

## Week 5: Brute Force Algorithms and Asymptotic Analysis

Class presentation (= reduced version of full class notes)

## Brute Force Algorithms

### What is a brute force algorithm?

A straight forward approach to solving a problem

usually directly based on the problem statement and definitions of the concepts involved.

Intuitive

Direct

Relies on sheer computing power

Could involve checking all possible solutions

#### Advantages:

- ✓ Guaranteed to find a correct solution
- ✓ Good for smaller problems
- ✓ Simple strategy
- ✓ Provides a baseline for algorithms that solve the problem in question

#### Disadvantages:

Slow

Inefficient

Much work

BAC x  
BCA x  
CBA x  
CAB x  
ACB x  
ABC ✓

for

if

elif

looping  
iteration

+ branching  
+ comparison

**Six examples of brute force algorithms from the class notes:**

Definition (Sorting Problem).

→ Input: A sequence of  $n$  numbers  $A = [a_1, a_2, \dots, a_n]$

→ Output: A permutation  $A' = [a'_1, a'_2, \dots, a'_n]$  of the input such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

This is ascending order.

We can also sort into descending order.

python  $[q_0, q_1, \dots, q_{n-1}]$   
↓  
 $[q'_0, q'_1, \dots, q'_{n-1}]$   
sorted  
 $q'_0 \leq q'_1 \leq \dots \leq q'_{n-1}$

→ ABC  
→ 123  
→ do-re-mi

### Example 1: Selection Sort

Here is one simple method:

Find the smallest element in the sequence; swap it into slot 1.

Find the 2<sup>nd</sup> smallest element in the sequence; swap it into slot 2.

Find the 3<sup>rd</sup> smallest element in the sequence; swap it into slot 3.

etc.

Let's look at an example of the idea in action. Let the original sequence be

33; 77; 55; 22; 11; 44.

In what follows bold numbers will denote the most recently placed number.

33; 77; 55; 22; **11**; 44

This is the original array A.

**11**; 77; 55; 22; 33; 44

11; **22**; 55; 77; 33; 44

11; 22; **33**; 77; 55; 44

11; 22; 33; **44**; 55; 77

11; 22; 33; 44; **55**; 77

No further swaps are done. This is the result A'.

This method of sorting is called the Selection Sort. The selection sort algorithm is given as:

```
1 # Input: An array A and indices i and j.
2 # Output: An array where A[i] and A[j] have been swapped.
3 def Swap(A, i, j):
4     temp = A[i]
5     A[i] = A[j]
6     A[j] = temp
7
8 # Input: An array A of integers.
9 # Output: Array A sorted in increasing order.
10 def SelectionSort(A):
11     for i in range(len(A)-1):
12         m = i
13         # Find the smallest element in A[i..n-1]
14         for j in range(i+1, len(A)):
15             if A[j] < A[m]:
16                 m = j
17             Swap(A, i, m)
```

*Handwritten notes and diagrams:*

- Diagram of array A with indices i and j. A swap is shown between A[i] and A[j].
- Handwritten sequence:  $0, 1, 2, \dots, n-2$  next to the for loop in SelectionSort.
- Handwritten sequence:  $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$  next to the inner for loop.
- Handwritten sequence:  $A[i], A[m] \leftarrow A[m], A[i]$  next to the Swap function call.
- Handwritten note: "45p" next to line 18.

## Asymptotic Analysis of Selection Sort

What is our input size?

Answer:  $n$ .

What's the worst-case running time?

To formally compute the worst-case running time, we create a chart to track each line of code.

Line	Cost	Count
11	$c_1$	$n-1$
12	$c_2$	$n-1$
15	$c_3$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
16	$c_4$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
17	$c_5$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
4	$c_6$	$n-1$
5	$c_7$	$n-1$
6	$c_8$	$n-1$

We then total the cost multiplied by the count for each line.

$$\begin{aligned}
 T(n) &= c_1(n-1) + c_2(n-1) + c_3(n-1)n/2 + c_4(n-1)n/2 + c_5(n-1)n/2 + c_6(n-1) + c_7(n-1) + c_8(n-1) \\
 &= (.5c_3 + .5c_4 + .5c_5)n^2 + (c_1 + c_2 - .5c_3 - .5c_4 - .5c_5 + c_6 + c_7 + c_8)n + (-c_1 - c_2 - c_6 - c_7 - c_8) \\
 &= c_9n^2 + c_{10}n + c_{11} \\
 &\leq (c_9 + c_{10} + c_{11})n^2 \\
 &= c_{12}n^2
 \end{aligned}$$

Since  $T(n) \leq$  some constant multiple of  $n^2$ , we say

$T(n)$  is in the asymptotic class  $O(n^2)$

so  $T(n) = O(n^2)$ .

Informally, we can identify the basic operation to be the comparison in line 16. Since this is inside of a pair of nested loops with the outer loop running  $n-1$  times and the inner loop running

$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = \frac{(n-1)n}{2}$  times, the work is  $O(n^2)$ . Done.

## example 2 - Bubble Sort

Another sort that people think about when talking about brute force sorts is bubble sort.

