CS 106X, Lecture 27
Polymorphism; Sorting

reading:

Programming Abstractions in C++, Chapter 19, Chapter 10

Plan For This Week

- Graphs: Topological Sort (HW8)
- Classes: Inheritance and **Polymorphism** (HW8)
- **Sorting Algorithms**

Plan For Today

- **Recap:** Inheritance
- Polymorphism
- Announcements
- Sorting Algorithms

- **Learning Goal 1:** understand how to create and use classes that build on each other's functionality.
- **Learning Goal 2:** understand different ways to sort data, and how to analyze and understand their implementations and tradeoffs.

Plan For Today

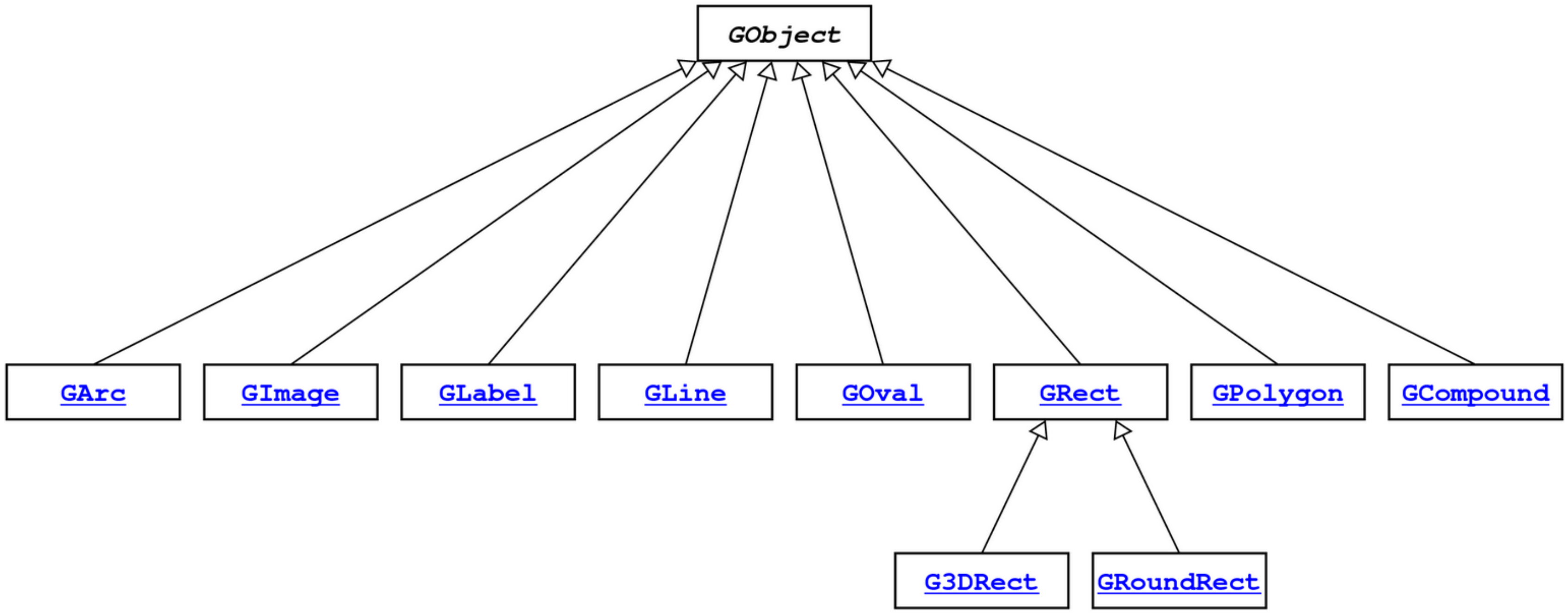
- **Recap: Inheritance and Composition**
- Polymorphism
- Announcements
- Sorting Algorithms

Inheritance vs. Composition

- **Inheritance** lets us relate our variable types to one another with *is-a* relationships ("A Lawyer is an Employee")
  - Good for when you are *extending* existing behavior
  - It makes sense to call existing methods on new type
- **Composition** lets us relate our variable types to one another with *has-a* relationships ("A sorted vector has a vector")
  - Good for when you are *utilizing* existing behavior
  - It doesn't always make sense to call existing methods on new type
- Composition or Inheritance?
  - I have a FileDownloader class, and I want to design a FileHandler class that both downloads *and* processes the file
  - I have a Book class, and I want to design an Anthology class

Example: GObjects

- The Stanford library uses an inheritance hierarchy of graphical objects based on a common superclass named **`GObject`**.



Lawyer.h

```
class Lawyer : public Employee {
public:
    Lawyer(const string& name, int yearsWorked,
           const string& lawSchool);
    void assignToClient(const string& clientName);
    ...

private:
    string lawSchool;
    Vector<string> clientNames;
    ...
};
```

Initialization

- When a subclass is initialized, C++ automatically calls its superclass's *0-argument constructor.*
  - **Intuition:** the "superclass" portion of the object must always be initialized. The subclass doesn't have access to private members to do this!
- If there is no *0-arg constructor*, or if you want to initialize with a different superclass constructor:

initialization list里面直接调用父类constructor实现

```
SubclassName::SubclassName(params)
            : SuperclassName(params) {
    statements;
}
```

Initialization

- When a subclass is initialized, C++ automatically calls its superclass's *0-argument constructor.*
  - **Intuition:** the "superclass" portion of the object must always be initialized. The subclass doesn't have access to private members to do this!
- If there is no *0-arg constructor*, or if you want to initialize with a different superclass constructor:

```
Lawyer::Lawyer(const string& name, int yearsWorked,
        const string& lawSchool) : Employee(name, yearsWorked) {

        // calls Employee constructor first
       this->lawSchool = lawSchool;
}
```

Overriding

- In addition to **adding new behavior** in our subclass, we may also want to **override existing** behavior, meaning replace a superclass's member function by writing a new version of that function in a subclass.
- To override a function, declare it in the superclass using the **virtual** keyword. This means subclasses can override it.

```
// Employee.h
virtual string getName();

// Employee.cpp
int Employee::getHoursWorkedPerWeek() {
    return 40;
}
```

```
// headta.h
string getName();

// headta.cpp
int HeadTA::getHoursWorkedPerWeek() {
  // override!
  return 20;
}
```

Overriding

- Sometimes, an overridden member may want to depend on its superclass's implementation.
  - **E.g.** a Head TA works half as many hours as a full-time employee

```
// Employee.h
int Employee::getHoursWorkedPerWeek() {
    return 40;
}

// HeadTA.h
int HeadTA::getHoursWorkedPerWeek() {
    return 20;
}
```

This implementation means we must change 2 files if an employees standard work hours are changed!

Overriding

- Sometimes, an overridden member may want to depend on its superclass's implementation.
  - **E.g.** a Head TA works half as many hours as a full-time employee
  - To call the superclass implementation of an overridden member, prefix the method call with **Superclass::**

```
// Employee.h
int Employee::getHoursWorkedPerWeek() {
    return 40;
}

// HeadTA.h
int HeadTA::getHoursWorkedPerWeek() {
    return Employee::getHoursWorkedPerWeek() / 2;
}
```

This implementation means if the Employee standard work hours are changed, the Head TA hours will change as well.

可调用父类里面的public函数但是不能访问，修改父类里面private members

Enforcing Subclass Behavior

- Sometimes, it may not make sense to implement a method in the superclass, but we may want to require all subclasses to have it.
  - **E.g.** all Employees should have a **work** method, but how should a generic Employee implement that?
- You can write a method like this by making it *purely virtual*.

```
class Employee {
    ...
    // every employee subclass must implement this method,
    // but it doesn't really make sense for Employee to.
    virtual void work() = 0;
};
```

Pure virtual base class

- **pure virtual base class**: One where every member function is declared as pure virtual. *(Also usually has no member variables.)*
  - Essentially not a superclass in terms of inheriting useful code.
  - But useful as a list of requirements for subclasses to implement.
  - Example: Demand that all shapes have an area, perimeter, # sides, ...

```
class Shape {    // pure virtual class; extend me!
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual int sides() const = 0;
};
```

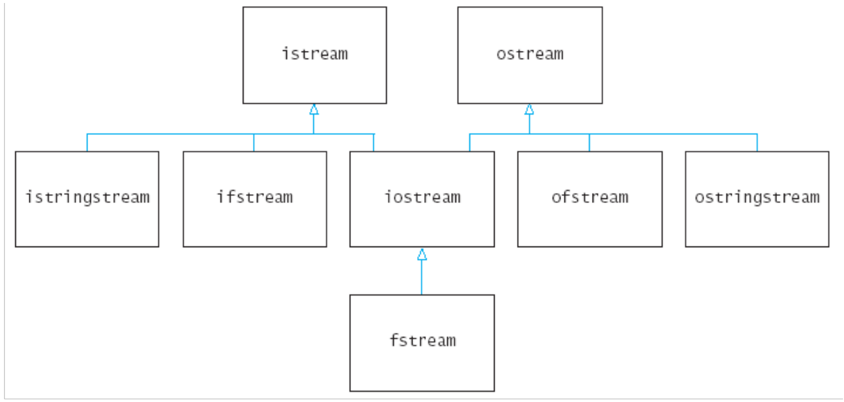
abstract class

  - FYI: In Java, this is called an *interface*.

