
\end{align*}
\]

*Analysis: \(\Theta(n \lg n)\)*

- **In plain language...**

  Repeatedly extract the highest value from heap and put it at front of a growing sorted array just after the heap
Terminology about heaps

<table>
<thead>
<tr>
<th>Johnsonbaugh-Schaefer</th>
<th>Cormen-Leiserson-Rivest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sift-down</td>
<td>Heapify</td>
</tr>
<tr>
<td>Heapify</td>
<td>Build-heap</td>
</tr>
<tr>
<td>Largest-delete</td>
<td>Extract-max</td>
</tr>
</tbody>
</table>

• Our reference is Cormen-Leiserson-Rivest
• Lesson: In computer science, law of frontier operates for terminology

Properties of binary trees

• Theorem: If $T$ is a full binary tree with $k$ nonleaves, then $T$ has $(2k + 1)$ vertices and $(k + 1)$ leaves

• Theorem: If $T$ is a binary tree of height $h$ and with $k$ leaves, then $k \leq 2^h$, i.e., $h \geq \log_2 k$

• Proofs: inductive
4. Divide and conquer

4. Graph algorithms

- How would you get to your destination on a trip, if you had a map but not GPS?

Subtopic objectives

4.4a Explain breadth-first or depth-first search
4.4b Describe directed acyclic graphs
Three problems

- **Depth-first search**: Label each vertex of graph with a number in order visited, using the strategy of exhausting one path before backtracking to start at a new edge.

- **Breadth-first search**: Label each vertex, using the strategy of visiting each vertex adjacent to the current one at each stage.

- **Topological sort of a dag** (2 versions)

---

**Depth-first search of graph**

\[ \text{GDFS(G)} \]

\[ G = (V, E) \]

> **Pre**: there is a vertex \( v \) in \( V \) s.t. \( \text{Mark}[v] = 0 \)

\[ \text{count} \leftarrow 0 \]

for each vertex \( v \in V \) do

  if \( \text{Mark}(v) = 0 \)

    \[ \text{vdfs}(v, \text{count}) \]

return \( \text{Mark} \)

> **Post**: Vertices are marked in order of some DFS traversal

**Running time**: \( \Theta(n) \)
Depth-first search from vertex

\[ \text{vdfs}(v, V) \]  
\[ \text{count} \leftarrow \text{count} + 1 \]  
\[ \text{Mark}[v] \leftarrow \text{count} \]  
For each vertex \( w \in V \) adjacent to \( v \) do  
\[ \text{if Mark}[w] = 0 \]  
\[ \text{vdfs}(w, V) \]  
> Post: \( v \)'s mark is set

**Challenge:** show termination  
[See URL found by B. Grozier]

Breadth-first search

- Checks all vertices 1 edge from origin, then 2 edges, then 3, etc., to find path to destination
- If \( G = \langle \{a..i\}, (a,b), (a,c), (a,d), (c,d), (b,e), (b,f), (c,g), (g,h), (g,i), (h,i) \rangle \), and
  - BFS input is \( G, a, f \), then
  - Order of search is (\( a,b \), (\( a,c \), (\( a,d \), (\( b,e \), (\( b,f \) )

[pic]
Breadth-first search

**BFS(G)** \[ G = (V, E) \]

Iterative

\[
\begin{align*}
\text{count} & \leftarrow 0 \\
\text{for each vertex } v \in V & \text{ do} \\
\quad & \text{if Mark}(v) = 0 \\
\quad & \quad \text{bfs}(v)
\end{align*}
\]

\[
\begin{align*}
\text{bfs}(v) & \\
\text{count} & \leftarrow \text{count} + 1 \\
\text{Mark}(v) & \leftarrow \text{count} \\
\text{Queue} & \leftarrow (v)
\end{align*}
\]

While not empty(\text{queue}) do

\[
\begin{align*}
\text{For each vertex } w \in V & \text{ adjacent to front}(\text{queue}) \text{ do} \\
\quad & \text{if Mark}(w) = 0 \\
\quad & \quad \text{count} \leftarrow \text{count} + 1 \\
\quad & \quad \text{append } w \text{ to } \text{queue} \\
\text{Remove front of } & \text{queue}
\end{align*}
\]

• Proceeds concentrically
• Uses queue rather than DFS’s stack
• See example (pic)

Counting connected components of a graph

• To see how many disjoint subgraphs \( G = (V, E) \) has:

\[
\begin{align*}
G' & \leftarrow G \\
\text{While } & G'.V \neq \emptyset \\
\text{Do a search on } & G, \text{ deleting vertices from } G'.V \text{ on the way} \\
\text{count} & \leftarrow \text{count} + 1
\end{align*}
\]
Bipartite graphs

• Definition: A graph $G = (V, E)$ is bipartite iff its vertices may be partitioned into independent sets $V_1, V_2 \subseteq V$ s.t. $(\forall (x,y) \in E) x \in V_1$ and $y \in V_2$

• One algorithm to determine if a graph is bipartite: see if the number of connected components is $|V|/2$

Topological sort of dag

• dag: directed acyclic graph

• Example: a listing of courses in a prerequisite relation so that no course is taken before its prerequisites

• Algorithm 1: Perform DFS, arrange vertices in reverse order of becoming dead-ends

• Algorithm 2: Find source (vertex lacking in-edges), delete it and its adjacent vertices recursively. Order of deletion is a topological ordering
References


Notation ideas by A. Bilodeau, 2006.