What is sorting?

- Input: sequence of n comparable values
- Reorder the input to be non-descending.
- Items we wish to sort are called "keys"
- Not here: retain associated information

Why discuss sorting?

- Standard library has sorting
- Why not use that and move on?

In this class, sorting is:

- a good intro for techniques
- a good intro to comparative algorithms

Bubble Sort

Idea: Think globally act locally

for $i \leftarrow 1$ to $n-1$ **do**
  for $j \leftarrow 1$ to $n-i$ **do**
    if $A[j+1] < A[j]$ **then**
      Swap $A[j]$ and $A[j+1]$

85	24	63	45	17	31	96	50

SelectionSort

Idea: Swap min into first spot, second-min to second, etc.

```
for i ← 1 to n − 1 do
    min ← i
    for j ← i + 1 to n do
        if A[j] < A[min] then
            min ← j
    Swap A[i] and A[min]
```

85	24	63	45	17	31	96	50

Let's talk about SelectionSort.

Question 1. Does SelectionSort waste memory?

Question 2. Does it only work for numbers?

Question 3. What other information do we need in order to run SelectionSort?

Question 4. Are there inputs that are sorted faster?

Question 5. Is there a lot of data movement?

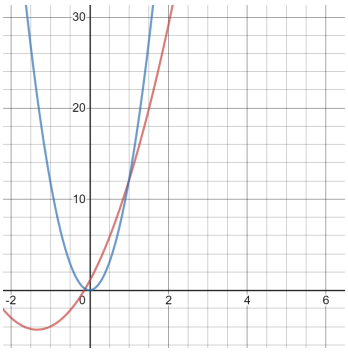
Question 6. A good question to ask is: "if I double the size of the input, how much longer does the algorithm take?" For each of the following operation counts, how much longer will the algorithm take if we do this?

Time	Change	Time	Change
n		$20n$	
$10n + 37$		n^2	
n^4		$n^5 + 10n^3 + 21$	

Notation for growth of functions

We say that $f(n)$ is $\mathcal{O}(g(n))$ (read: f of n is big-oh of g of n) if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that for all $n > n_0$, $f(n) \leq cg(n)$.

Question 7. Use this definition to show that $3n^2 + 12n + 1$ is $\mathcal{O}(n^2)$.



Question 8. Find the $\mathcal{O}$-notation for the worst-case running time of the following algorithms. You do not need to, nor should you, give the leading constant or the n_0 value. For binarySearch, assume the first listed function is called with a *sorted* vector. For linearSearch, does your answer depend on whether or not the vector is sorted?

```
int linearSearch(const std::vector<int> & numbers, int target)
{
    int i;
    int n = numbers.size();
    for(i=0; i < n; i++)
    {
        if( numbers[i] == target )
        {
            return i;
        }
    }
    throw ElementNotFoundException("Element not found by linear search.");
}
```

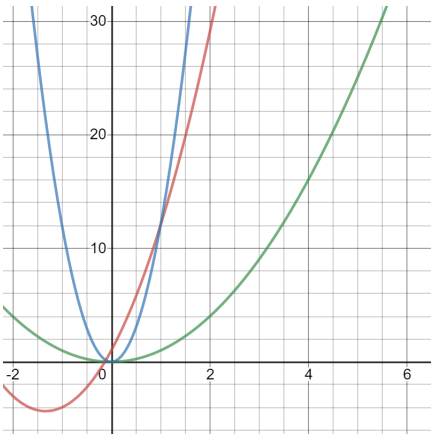
```
int binarySearch(const std::vector<int> & numbers, int target);
int binarySearch(const std::vector<int> & numbers, int target, int low, int high);


int binarySearch(const std::vector<int> & numbers, int target)
{
    return binarySearch(numbers, target, 0, numbers.size() - 1);
}

int binarySearch(const std::vector<int> & numbers, int target, int low, int high)
{
    if( low > high )
    {
        throw ElementNotFoundException("Element not found by binary search.");
    }
    int mid = (low + high) / 2;
    int e = numbers[mid];
    if( e == target )
    return mid;
    else if (target < e)
        return binarySearch(numbers, target, low, mid-1);
    else
        return binarySearch(numbers, target, mid+1, high);
}
```

2	4	5	7	8	9	12	14	17	19	22	25	27	28	33

Question 9. We saw that $\mathcal{O}$ notation provides an *upper bound on the growth rate of a function.* What are Ω and Θ used to describe?