Multiple inheritance

```
class Name : public Superclass1, public Superclass2, ...
```

- **multiple inheritance**: When one subclass has multiple superclasses.
  - Forbidden in many OO languages (e.g. Java) but allowed in C++.
  - Convenient because it allows code sharing from multiple sources.
  - Can be confusing or buggy, e.g. when both superclasses define a member with the same name.
  - Example: The C++ I/O streams use multiple inheritance:



Plan For Today

- Recap: Inheritance and Composition
- **Polymorphism**
- Announcements
- Sorting Algorithms

Polymorphism

- How can we store different types of objects together? E.g. what if we wanted to store Lawyer and HeadTA objects in the same Vector?

```
Lawyer *ken = new Lawyer(“Ken”, 10, “GWU”);
HeadTA *zach = new HeadTA(“Zach”, 1, “CS106X”);

Vector<?> all;
all.add(ken);
all.add(zach);
```

Polymorphism

- How can we store different types of objects together? E.g. what if we wanted to store Lawyer and HeadTA objects in the same Vector?

```
Lawyer *ken = new Lawyer("Ken", 10, "GWU");
HeadTA *zach = new HeadTA("Zach", 1, "CS106X");

Vector<Employee *> all;
all.add(ken);
all.add(zach);

// A pointer to a Lawyer or Head TA is by
// definition a pointer to an Employee!
```

Polymorphism

- How can we store different types of objects together? E.g. what if we wanted to store Lawyer and HeadTA objects in the same Vector?

```
Lawyer ken("Ken", 10, "GWU");
HeadTA zach("Zach", 1, "CS106X");

Vector<Employee> all;
all.add(ken);
all.add(zach);

// Direct casting causes issues in C++ because
// all these variables live on the stack.
```

Polymorphism

- Now we have one collection for these different types! But can we still call methods on them that utilize their unique behavior?

```
Lawyer *ken = new Lawyer(“Ken”, 10, “GWU”);
HeadTA *zach = new HeadTA(“Zach”, 1, “CS106X”);

Vector<Employee *> all = { ken, zach };

cout << all[0]->getHoursWorkedPerWeek() << endl;
cout << all[1]->getHoursWorkedPerWeek() << endl;
```

Polymorphism

Polymorphism is the the ability for the same code to be used with different types of objects and behave differently with each.

```
Lawyer *ken = new Lawyer(“Ken”, 10, “GWU”);
HeadTA *zach = new HeadTA(“Zach”, 1, “CS106X”);

Vector<Employee *> all = { ken, zach };

cout << all[0]->getHoursWorkedPerWeek() << endl;     // 40
cout << all[1]->getHoursWorkedPerWeek() << endl;     // 20
```

Polymorphism

For example, even if you have a pointer to a superclass, if you call a method that a subclass overrides, it will call the **subclass's implementation**.

```
Lawyer *ken = new Lawyer("Ken", 10, "GWU");
HeadTA *zach = new HeadTA("Zach", 1, "CS106X");

Vector<Employee *> all = { ken, zach };

cout << all[0]->getHoursWorkedPerWeek() << endl;     // 40
cout << all[1]->getHoursWorkedPerWeek() << endl;     // 20
```

Why Is This Important?

Polymorphism is important because for instance by default, with a vector of the same type of object, you might expect that calling a method on all of them would execute the exact same code.

Polymorphism means that is not true!

```
cout << all[0]->getHoursWorkedPerWeek() << endl;     // 40
cout << all[1]->getHoursWorkedPerWeek() << endl;     // 20
```

Templates

- With **templates**, we create one class that works with any type parameter:

```
template<typename T>
class Vector {
    …
}
```

- This is *also* polymorphism; C++ knows to execute different code for `Vector<int>` vs. `Vector<string>`, even though they are all Vectors.
- At compile-time, C++ generates a version of this class for each type it will be used with. This is called **compile-time** polymorphism.

Inheritance

- With **inheritance**, we create multiple classes that inherit and override behavior from each other.

```
class Employee { ... }
class Head TA : public Employee { ... }
class Lawyer : public Employee { ... }
```

- **Problem**: can C++ know which version of a method to call at compile time?

Inheritance

```
Employee *createEmployee() {
    string type = getLine("Employee type: ");
    if (type == "Head TA") {
        ...
        return new HeadTA(...);
    } else if (type == "Lawyer") {
        ...
        return new Lawyer(...);
    } else {...}
}
// It's impossible for the compiler to know until
// the program runs what type will be returned!
```

Inheritance

- With **inheritance**, we create multiple classes that inherit and override behavior from each other.

```
class Employee { ... }
class Head TA : public Employee { ... }
class Lawyer : public Employee { ... }
```

- **Problem**: C++ can't always figure out until runtime which version of a method to use!
- C++ instead figures it out at **runtime** using a *virtual table* of methods. This is called **run-time** polymorphism.

Casting

- When you store a subclass in a superclass pointer, you cannot utilize any additional behavior from the subclass.

```
Employee *zach = new HeadTA("Zach", 1, "CS106X");
cout << zach->getFavoriteProgrammingLanguage() << endl;   // compile error!
```

- If you would like to use this behavior, you must **cast**:

```
Employee *zach = new HeadTA("Zach", 1, "CS106X");
cout << ((HeadTA *)zach)->getFavoriteProgrammingLanguage() << endl;
```

- Be careful to not cast a variable to something it is not!

"Polymorphism mystery"

```
class Snow {
public:
    virtual void method2() {
        cout << "Snow 2" << endl;
    }
    virtual void method3() {
        cout << "Snow 3" << endl;
    }
};

class Rain : public Snow {
public:
    virtual void method1() {
        cout << "Rain 1" << endl;
    }
    virtual void method2() {
        cout << "Rain 2" << endl;
    }
};
```

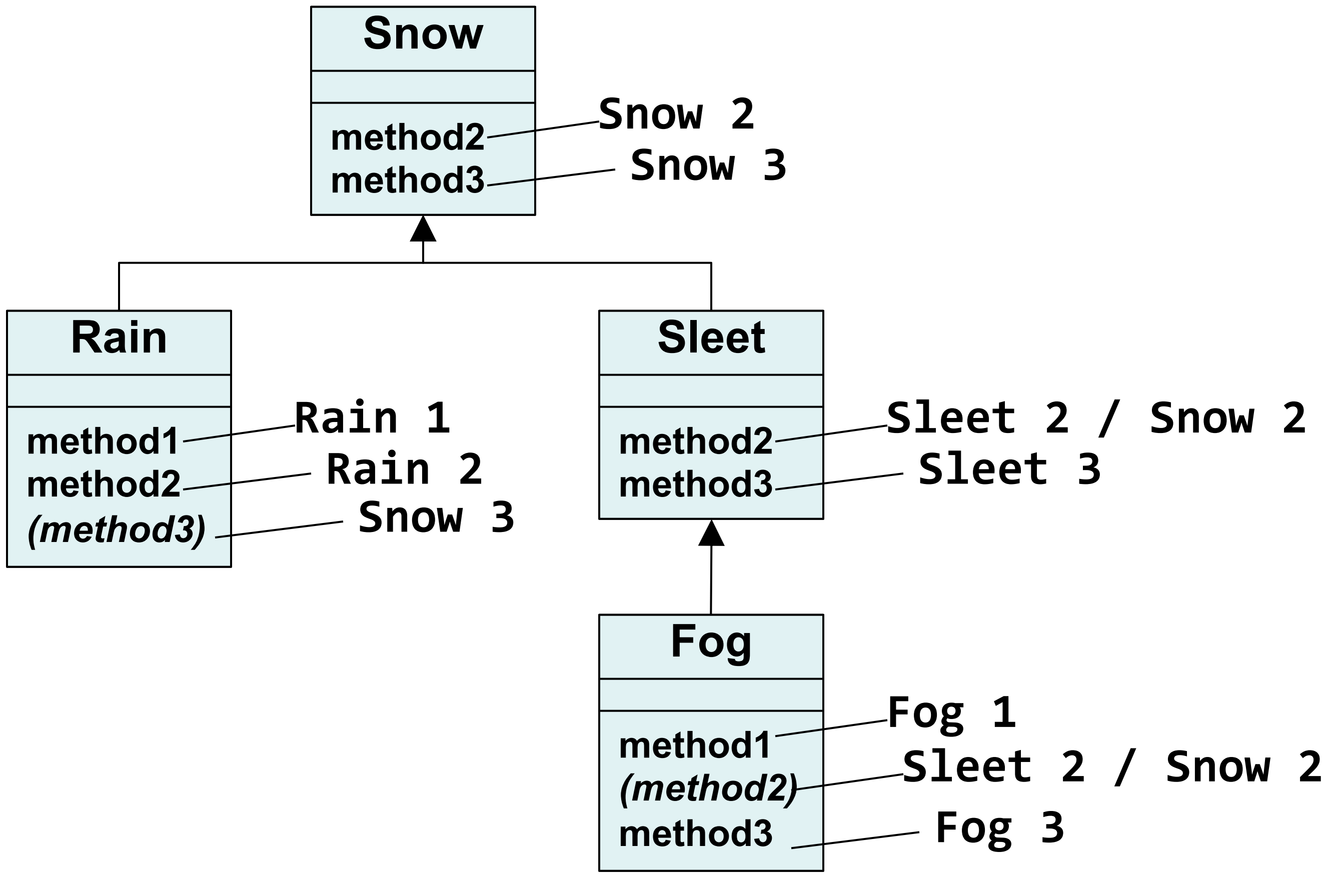
"Polymorphism mystery"

```
class Sleet : public Snow {
public:
    virtual void method2() {
        cout << "Sleet 2" << endl;
        Snow::method2();
    }
    virtual void method3() {
        cout << "Sleet 3" << endl;
    }
};

class Fog : public Sleet {
public:
    virtual void method1() {
        cout << "Fog 1" << endl;
    }
    virtual void method3() {
        cout << "Fog 3" << endl;
    }
};
```