The intuition behind the bubble sort is:

Walk through the array comparing adjacent elements in the list, exchanging them if they are out of order.

The first pass will cause the largest element to “bubble up” and end up in the last slot in the list.

Pass  $i$  bubbles up the  $i$ -th largest element in the same way.

After pass  $i$ , at least the  $i$  largest elements will be located in the last  $i$  slots of the array, so do not do any comparisons in this sorted tail portion of the array.

Let's look at an example of the bubble sort idea in action. Let the original sequence be

33; 77; 55; 22; 11; 44.

In what follows bold numbers will denote the most recently compared numbers.

33; 77; 55; 22; 11; 44

This is the original array A.

<b>33</b> ; 77; 55; 22; 11; 44	compare 33 and 77
33; <b>77</b> ; 55; 22; 11; 44	compare 77 and 55, and <u>swap them</u>
33; 55; <b>77</b> ; 22; 11; 44	compare 77 and 22, and <u>swap them</u>
33; 55; 22; <b>77</b> ; 11; 44	compare 77 and 11, and <u>swap them</u>
33; 55; 22; 11; <b>77</b> ; 44	compare 77 and 44, and <u>swap them</u>
33; 55; 22; 11; 44; <b>77</b>	

finished 1st pass through array, the last item is in its correct place

<b>33</b> ; 55; 22; 11; 44; 77	compare 33 and 55
33; <b>55</b> ; 22; 11; 44; 77	compare 55 and 22, and <u>swap them</u>
33; 22; <b>55</b> ; 11; 44; 77	compare 55 and 11, and <u>swap them</u>
33; 22; 11; <b>55</b> ; 44; 77	compare 55 and 44, and <u>swap them</u>
33; 22; 11; 44; <b>55</b> ; 77	

finished 2nd pass through array, the last 2 items are in their correct places

<b>33</b> ; 22; 11; 44; 55; 77	compare 33 and 22, and swap them
22; <b>33</b> ; 11; 44; 55; 77	compare 33 and 11, and swap them
22; 11; <b>33</b> ; 44; 55; 77	compare 33 and 44
22; 11; 33; <b>44</b> ; 55; 77	

finished 3rd pass through array, the last 3 items are in their correct places

<b>22</b> ; 11; 33; 44; 55; 77	compare 22 and 11, and swap them
11; <b>22</b> ; 33; 44; 55; 77	compare 22 and 33
11; 22; <b>33</b> ; 44; 55; 77	

finished 4th pass through array, the last 4 items are in their correct places

<b>11</b> ; 22; 33; 44; 55; 77	compare 11 and 22
11; <b>22</b> ; 33; 44; 55; 77	

finished 5th pass through array, the last 5 items are in their correct places

No further swaps are done. This is the result A'.

```

1 # Input: An array A of integers.
2 # Output: An array A sorted in increasing order.
3 def BubbleSort(A):
4     for i in range(len(A)-1):
5         for j in range(len(A)-i-1):
6             if A[j+1] < A[j]:
7                 Swap(A, j+1, j)

```

What's the worst-case running time?

Line	Cost	Count
4	$c_1$	$n-1$
5	$c_2$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
6	$c_3$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
7	$c_4$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$ for each of the three lines of Swap

*= 3 statements*

We then total the cost multiplied by the count for each line.

$$\begin{aligned} T(n) &= c_1(n-1) + c_2(n-1)n/2 + c_3(n-1)n/2 + c_4(n-1)n/2 \\ &= (.5c_2 + .5c_3 + .5c_4)n^2 + (c_1 - .5c_2 - .5c_3 - .5c_4)n + (-c_1)1 \\ &= c_5n^2 + c_6n + c_71 \\ &\leq (c_5 + c_6 + c_7)n^2 \\ &= c_8n^2 \end{aligned}$$

Since  $T(n) \leq$  some constant multiple of  $n^2$ , we say  $T(n)$  is in the asymptotic class  $O(n^2)$   
so  $T(n) = O(n^2)$ .

Informally, we can identify the basic operation to be the comparison in line 6. Since this is inside of a pair of nested loops with the outer loop running  $n-1$  times and the inner loop running

$$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = \frac{(n-1)n}{2} \text{ times, the work is } O(n^2). \text{ Done.}$$

Note: If no swaps are performed on a pass through the outer loop of bubble sort, the array is sorted. If we make the algorithm monitor the number of swaps on each pass, we can make it terminate early.

### example 3 - String Matching

Another classic problem that has a brute force solution is the string matching problem.

Definition (String Matching Problem).

Input: An  $n$  symbol string  $\sigma = \sigma_1\sigma_2\sigma_3 \dots \sigma_n$  and an  $m \leq n$  symbol search string  $s = s_1s_2s_3 \dots s_m$

Output: If there exists an index  $i$  such that  $\sigma_i = s_1, \sigma_{i+1} = s_2, \sigma_{i+2} = s_3, \dots, \sigma_{i+m-1} = s_m$  then output  $i$ ; otherwise, output None.

Alternatively, we can call these the text (what we are looking in) and the pattern (what we are looking for).

Or we can call them the haystack and the needle.

How might you go about solving this problem using brute force?

Here is one way:

Continually shift the search-string/pattern/needle one character right until it lines up.