The graph images were created with the graphing utility at www.desmos.com/calculator. I encourage students to create such graphs in order to better visualize what $\mathcal{O}$ notation and related concepts **mean** rather than memorizing the mechanics of them. Graph the function you want to find the asymptotic notation for, then also find appropriate values for n_0 and c and graph $cg(n)$.

Question 10. $f(n) = \log_{10} n$ and $g(n) = \log_2 n$. How do they relate?

Question 11. $f(n) = \log n$. What base do I mean?

Hierarchy of Running Times

Rank the following running times from smallest (slowest growing) to largest (fastest growing), so that if f_x appears before f_y in your listing, then f_x is $\mathcal{O}(f_y)$.

- $f_A = 2^{100n}$
- $f_B = 2^{n^2}$
- $f_C = 2^{n!}$
- $f_D = 2^{2^n}$
- $f_E = n^{\log n}$
- $f_F = n \log n \log \log n$
- $f_G = n^{3/2}$
- $f_H = n \log^{3/2} n$
- $f_I = n^{4/3} \log^2 n$

In principle, you could perform a find-min style operation to determine which of these is the slowest growing, write it on your answer, cross it off the list, and repeat. However, we should first classify each into a hierarchy. In general, the running times we see tend to fit into these categories:

1. Constant time, such as pushing to a linked-list based stack.
2. Poly-logarithmic; that is, a function that is a polynomial function of the *logarithm* of the input sized. If the input is sized n, then $\mathcal{O}(\log n)$ is poly-logarithmic, as is $\mathcal{O}(\log^2 n)$, but $\mathcal{O}(n \log n)$ would not be.
3. Polynomial; a polynomial function of the size of the input. $\mathcal{O}(n^2)$ is polynomial if n is the size of the input (such as the number of elements in a vector), but not n is an integer input (as the input size would be $\mathcal{O}(\log n)$ bits).
4. Worse than polynomial, such as exponential. In general, you want to avoid these running times unless you either have a small enough input or they are absolutely unavoidable.

InsertionSort

```
for j ← 2 to n do
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] >key do
        A[i + 1] ← A[i]
        i = i − 1
    A[i + 1] ← key
```

85	24	63	45	17	31	96	50

Question 12. What is the worst-case running time of InsertionSort?

Question 13. Why is InsertionSort correct?

Question 14. What is true *every time* we check the **for** loop?
(including the time we find $j > n$ and stop)

Question 15. The ordered pair (i, j) is called an *inversion* in a permutation of the first n positive integers if $i < j$ but j preceeds i in the permutation. For example, there are six inversions in the permutation $3, 5, 1, 4, 2$.

What is the expected number of inversions in a permutation of the first n positive integers, assuming all permutations are equally likely?

Question 16. What is the average number of comparisons used by InsertionSort to sort an array of n distinct elements?

To solve this, let's begin with allowing X to be the random variable equal to the number of comparisons used by the algorithm. We can separate it into a sum of X_i values, where X_i is the random variable equal to the number of comparisons used to insert a_i into the proper position after the first $i - 1$ elements have been sorted.
Accordingly, $X = X_2 + X_3 + \ldots + X_n$ Similarly, $E(X) = E(X_2) + E(X_3) + \ldots + E(X_n)$
We now need only to determine each $E(X_i)$.

Heaps and HeapSort

Using the following 1-based array:

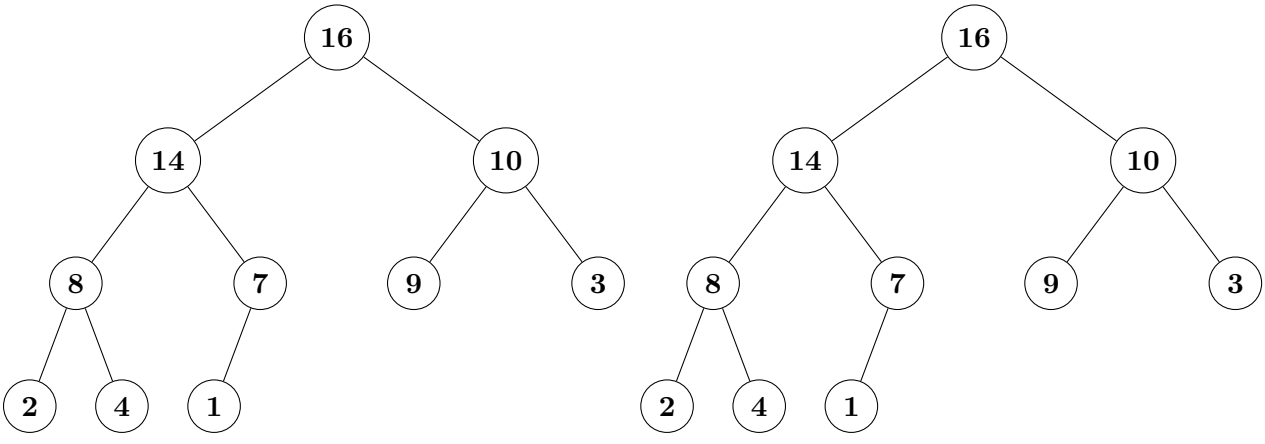
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