Diagramming classes

- Draw a diagram of the classes from top (superclass) to bottom.



Mystery problem

```
Snow* var1 = new Sleet();
var1->method2();            // What's the output?
```

- To find the behavior/output of calls like the one above:
  1. Look at the variable's type.
     If that type does not have that member: COMPILER ERROR.

  2. Execute the member.
     Since the member *is* virtual: behave like the object's type,
     not like the variable's type.

先左后右：左边管compile（共性方法）；右边管具体执行用哪种方案

Example 1

- **Q:** What is the result of the following call?

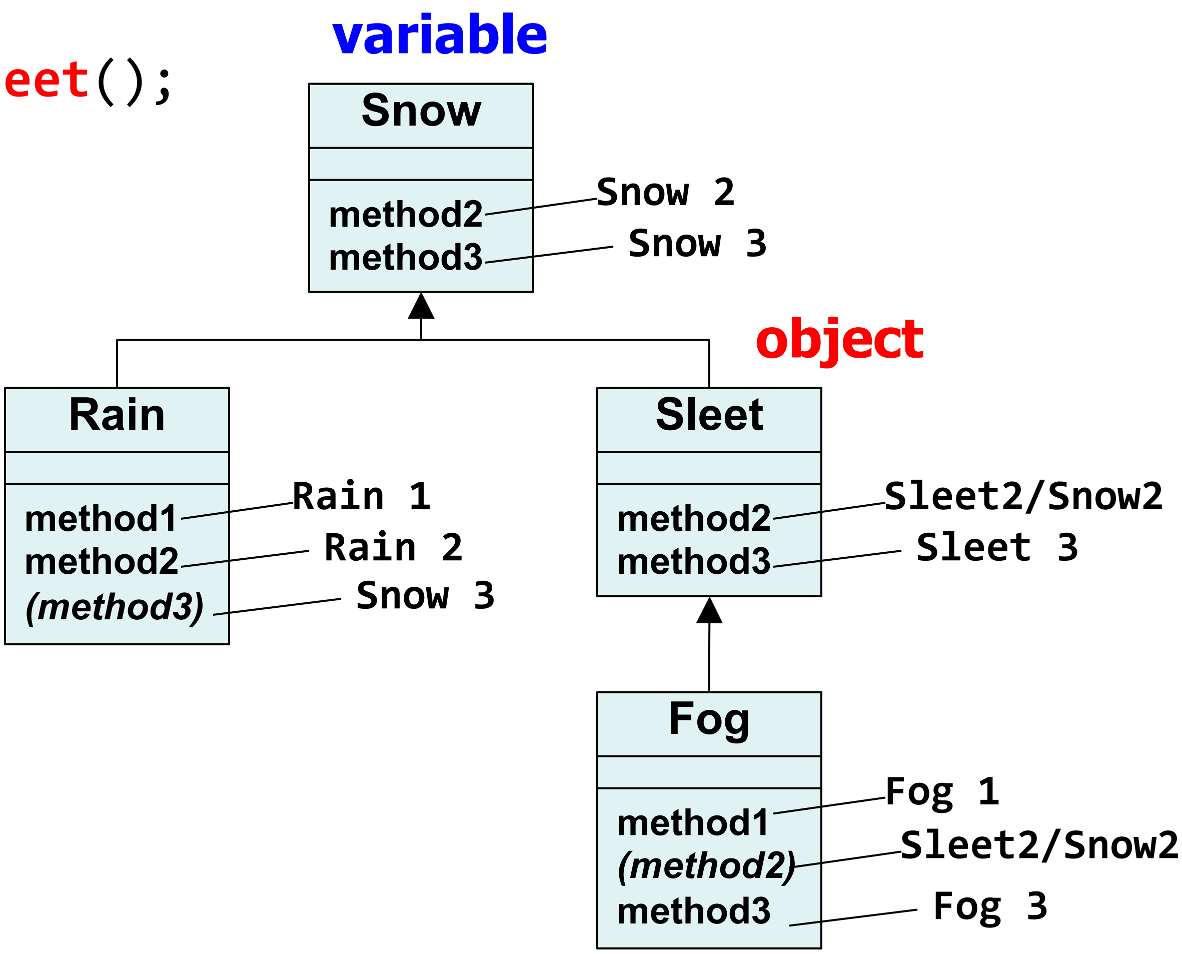
```
Snow* var1 = new Sleet();
var1->method2();
```

A. Snow 2

B. Rain 2

C. Sleet 2
Snow 2

D. COMPILER ERROR



Example 2

- **Q:** What is the result of the following call?

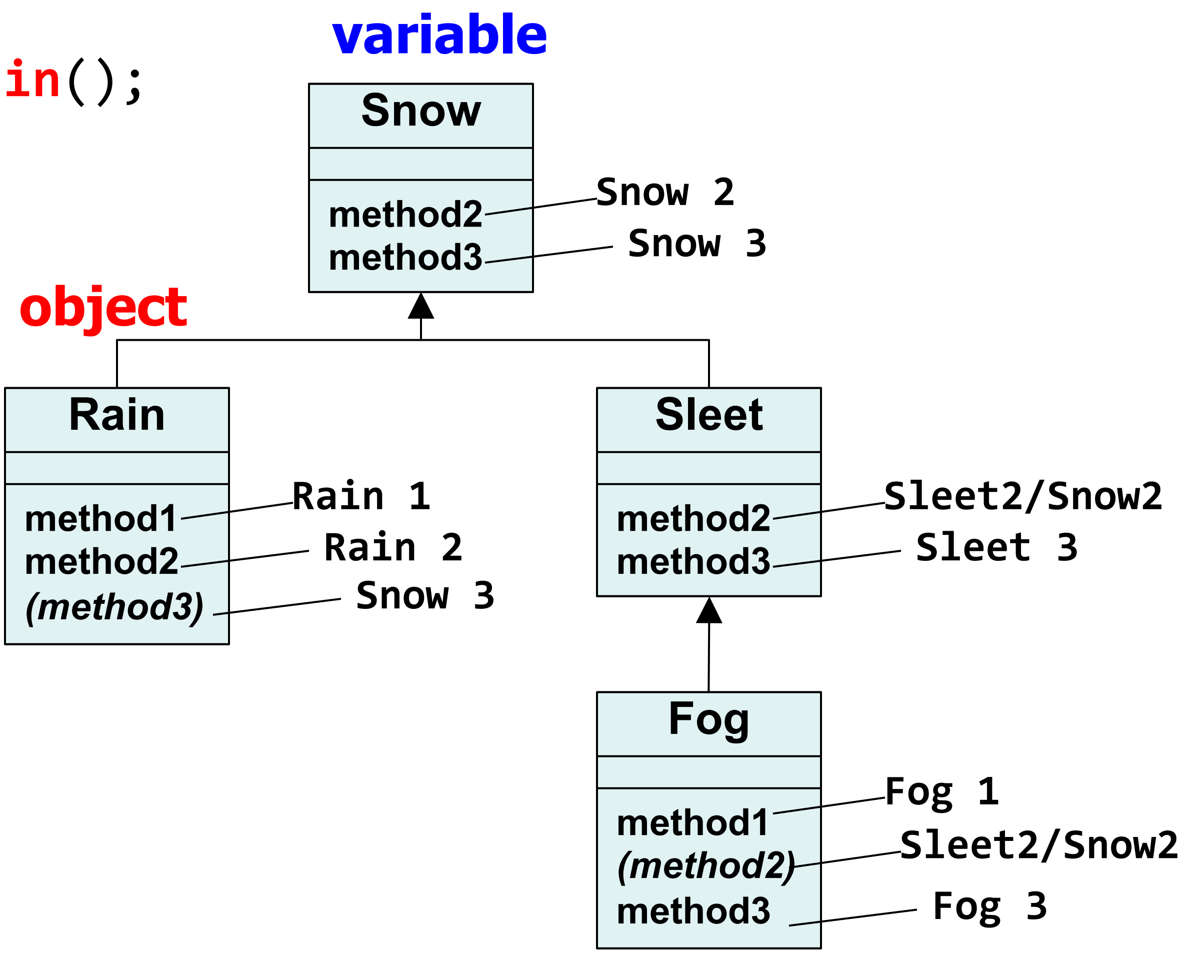
```
Snow* var2 = new Rain();
var2->method1();
```