Once you have less than  $m-1$  characters to the right of the start of the match attempt you can safely stop.

For example, let us trace the algorithm on the symbol string/text/haystack  $\sigma = \text{"ababacdabc"}$  with search string/pattern/needle  $s = \text{"abc"}$ .

	0	1	2	3	4	5	6	7	8	9
Haystack	a	b	a	b	a	c	d	a	b	c
Needle (abc)	a	b	c							
Needle (abc)		a	b	c						
Needle (abc)			a	b	c					
Needle (abc)				a	b	c				
Needle (abc)					a	b	c			
Needle (abc)						a	b	c		
Needle (abc)							a	b	c	
Needle (abc)								a	b	c

Note: The needle  $\text{"aaaab"}$  and the haystack  $\text{"aaaaaaaaaaaaaaaaaab"}$  is an example of the worst case where the maximum number of comparisons is required – every slot in the needle must be compared with the corresponding slot in the haystack before sliding the needle one position to the right.

In pseudo-code we have the following string match algorithm:

```

1  # Input: An array A of n symbols and an array S of m < n
2  # symbols.
3  # Output: The index i of the start of the pattern in A or
4  # None, otherwise.
5  def StringMatch(A, S):
6      for i in range(len(A)-m+1):
7          j = 0
8          while j < len(S) and A[i+j] == S[j]:
9              j = j+1
10         if j == len(S):
11             return i
12     return None

```

*n*  
haystack

*m*  
needle

*i = 0 1 2 3 ... n-m*

*match / found*

*no match / not found*

What is our input size?

Answer:  $(n, m)$ .  $n$  for the length of the haystack and  $m$  for the length of the needle

What's the worst-case running time?

Line	Cost	Count
6	$c_1$	$n-m+1$
7	$c_2$	$n-m+1$
8	$c_3$	$m(n-m+1)$
9	$c_4$	$m(n-m+1)$
10	$c_5$	$n-m+1$
11	$c_6$	1 at most
12	$c_7$	1 at most

*m < n*  
needle haystack

*$c_6 + c_7$*

$$\begin{aligned}
 T(n, m) &= c_1(n-m+1) + c_2(n-m+1) + c_3 m \cdot (n-m+1) + c_4 m \cdot (n-m+1) + c_5(n-m+1) + \max(c_6, c_7) \\
 &= (c_3 + c_4)nm + (-c_3 - c_4)m^2 + (c_1 + c_2 + c_5)n + (-c_1 - c_2 + c_3 + c_4 - c_5)m + (c_1 + c_2 + c_5 + \max(c_6, c_7))1 \\
 &= c_8 mn + c_9 m^2 + c_{10} n + c_{11} m + c_{12} 1 \\
 &\leq (c_8 + c_9 + c_{10} + c_{11} + c_{12})nm \\
 &= c_{13} nm
 \end{aligned}$$

*upper bound*

Since  $T(n, m) \leq$  some constant multiple of  $nm$ ,  
 $T(n, m)$  is in the asymptotic class  $O(nm)$   
 so  $T(n) = O(nm)$ .

*Worst Case*

*len(haystack) = n → aaaaaaaaaaaaaaaaaa b*  
*len(needle) = m → aab*



We will consider two very fundamental problems in computational geometry:

1. The closest-pair problem
2. The convex-hull problem

#### example 4 - Closest Pair

Definition (Closest Pair Problem).

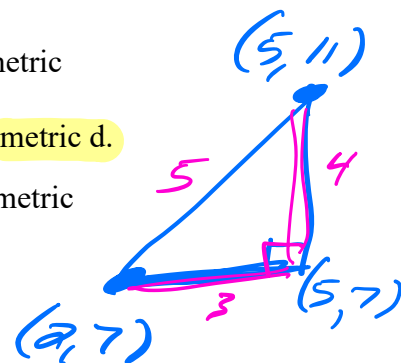
Input: A set  $P = \{p_1, p_2, p_3, \dots, p_n\}$  of  $n$  points in  $d$ -dimensional space and a metric  $d: P \times P \rightarrow R$

Output: The distance between the two closest points  $p_i$  and  $p_j$  according to the metric  $d$ .

For our purposes we will only consider the 2-D Euclidean plane and with the metric

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

*Handwritten calculation for points (2,7) and (5,11):*  
 $(2-5)^2 + (7-11)^2 = 3^2 + 4^2 = 9 + 16 = 25$   
 $\sqrt{25} = 5$



What is the brute force approach here?

For every pair of points, compute the distance between the two points and find the pair with the smallest distance.