- What is that array, interpreted as a binary tree (for heap purposes)?
- Where is the parent of node i?
- Where is left child of node i?
- Where is right child of node i?
- What is a **complete binary tree**?

- What is the **max heap property**?

- How tall is a heap?

Here is the same heap, drawn twice, as you will want to "reset" during the lecture at one point:



HeapSort

Idea 1: Insert all n elements into an (initially empty) max heap. Remove the max element, placing it in the original array's position n. Then remove the new max, placing it in position $n-1$. Continue in this fashion.
Question 17. How long does this take?

Idea 2: Bottom-up heap construction. We know which locations will be leaf nodes.

4	1	3	2	16	9	10	14	8	7

Question 18. Once we have the array turned into a max-heap, what do we do? Where do you place the result of a `remove-max` operation?

Lower Bound on Comparison-Based Sorting

We have seen algorithms that take $\mathcal{O}(n^2)$ time. We saw HeapSort which takes $\mathcal{O}(n \log n)$ time. You might also be familiar with TreeSort, MergeSort, and QuickSort, which take (or can take) $\mathcal{O}(n \log n)$ time. You might wonder: are there algorithms which are strictly better than $\mathcal{O}(n \log n)$ time for a general comparison-based sort?

To answer this question, we will build a **decision tree** that represents any comparison based sorting algorithm. Such an algorithm will ask questions of the form "is $x_i < x_j$?"

Question 19. What are leaf nodes of decision tree?

Question 20. What are internal nodes?

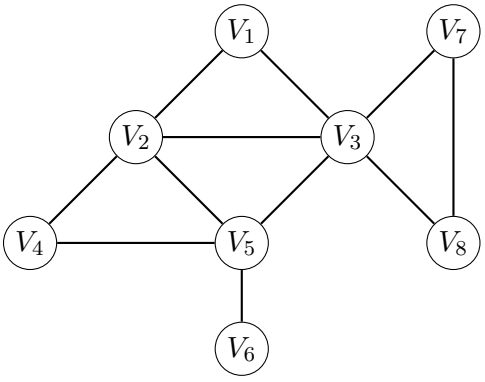
Question 21. What is height of the tree?

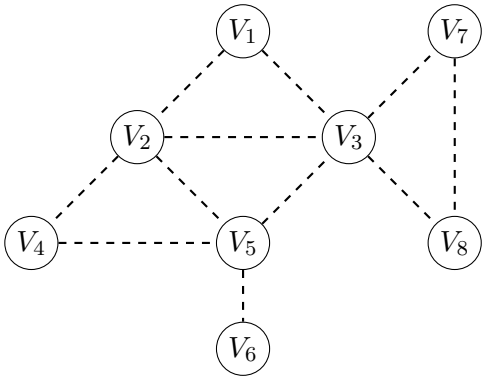
Question 22. What does this tell us about any such algorithm?

Depth-First Search

Given a graph G and two nodes s and t, how could we find a path from s to t?

For example, how do we find a path from V_1 to V_6 in the following graph?





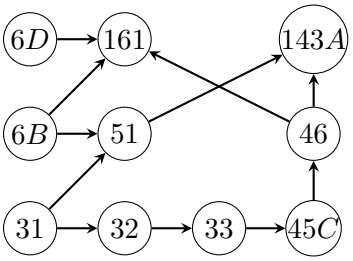
Depth-first search creates a tree, rooted at the starting vertex. Because the path between two vertices in a tree is unique, we can say that the path found is the unique path from the root to any given vertex. We can express the code iteratively or recursively:

```
DFS-recursive(u)
  Mark u as "discovered"
  for each edge (u, v) do
    if v is not marked "discovered" then
      DFS-recursive(v)
```

```
DFS-iterative(s)
  ∀_v discovered[v] = false
  Initialize S : a stack with s as only element
  while S is not empty do
    u ← pop(S)
    if discovered[u] = false then
      discovered[u] = true
      for all edges (u, v) do
        push(S, v)
```

Question 23. We can see that depth-first search finds *some path* between v_1 and v_6; does this find the *shortest path*?