A. Snow 1

B. Rain 1

C. Snow 1
   Rain 1

D. COMPILER ERROR



Example 3

- **Q:** What is the result of the following call?

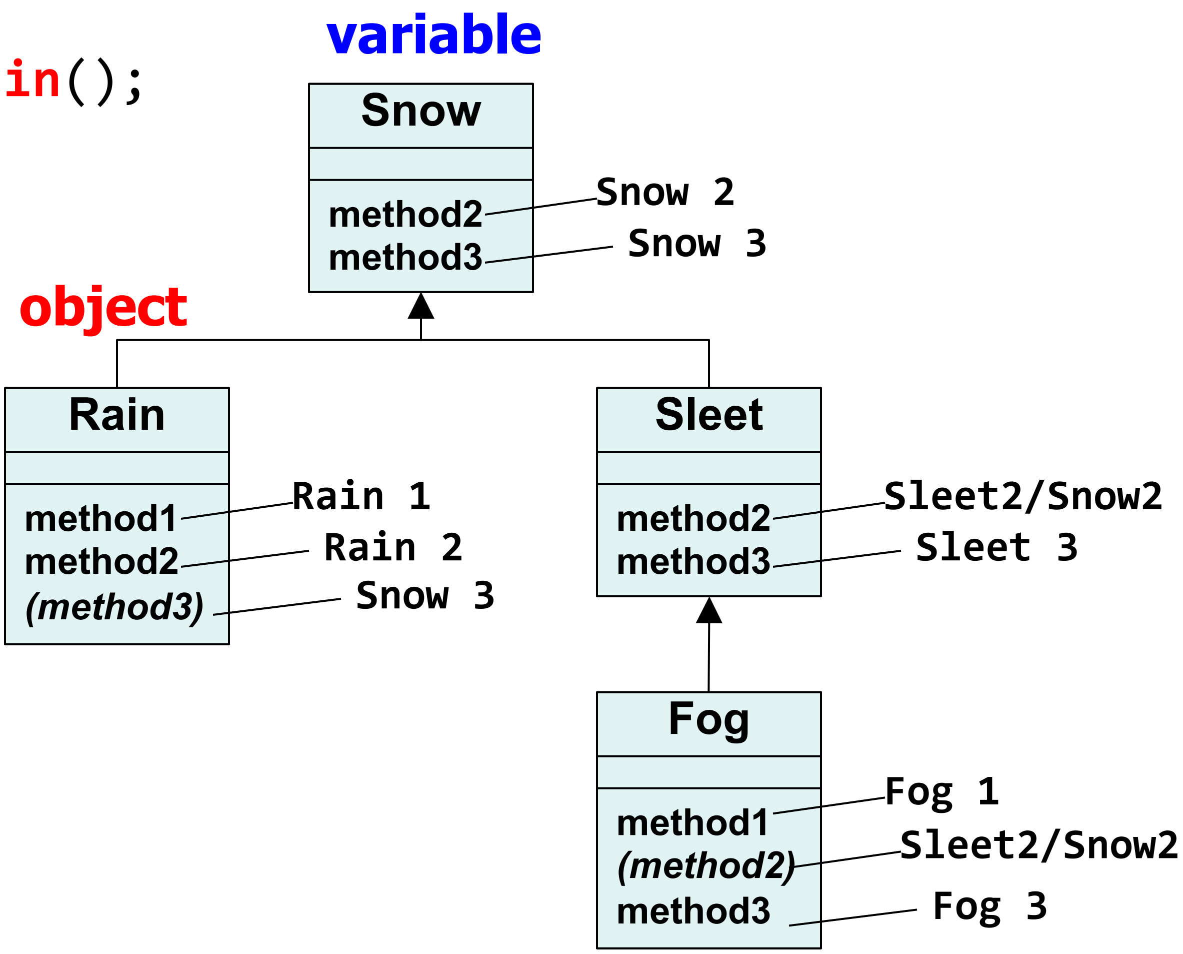
```
Snow* var3 = new Rain();
var3->method2();
```

A. Snow 2

B. Rain 2

C. Sleet 2
Snow 2

D. COMPILER ERROR



Mystery with type cast

```
Snow* var4 = new Rain();
((Sleet*) var4)->method1();    // What's the output?
```

- If the mystery problem has a type cast, then:

1. Look at the cast type.
   If that type does not have the method: COMPILER ERROR.
   (Note: If the object's type were not equal to or a subclass of the cast type, the code would CRASH / have unpredictable behavior.)

2. Execute the member.
   Since the member is virtual, behave like the object's type.

Example 4

- **Q:** What is the output of the following call?

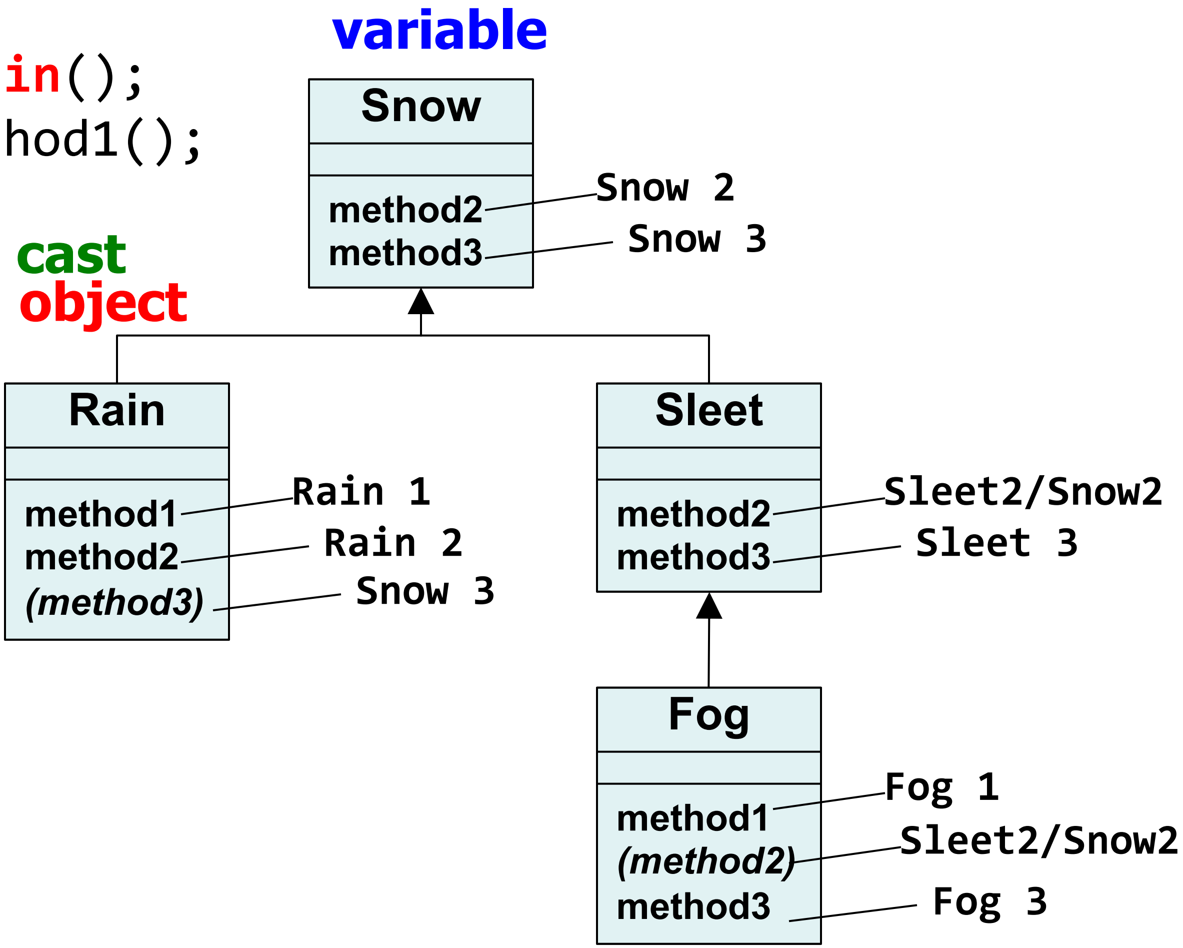
```
Snow* var4 = new Rain();
((Rain*) var4)->method1();
```

A. Snow 1

B. Rain 1

C. Sleet 1

D. COMPILER ERROR



Example 5

- **Q:** What is the output of the following call?