Formally, we have the algorithm:

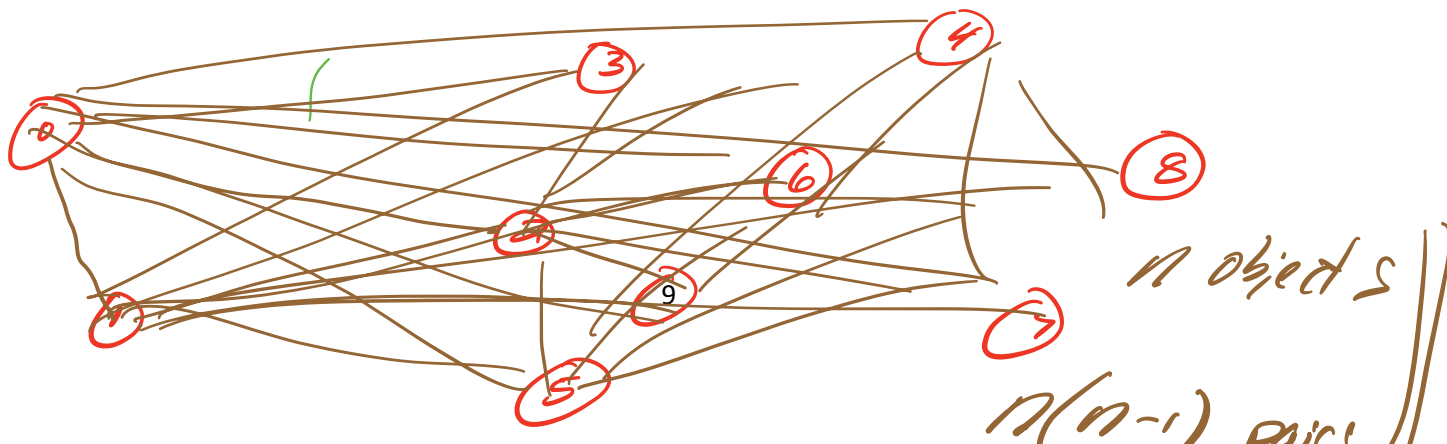
```

1  # Input: Two points A and B in the XY plane.
2  # Output: The Euclidean distance between the two points.
3  def Distance(A, B):
4  → return math.sqrt((A[0]-B[0])**2 + (A[1]-B[1])**2)
5
6  # Input: P a list of n>2 points in the XY plane.
7  # Output: The distance between the closest pair of points.
8  def ClosestPair(P):
9      d = 1e12 # a large number, might as well be infinity
10     for i in range(len(P)-1):
11         for j in range(i+1, len(P)):
12             d = min(d, Distance(P[i], P[j]))
13     return d

```

*Handwritten note:* or  $d = \text{Distance}(P[i], P[j])$  on actual value

Note: `math.sqrt()` might require you to import `math`.



## Asymptotic analysis

What is our input size?

Answer:  $n$  = number of points

$n$  points  
so

$\frac{n(n-1)}{2}$  pairs of points

What's the worst-case running time?

Line	Cost	Count
9	$c_1$	1
10	$c_2$	$n-1$
11	$c_3$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
12	$c_4$	$(n-1)+(n-2)+(n-3)+\dots+3+2+1 = (n-1)n/2$
13	$c_5$	1

$$\begin{aligned}T(n) &= c_1(1) + c_2(n-1) + c_3 (n-1)n/2 + c_4 (n-1)n/2 + c_5 (1) \\&= (.5c_3 + .5c_4)n^2 + (c_2 - .5c_3 - .5c_4)n + (c_1 - c_2 + c_5)1 \\&= c_6n^2 + c_7n + c_81 \\&\leq (c_6 + c_7 + c_8)n^2 \\&= c_9n^2\end{aligned}$$

Since  $T(n) \leq$  some constant multiple of  $n^2$ ,  
 $T(n)$  is in the asymptotic class  $O(n^2)$   
so  $T(n) = O(n^2)$ .



### example 5 - Convex Hull

We now turn our attention to what is considered the most important problem in computational geometry. The convex hull problem. Applications include:

- Collision detection in video games.
- Path planning
- Processing satellite maps for accessibility
- Detecting statistical outliers
- Computing the diameter of a set of points (i.e. the distance between the two farthest points).

First let's define what it means for a set to be convex.

Definition (Convex Set). A set of points (finite or infinite) in the plane is called convex if for any two points  $p$  and  $q$  in the set, the entire line segment  $pq$  belongs to the set.

Examples of convex sets are the set of points that make up a square, rectangle, parallelepiped, etc. In fact, the vertices of any convex polygon form a convex set.

The convex hull of a set of points is the boundary of the region in the plane that includes the line segments joining all pairs of points in the set.

“Same Side Test”: A line segment connecting two points  $p_i$  and  $p_j$  of a set of  $n$  points is part of the convex hull's boundary if all other points of the set lie on the same side of the straight line through these two points.

Given a set of  $n$  points in the plane we can think of the convex hull as the smallest convex polygon such that the points in the set are either inside or on the boundary of the polygon.

Exams of convex hulls are:

If  $S$  is a single point,  $S$  is its own convex hull.

If  $S$  is 2 points the convex hull is the line segment joining the two points.

If  $S$  is three points not on the same line then the convex hull is a triangle.

We are now equipped to define the convex hull problem.