Topological Sort



This directed graph is a **prerequisite graph**.

An edge (u, v) indicates you must take class u before you take class v. What would the neighbor set $N(u)$ indicate?

Suppose your friend wants to minor in ICS and choose the following classes listed in the graph above. They have ten quarters left, and have room to take one course per quarter. Give a valid ordering in which they can complete the minor at one class per quarter, while respecting prerequisite requirements.

$\deg^-(v)$ is the in-degree of vertex v, indicating the number of edges with v as their destination.

$\deg^+(v)$ is the out-degree of vertex v, indicating the number of edges with v as their start point.

$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$

Topological Sort

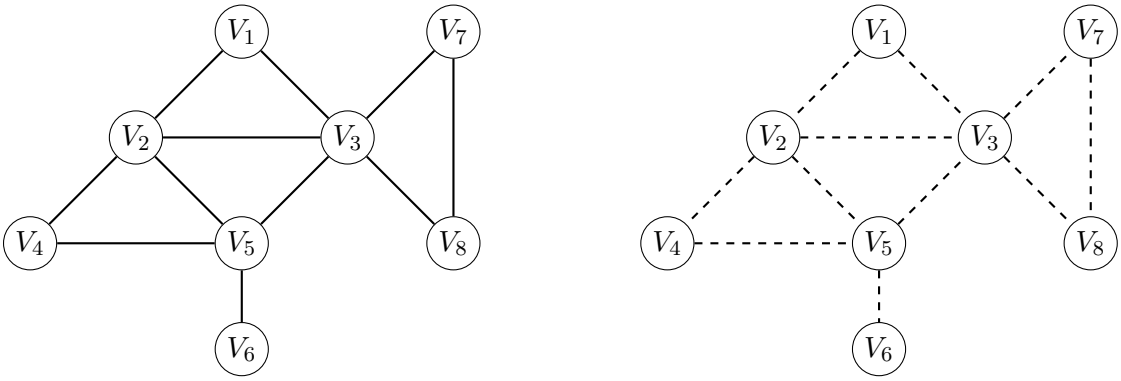
A *topological order* of a directed graph is an ordering of the vertices such that, if v_i appears before v_j, then there is no path from v_j to v_i in the graph. For example, on the above graph, 51 cannot be before 31 or $6B$.

1. What must be true about the graph in order for it to have a topological order?

2. How do we find a topological order *on paper*?

3. How do we find a topological order by computer?

Breadth-First Search



Like DFS, this algorithm will create a tree rooted at the start vertex.

```
BFS(G, s)
  Set discovered[s] = true and discovered[v] = false for all other v
  L[0] ← {s}
  i ← 0
  while L[i] is not empty do
    Make L[i + 1] as empty list
    for all vertices u ∈ L[i] do
      for all edges (u, v) do
        if discovered[v] = false then
          discovered[v] ← true
          Add v to list L[i + 1]
    i ← i + 1
```

L_i consists of all nodes whose shortest path to s is length exactly i. If $(x, y) \in E$, then x, y differ by at most one level.

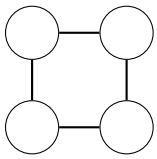
Question 24. Does Breadth-First Search find the shortest path between the start vertex and any other vertex? Do you have to make any assumptions about the edges' relative importance to say this?

Graph Coloring

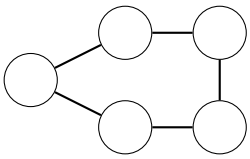
The **Graph Coloring** problem is as follows. We are given a simple, undirected graph $G = (V, E)$ and a natural number $k \geq 2$. Our goal is to either assign to each vertex one of k distinct colors in such a way that every edge is dichromatic (two different color endpoints) or to correctly report that this cannot be done for this graph with this choice of k.

Any graph for which $k = 2$ can be done is called **bipartite**. The **chromatic number** of a graph ($\chi(G)$) is the smallest value of k for which we can solve the graph coloring problem on G.

Question 25. Which (neither, one or the other, or both) of the following graphs is bipartite? Justify your answer.



C_4



C_5

Question 26. How can we use Breadth-First Search to determine if a graph is bipartite?

Question 27. If the output to the previous solution says the graph is bipartite, how could we convince a third party of it? That is, use the output to convince someone who won't look at the algorithm that the graph is bipartite.

Question 28. Similarly, if the graph is **not** bipartite, how could we convince a third party of it?