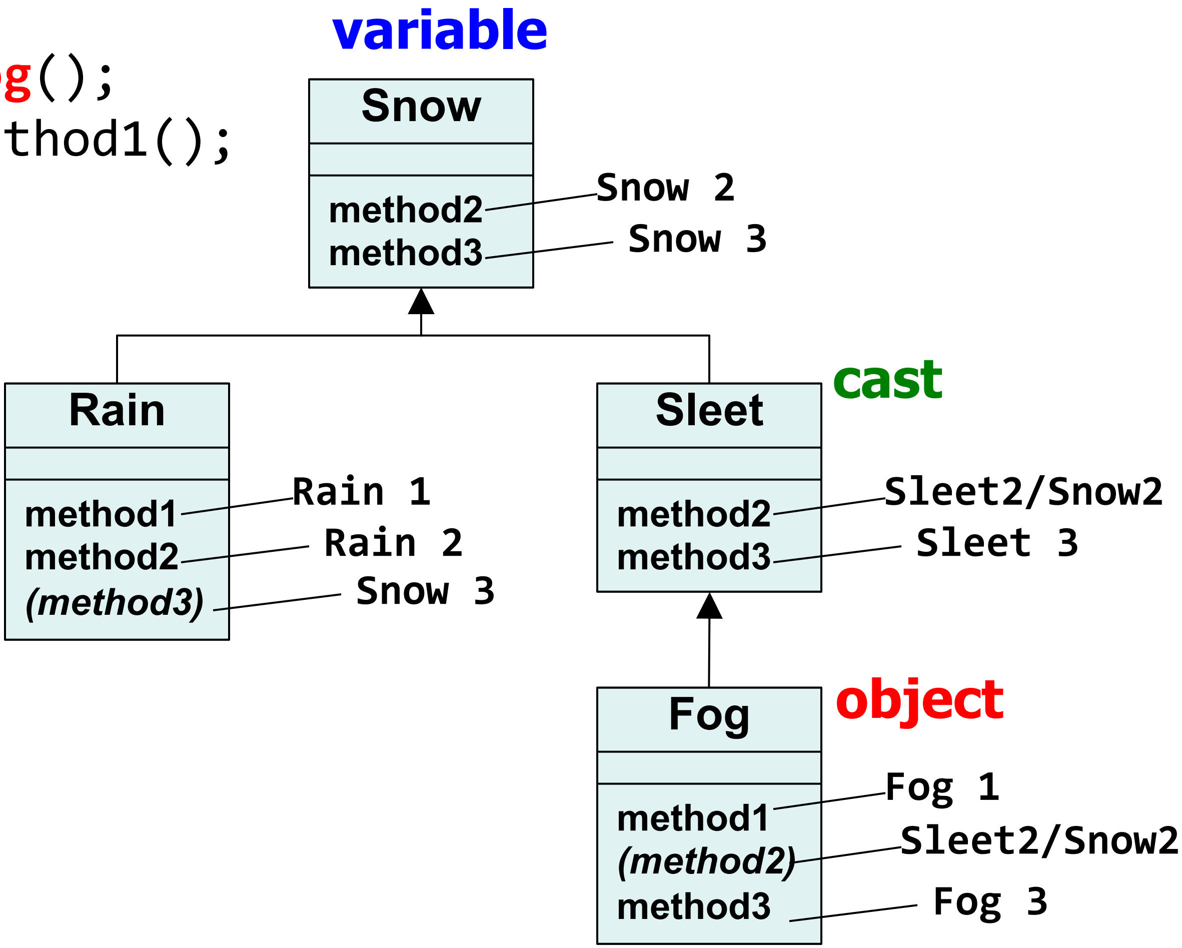
```
Snow* var5 = new Fog();
((Sleet*) var5)->method1();
```

A. Snow 1

B. Sleet 1

C. Fog 1

D. COMPILER ERROR



能不能转；有没有方法

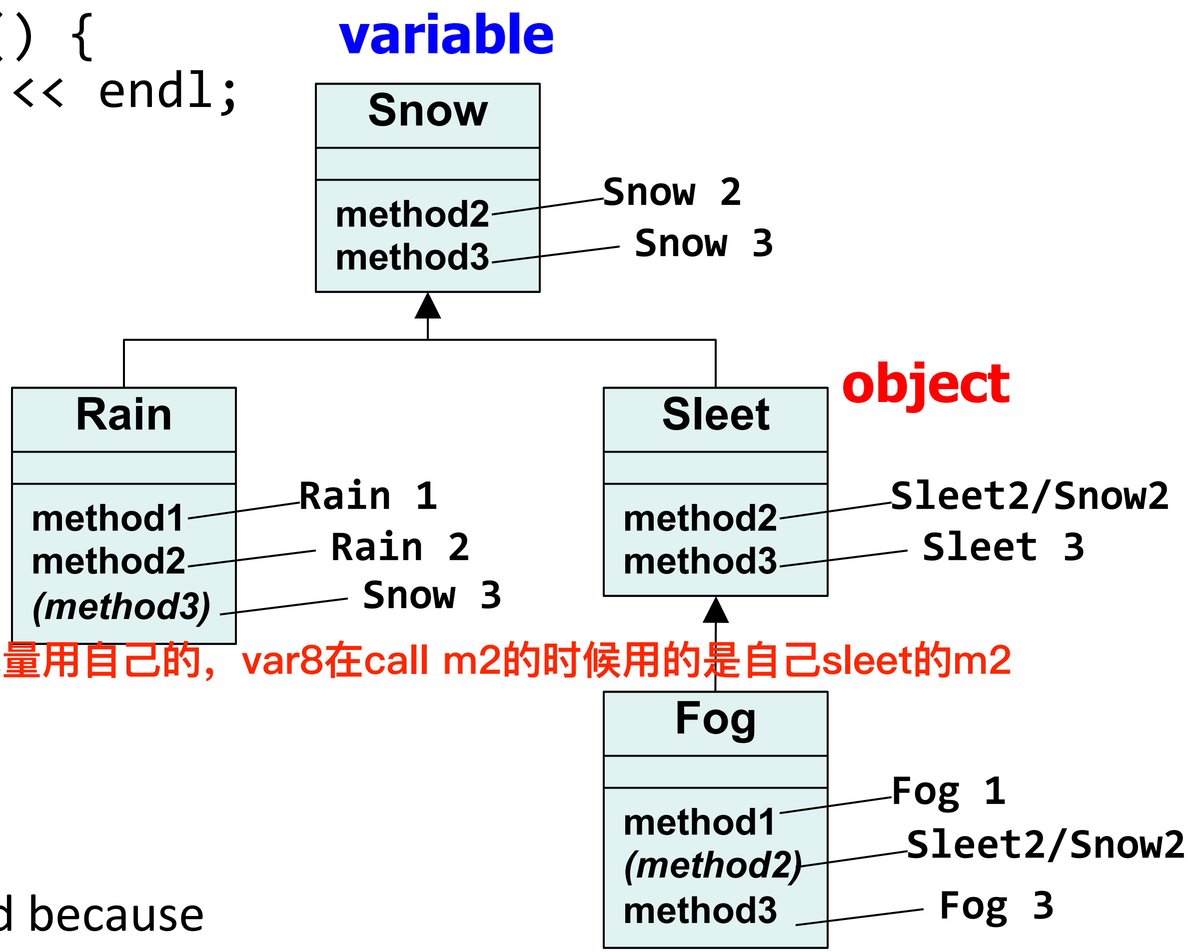
Example 6

- Suppose we add the following method to base class Snow:

```
virtual void method4() {
    cout << "Snow 4" << endl;
    method2();
}
```

- What is the output?

```
Snow* var8 =
  new Sleet();
var8->method4();
```



能用自己的尽量用自己的，var8在call m2的时候用的是自己sleet的m2

- Answer:

```
Snow 4
Sleet 2
Snow 2
```

(Sleet's method2 is used because method4 and method2 are virtual.)