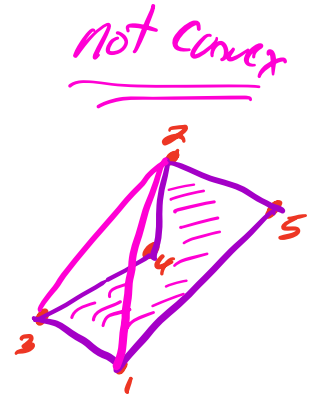
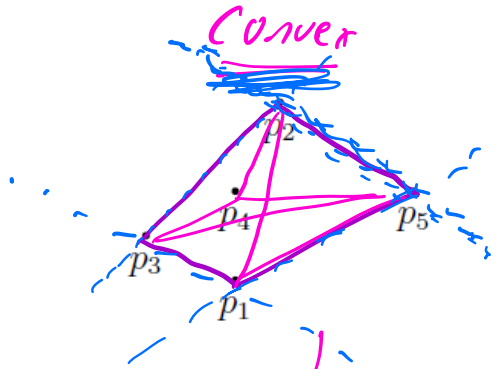
Definition 8 (Convex Hull Problem).

Input: A set  $S$  of  $n$  points

Output: The list of points that form the vertices of the convex polygon

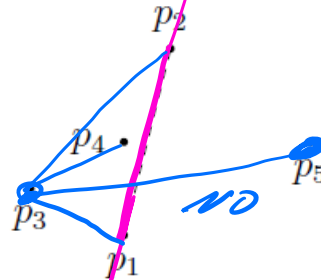
Our brute force algorithm involves repeating the “Same Side Test” for every pair of points in the set  $S$ , and compiling a list of points whose line segments pass the test.

We will not provide pseudocode for this algorithm, but we will work through a few steps on the following example set of points:



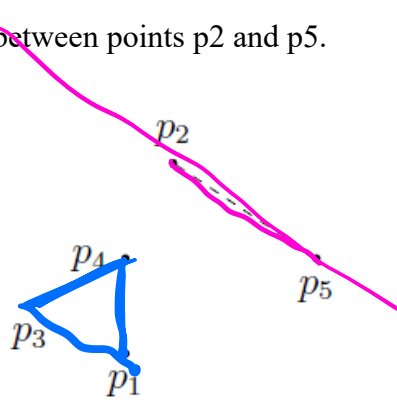
Bad segment example: Test the line between points  $p_1$  and  $p_2$ .

Same Side Test



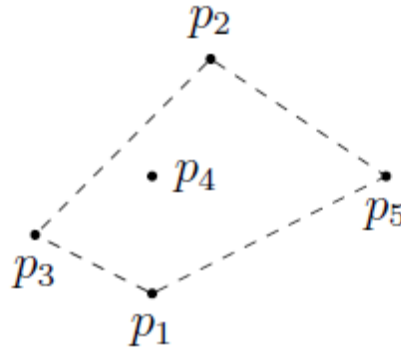
This line segment “splits the plane” in such a way that there are points on both sides of the line segment. Therefore, this segment cannot be part of the convex hull.

Good segment example: Test the line between points  $p_2$  and  $p_5$ .



This line segment “splits the plane” in such a way that all remaining points lie on the same side of the line segment. Therefore, this segment is part of the convex hull.

The convex hull for the entire set is



so, our algorithm would return the subset of points  $\{p_1, p_2, p_3, p_5\}$  that lie on the convex hull.

This is a very inefficient way of solving the problem (as most brute force algorithms are). For every pair of points, we need to consider the line segment joining these two points and check to see if all remaining points lie on the same side of the line segment.

For a set of  $n$  points, how many line segments are there? There is one line segment for each pair of points, so there are “ $n$  choose 2” =  $nC2 = \binom{n}{2} = \frac{n(n-1)}{2}$  segments. For each line segment, we need to determine on which side of the line does each of the remaining  $n-2$  points lie. How we do this is quite complicated. But clearly, this is the essential unit of work that we need to count. The counting is simple in the worst case:

$$\frac{n(n-1)}{2} (n-2) = \frac{(n^2 - n)(n-2)}{2} = \frac{n^3 - 3n^2 + 2n}{2} < n^3$$

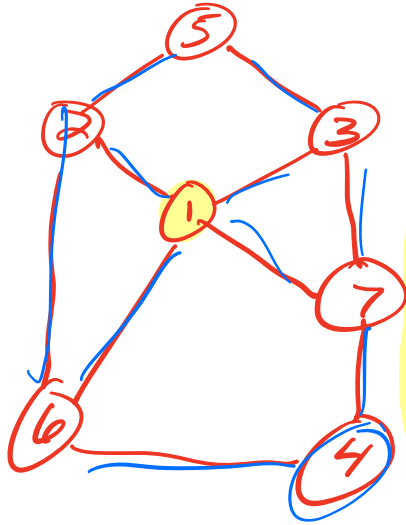
so, this brute force algorithm for the convex hull of a set of  $n$  points is  $O(n^3)$ .

For every  
pair of  
points,  
draw the  
line they  
define  
and ask

are all  
other points  
on same side  
of this line?

If YES, keep this  
segment as  
boundary of  
convex hull.

# Graph

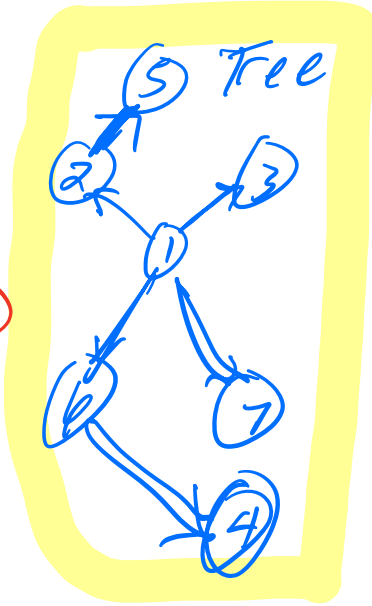


$n=7, e=10$

## Adjacency Lists

1	2	3	6	7
2	1	5	6	
3	1	5	7	
4	6	7		
5	2	3		
6	1	2	4	
7	1	3	4	

# BFS



# Queue

[]  
 [1]  
 [1, 2]  
 [1, 2, 3]  
 [1, 2, 3, 6]  
 [1, 2, 3, 6, 7]  
 [2, 3, 6, 7]  
 [2, 3, 6, 7, 5]  
 [3, 6, 7, 5]  
 [6, 7, 5]

[6, 7, 5, 4]  
 [2, 5, 4]  
 [5, 4]  
 [4]  
 []

## Processing Order

1 2 3 6 7 5 4

## Matrix

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

## Searching in Graphs

Week 6

We can use exhaustive search to process all vertices and edges in a graph.

There are two such algorithms we will investigate to perform this type of search

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)

Both types of graph searches are extremely frequent in Computer Science. We will implement both algorithms using the adjacency list representation of a graph.

BFS is a brute-force algorithm that uses a queue to organize the work to be done next.

Algorithm DFS is a recursive algorithm; we will cover it right away in the next class on recursive algorithms because it is logically related to BFS. We could have implemented DFS as a brute-force algorithm using a stack, but that is exactly what is taking place behind the scenes in recursion.

### example 6 - Breadth First Search (BFS)

The basic idea of Breadth First Search is to start at a chosen vertex, visits all of its not-yet-visited neighbors, then all of their not-yet-visited neighbors, ... If the graph is connected (if there is a path from the source vertex to every other vertex), all vertices will eventually be visited. The algorithm gets its name from its attempt to go as wide as possible before proceeding to vertices that are another level away from the source.

A high-level algorithm using a queue is as follows:

We use a queue to maintain the list of discovered vertices.

Queue initially contains the source vertex.

At each iteration of the BFS algorithm:

1. Enqueue all vertices adjacent to the vertex that is at the front of the queue.
2. Mark each newly inserted vertex visited.
3. Dequeue the vertex at the front of the queue.

If we finish our search from the source vertex, and not all nodes have been visited we must start BFS again from an arbitrary unvisited vertex.

When a visited vertex considers one of its visited neighbors, we insert a new “cross edge” that goes from the vertex to its previously visited neighbor.

Like DFS the pseudo-code for BFS is divided into two algorithms BFS and BFSVisit.



The BFS algorithm is as follows:

```

1  # Input: A graph G = (V, E) and a global variable, count.
2  # Output: A graph G = (V, E') where E' = E and V' = V
3  #      except that the nodes of V' are marked with integers
4  #      that indicate in what order they were visited.
5  def BFS(G):
6      global count
7      count = 0
8      # Mark every vertex in V with 0 to indicate not yet visited.
9      for v in G:
10         v.visited = 0
11     # Start BFSVisit algorithm for the first unvisited node.
12     # Restart as needed (if graph is not connected).
13     for v in G:
14         if v.visited == 0:
15             BFSVisit(v)

```

Graph  $G = (V, E)$

$|V| = n$  nodes/vertices

$|E| = e$  edges

The BFSVisit algorithm is as follows:

```

1  # Input: v is an unvisited vertex.
2  # Output: All descendants of v have been visited.
3  def BFSVisit(v):
4      count = count + 1
5      v.visited = count
6      # Create a queue, enqueueing the vertex.
7      Q = [v]
8      while Q is not empty:
9          w = dequeue(Q)
10         # for each unvisited neighbor of w, visit it, mark it as visited, and enqueue it
11         for u in w.getAdjacent():
12             if u.visited == 0:
13                 count = count + 1
14                 u.visited = count
15                 Q.append(u)

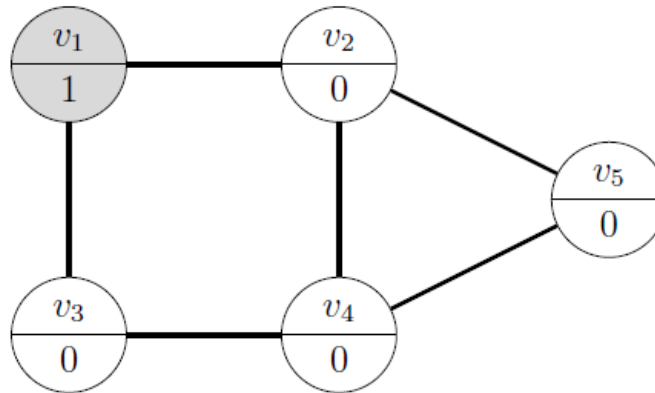
```