Example 7

- **Q:** What is the output of the following call?

```
Snow* var6 = new Sleet();
((Rain*) var6)->method1();
```

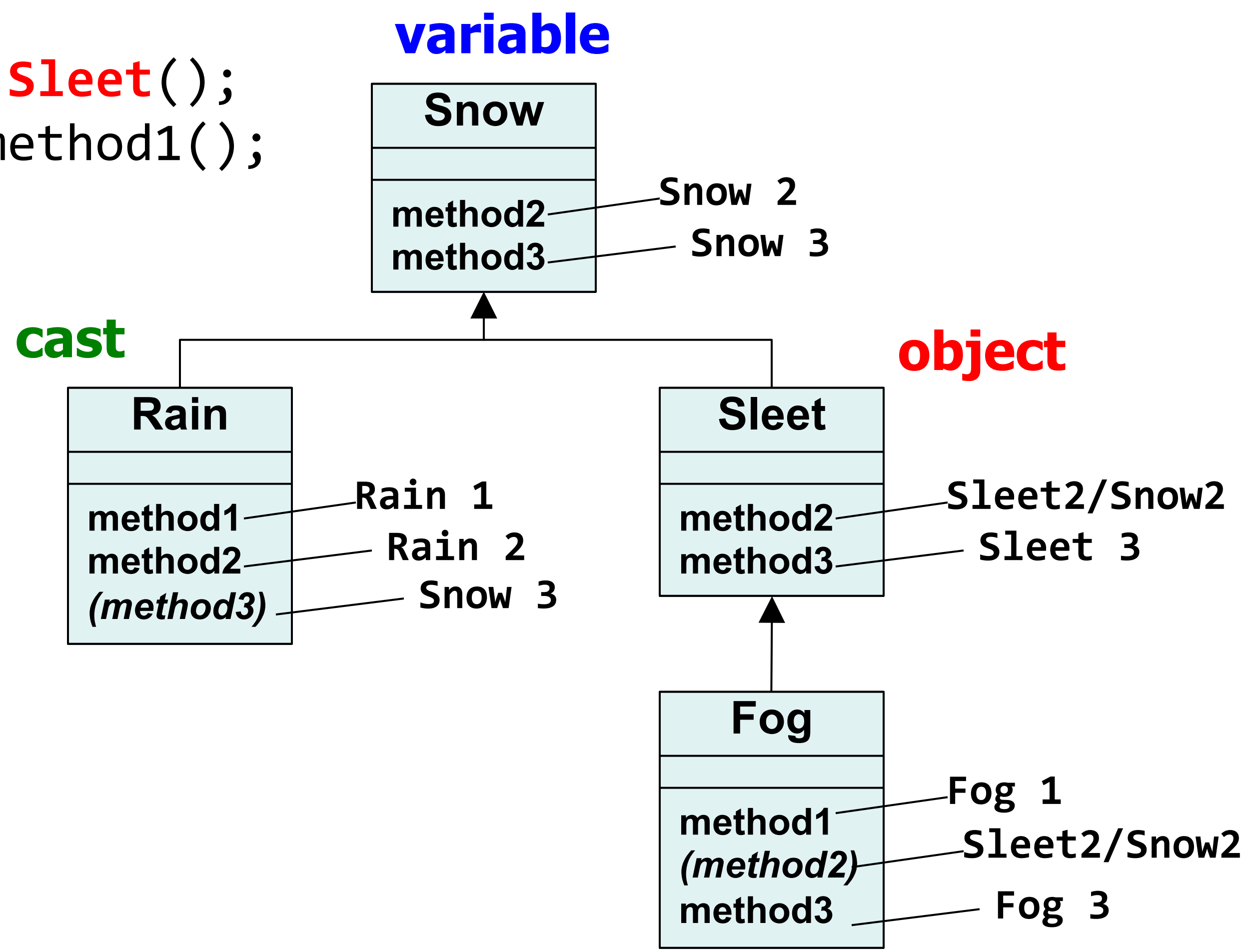
A. Snow 1

B. Sleet 1

C. Fog 1

D. COMPILER ERROR

E. CRASH



Plan For Today

- Recap: Inheritance and Composition
- Polymorphism
- **Announcements**
- Sorting Algorithms

Announcements

- **HW8** (106XCell) is out now, and is due **12/7 at 6PM**.
  - **No late submissions will be accepted**
- Final exam review session **Wed. 12/5 7-8:30PM**, location TBA
- Poster sessions for AI (CS221) and Generative Model (CS236) classes
  - CS221: Monday, 12/3 1-5PM in Tresidder Union Oak Lounge
  - CS236: Today, 11/30 12:30-4:30PM in Gates Building AT&T Patio
- Donald Knuth's *Christmas Lecture*
  - Dancing Links data structuring idea
  - Tuesday, 12/4 6:30-7:30PM in Huang Building, NVIDIA Auditorium

Plan For Today

- Recap: Inheritance and Composition
- Polymorphism
- Announcements
- **Sorting Algorithms**

Sorting

- In general, sorting consists of putting elements into a particular order, most often the order is numerical or lexicographical (i.e., alphabetic).
- Why study sorting?
  - Sorting algorithms can be designed in various ways with different tradeoffs
  - Sorting algorithms are a great application of algorithm design and analysis
- Cool visualizations: https://www.toptal.com/developers/sorting-algorithms

Sorting

- **bogo ("monkey") sort**: shuffle and hope
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the data in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively "partition" data based on a middle value
- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...

...

Sorting

- **bogo ("monkey") sort**: shuffle and hope
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the data in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively "partition" data based on a middle value
- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...

...

Selection sort example

- **selection sort**: Repeatedly swap smallest unplaced value to front.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	**-4**	18	12	**22**	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	**7**	22	27	30	36	50	**12**	68	91	56	18	85	42	98	25

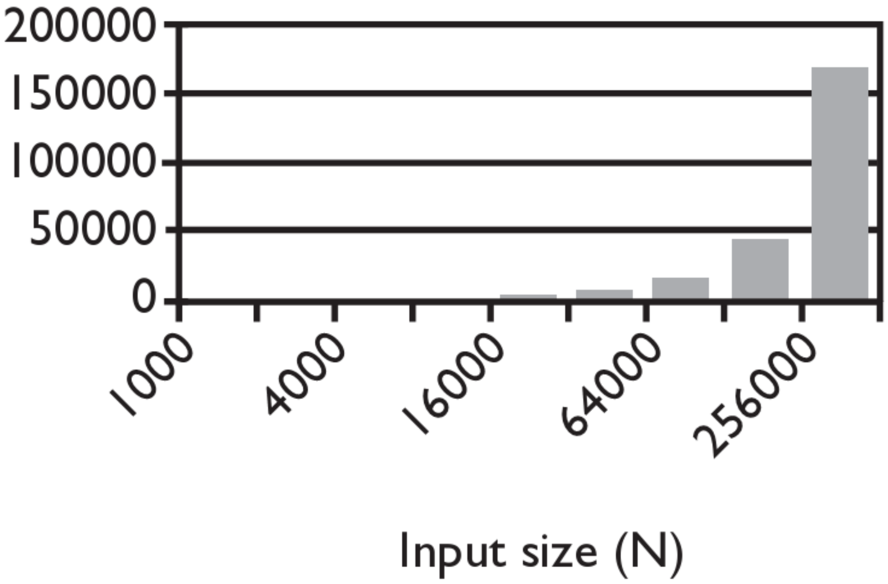
Selection sort code

```
// Rearranges elements of v into sorted order.
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size() - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[min]) {
                min = j;
            }
        }
        // swap smallest value to proper place, v[i]
        if (i != min) {
            int temp = v[i];
            v[i] = v[min];
            v[min] = temp;
        }
    }
}
```

Selection sort runtime

- What is the complexity class (Big-Oh) of selection sort?
  - **O(N^2). Best case still O(N^2).**

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert second item into sorted sublist, shifting first item if needed
  - insert third item into sorted sublist, shifting items 1-2 as needed
  - ...
  - repeat until all values have been inserted into their proper positions

- Runtime: **$O(N^2)$. But best case $O(N)$!**
  - Generally somewhat faster than selection sort for most inputs.

Insertion sort example

- Makes N-1 passes over the array.
- At the end of pass i, the elements that occupied $A[0]...A[i]$ originally are still in those spots and in sorted order.

index	0	1	2	3	4	5	6	7
value	15	2	8	1	17	10	12	5
pass 1	**2**	**15**	8	1	17	10	12	5
pass 2	**2**	**8**	**15**	1	17	10	12	5
pass 3	**1**	**2**	**8**	**15**	17	10	12	5
pass 4	**1**	**2**	**8**	**15**	**17**	10	12	5
pass 5	**1**	**2**	**8**	**10**	**15**	**17**	12	5
pass 6	**1**	**2**	**8**	**10**	**12**	**15**	**17**	5
pass 7	**1**	**2**	**5**	**8**	**10**	**12**	**15**	**17**

Insertion sort code

```
// Rearranges the elements of v into sorted order.
void insertionSort(Vector<int>& v) {
    for (int i = 1; i < v.size(); i++) {
        int temp = v[i];

        // slide elements right to make room for v[i]
        int j = i;
        while (j >= 1 && v[j - 1] > temp) {
            v[j] = v[j - 1];
            j--;
        }
        v[j] = temp;
    }
}
```

Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

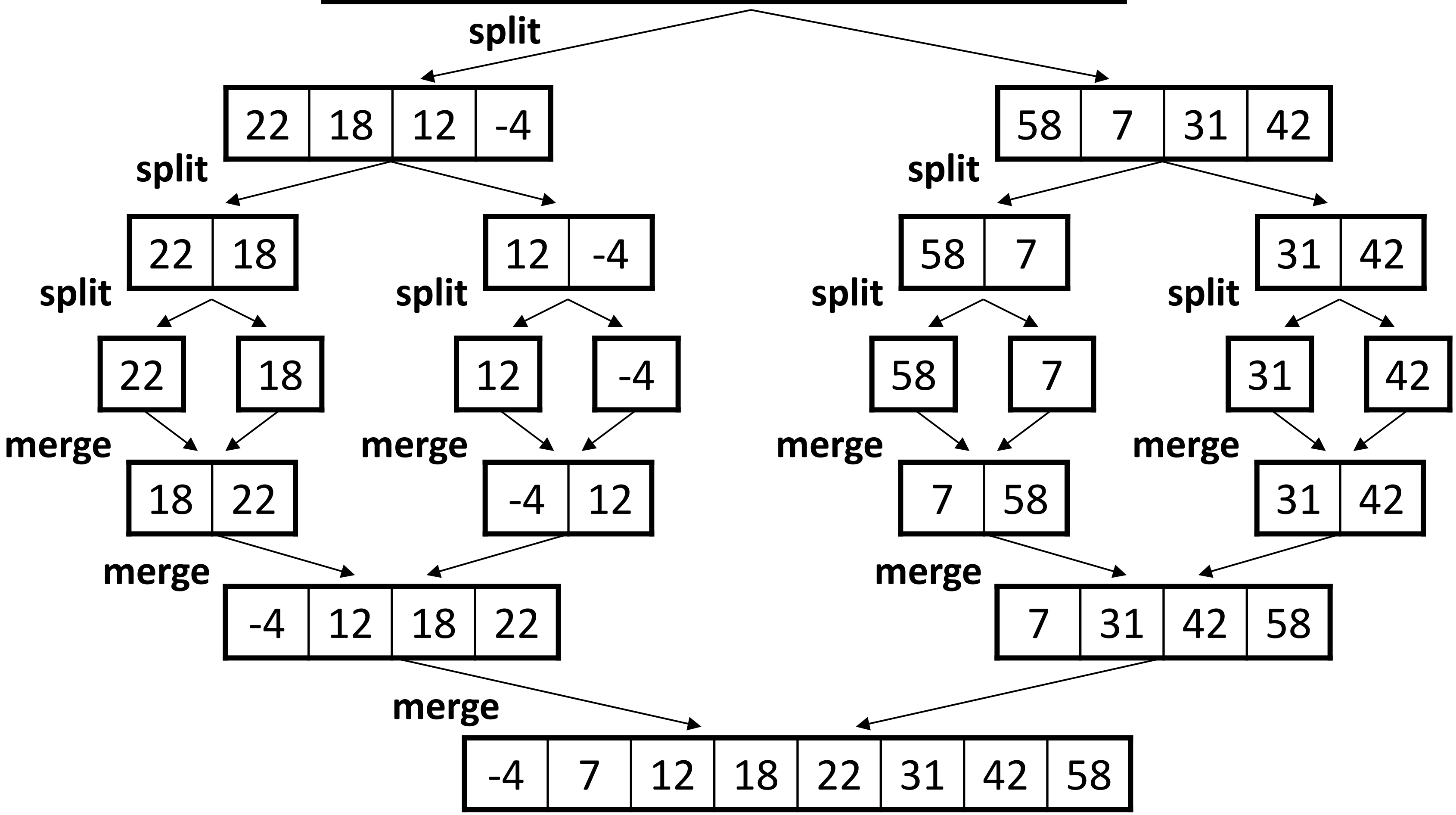
  The algorithm:
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.

  - Often implemented recursively.
  - An example of a "divide and conquer" algorithm.
    - Invented by John von Neumann in 1945

  - Runtime: **O(*N* log *N*)**. Somewhat faster for asc/descending input.

Merge sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Merging sorted halves

Subarrays								Next include	Merged array							
0	1	2	3	0	1	2	3		0	1	2	3	4	5	6	7
14	32	67	76	23	41	58	85	14 from left	14							
i1				i2					i							
14	32	67	76	23	41	58	85	23 from right	14	23						
	i1			i2						i						
14	32	67	76	23	41	58	85	32 from left	14	23	32					
	i1				i2						i					
14	32	67	76	23	41	58	85	41 from right	14	23	32	41				
		i1			i2							i				
14	32	67	76	23	41	58	85	58 from right	14	23	32	41	58			
		i1				i2							i			
14	32	67	76	23	41	58	85	67 from left	14	23	32	41	58	67		
		i1					i2							i		
14	32	67	76	23	41	58	85	76 from left	14	23	32	41	58	67	76	
			i1				i2								i	
14	32	67	76	23	41	58	85	85 from right	14	23	32	41	58	67	76	85
							i2									i

Merge sort code

```
// Rearranges the elements of v into sorted order using
// the merge sort algorithm.
void mergeSort(Vector<int>& v) {
    if (v.size() >= 2) {
        // split vector into two halves
        Vector<int> left;
        for (int i = 0; i < v.size()/2; i++) {left += v[i];}
        Vector<int> right;
        for (int i = v.size()/2; i < v.size(); i++) {right += v[i];}

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        v.clear();
        merge(v, left, right);
    }
}
```

Merge halves code

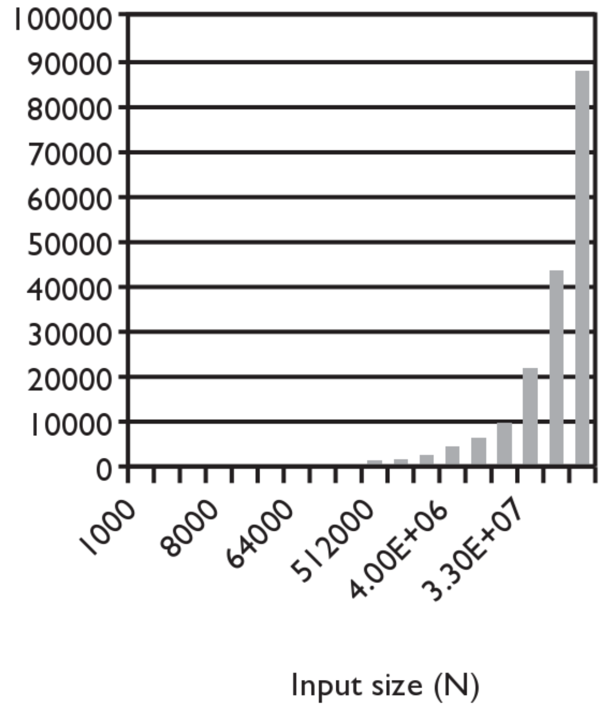
```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
void merge(Vector<int>& result,
           Vector<int>& left, Vector<int>& right) {
    int i1 = 0;   // index into left side
    int i2 = 0;   // index into right side

    for (int i = 0; i < left.size() + right.size(); i++) {
        if (i2 >= right.size() ||
           (i1 < left.size() && left[i1] <= right[i2])) {
            result += left[i1];    // take from left
            i1++;
        } else {
            result += right[i2];   // take from right
            i2++;
        }
    }
}
```