enqueue/append - L7 or L15

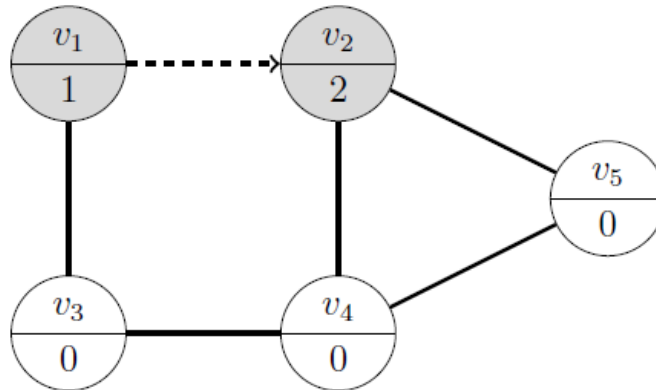
Examine edge - L11

Let's look at an example trace of BFS on a small graph. In what follows, the numbers represent the visit counter value, the tree edges are denoted by dashed lines with arrows, and the cross edges are denoted by dotted lines with arrows.

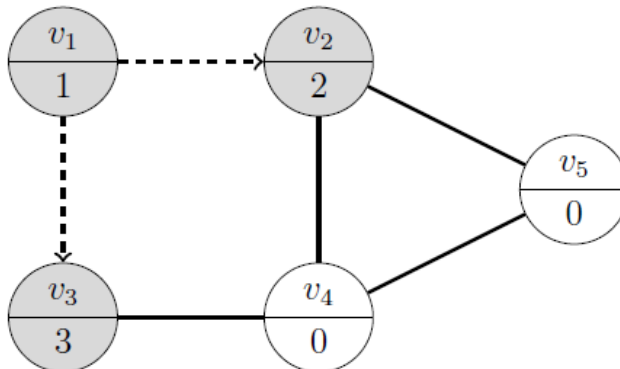
1. The source vertex is  $v_1$  and  $Q = [v_1]$ . Mark  $v_1$  as the 1<sup>st</sup> visited vertex.



2. We visit the first descendant of  $v_1$  and  $Q = [v_1, v_2]$ . Mark  $v_2$  as the 2<sup>nd</sup> visited vertex. Make edge  $(v_1, v_2)$  a tree edge.

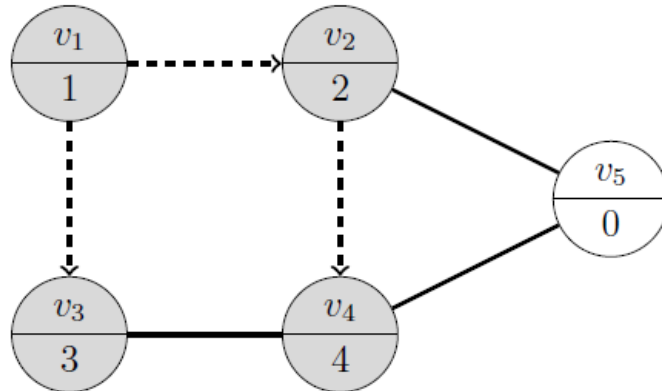


3. We visit the next descendant of  $v_1$  and  $Q = [v_1, v_2, v_3]$ . Mark  $v_3$  as the 3<sup>rd</sup> visited vertex. Make edge  $(v_1, v_3)$  a tree edge.

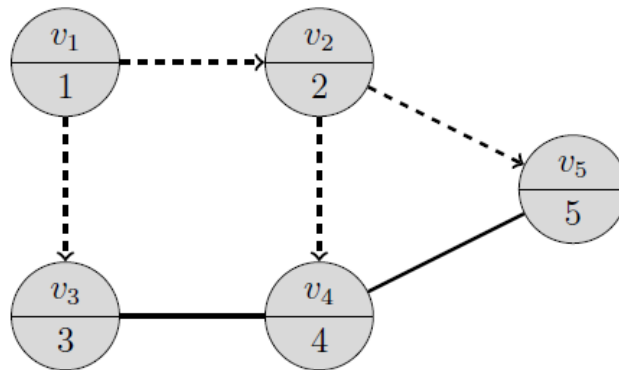


4. Vertex  $v_1$  has no more descendants, it is removed from the queue. The queue is now  $Q = [v_2, v_3]$ .

5. We visit the first descendant of  $v_2$ .  $Q = [v_2, v_3, v_4]$ . Mark  $v_4$  as the 4th visited vertex. Make edge  $(v_2, v_4)$  a tree edge.

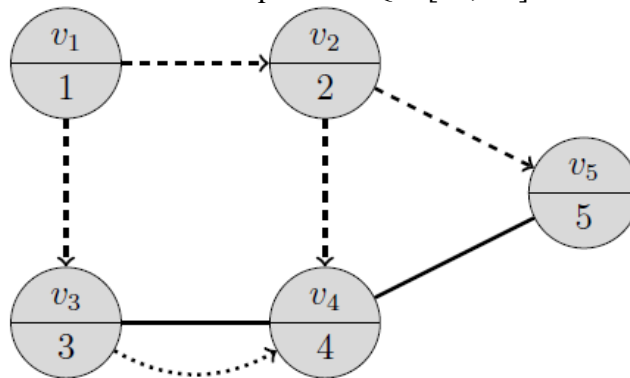


6. We visit the second descendant of  $v_2$ .  $Q = [v_2, v_3, v_4, v_5]$ . Mark  $v_5$  as the 5th visited vertex. Make edge  $(v_2, v_5)$  a tree edge.

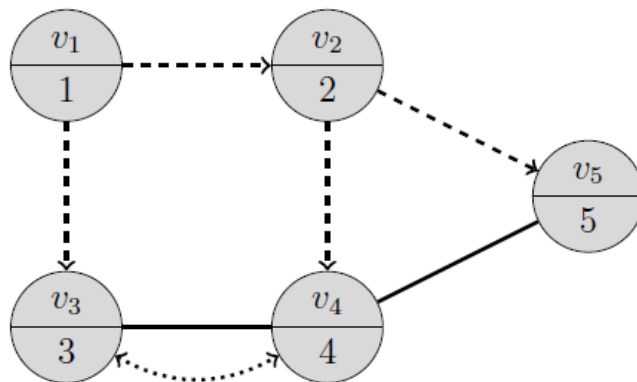


7. Vertex  $v_2$  has no more descendants, it is removed from the queue. The queue is now  $Q = [v_3, v_4, v_5]$ .

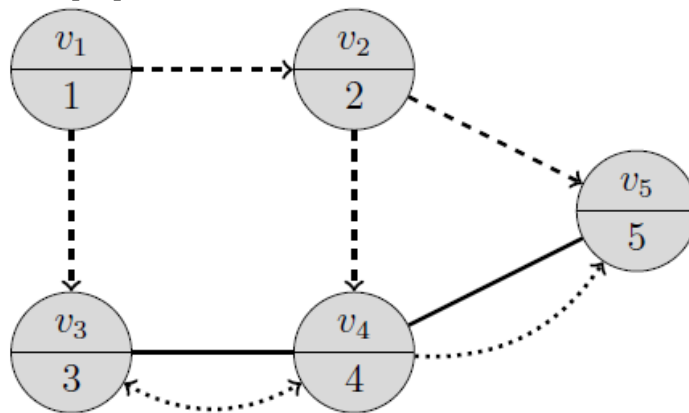
8. Vertex  $v_3$ 's only descendant is  $v_4$  which has already been visited. This means  $(v_3, v_4)$  is a cross edge. Vertex  $v_3$  is removed from the queue so  $Q = [v_4, v_5]$ .



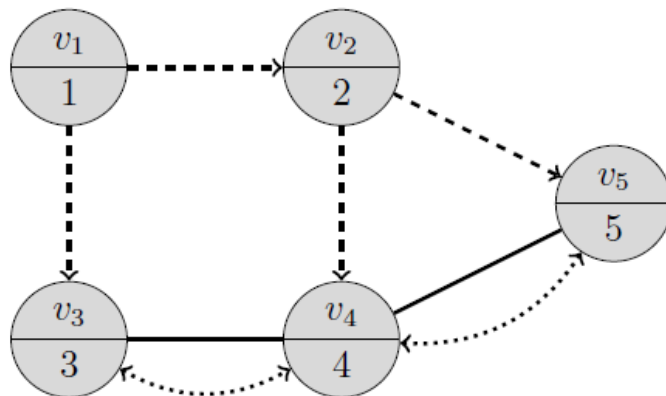
9. Vertex  $v_4$ 's first descendant is  $v_3$  which has already been visited this means  $(v_4, v_3)$  is a cross edge.



10. Vertex  $v_4$ 's second descendant is  $v_5$  which has already been visited. This means  $(v_4, v_5)$  is a cross edge. Since, there are no additional descendants  $v_4$  is removed from the queue. This results in  $Q = [v_5]$ .



11. Vertex  $v_5$ 's only descendant is  $v_4$  which has already been visited. This means  $(v_5, v_4)$  is a cross edge.  $v_5$  is removed from the queue so  $Q = []$  and our BFS has finished since there are no unvisited vertices in  $G$ .



What is the work done by the BFS algorithm in the worst case?

We will not do this formally with a chart. Instead, we identify the crucial unit of work to be considering whether or not to enqueue a vertex.

In BFS(), after marking each vertex as unvisited, the algorithm calls BFSVisit() at most once for each vertex (lines 13-15) and in BFSVisit, the vertex is added to the queue (line 7). So, this bound is  $|V|$ , the number of vertices in the graph.

In BFSVisit(), lines 11-15 examine every edge from this vertex to decide whether or not to enqueue each of its neighbors. Since each edge appears twice in the graph's adjacency structure, this bound is  $2|E|$  or twice the number of edges in the graph.

Therefore, the number of "enqueings" that are considered is  $\leq |V| + 2|E| < 2(|V| + |E|)$ , so the BFS algorithm's asymptotic class is  $O(|V| + |E|)$ .

$$\begin{aligned} T(n,e) &\rightarrow \underline{n + 2e} \\ &< 2n + 2e \\ &= 2(n+e) \\ &\rightarrow \underline{\underline{O(n+e)}} \end{aligned}$$