Merge sort runtime

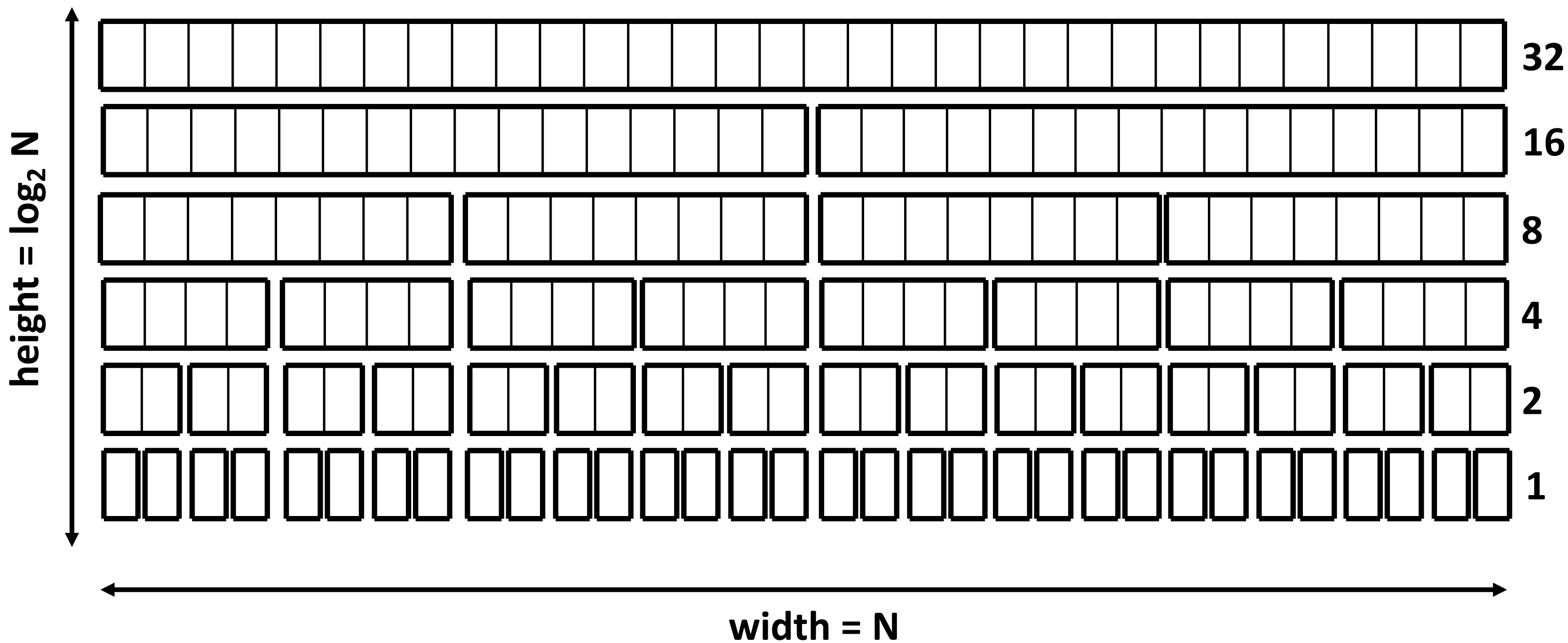
- What is the complexity class (Big-Oh) of merge sort?
  - **O(*N* log *N*).**

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



More runtime intuition

- Merge sort performs $O(N)$ operations on each level. *(width)*
  - Each level splits the data in 2, so there are $\log_2 N$ levels. *(height)*
  - Product of these = $N * \log_2 N = O(N \log N)$. *(area)*
  - Example: $N = 32$. Performs ~ $\log_2 32 = 5$ levels of N operations each:



Quick sort

- **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960

- Quick sort is another divide and conquer algorithm:
  - Choose one element in the list to be the pivot.
  - *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - *Conquer* by applying quick sort (recursively) to both partitions.

- Runtime: **O(*N* log *N*)** average, but O(N^2) worst case.
  - Generally somewhat faster than merge sort.

Choosing a "pivot"

- The algorithm will work correctly no matter which element you choose as the pivot.
  - A simple implementation can just use the first element.

- But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  - What kind of value would be a good pivot? A bad one?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

Partitioning an array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until *i*,*j* meet):
  - Starting from *i* = 0, find an element a[*i*] ≥ pivot.
  - Starting from *j* = *N*-1, find an element a[*j*] ≤ pivot.
  - These elements are out of order, so swap a[*i*] and a[*j*].
- Swap the pivot back to index *i* to place it between the partitions.

index	0	1	2	3	4	5	6	7	8	9
value	6	1	4	9	0	3	5	2	7	8

8 i							←	j	6
2	i	→	→			j	8		
			5	i	→	9			
						6			9
2	1	4	5	0	3	6	8	7	9

Quick sort example

index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65
	32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
	32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
	32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
	32	23	16	43	57	39	92	81	75	65	
	32	23	16	43	57	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	43	57	39	pivot=32
39	23	16	43	57	**32**	swap to end
16	23	**39**	43	57	32	swap 39, 16
16	23	**32**	43	57	**39**	swap 32 back in

...

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

Quick sort code

```
void quickSort(Vector<int>& v) {
    quickSortHelper(v, 0, v.size() - 1);
}

void quickSortHelper(Vector<int>& v, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = v[min];
    swap(v, min, max);      // move pivot to end

    // partition the two sides of the array
    int middle = partition(v, min, max - 1, pivot);

    swap(v, middle, max);   // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSortHelper(v, min, middle - 1);
    quickSortHelper(v, middle + 1, max);
}
```

Partition code

```
// Partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
int partition(Vector<int>& v, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center
        // until we find a pair of out-of-order elements
        while (i <= j && v[i] < pivot) { i++; }
        while (i <= j && v[j] > pivot) { j--; }

        if (i <= j) {
            swap(v, i++, j--);
        }
    }
    return i;
}

// Moves the value at index i to index j, and vice versa.
void swap(Vector<int>& v, int i, int j) {
    int temp = v[i];  v[i] = v[j];  v[j] = temp;
}
```

Choosing a better pivot

- Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  - does not partition the array into roughly-equal size chunks

- Alternative methods of picking a pivot:
  - *random:* Pick a random index from [*min* .. *max*]
  - *median-of-3:* look at left/middle/right elements and pick the one with the medium value of the three:
    - `v[min]`, `v[(max+min)/2]`, and `v[max]`
    - better performance than picking random numbers every time
    - provides near-optimal runtime for almost all input orderings

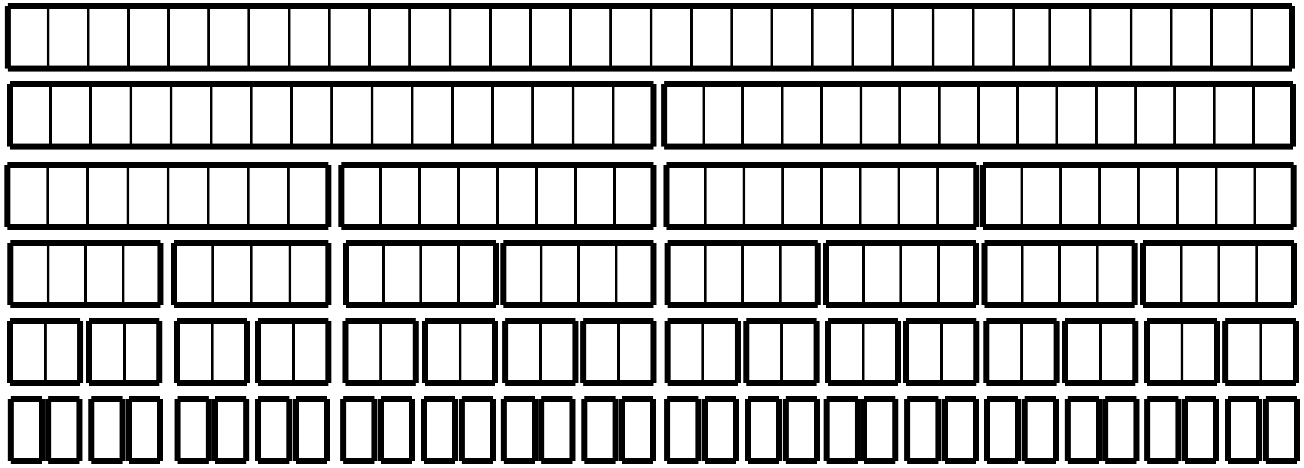
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**

Sorting

Sorting Big-O Cheat Sheet			
Sort	**Worst Case**	**Best Case**	**Average Case**
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Parallel sorts

- **parallel sorting algorithms**: modify existing algos. to work with multiple CPUs/cores
- common example: **parallel merge sort**.
  - general algorithm idea:
    - Split array into two halves.
    - One core/CPU sorts each half.
    - Once both halves are done, a single core merges them.



Recap

- **Recap:** Inheritance
- Polymorphism
- Announcements
- Sorting Algorithms

- **Learning Goal 1:** understand how to create and use classes that build on each other's functionality.
- **Learning Goal 2:** understand different ways to sort data, and how to analyze and understand their implementations and tradeoffs.

- **Next time:** Hashing