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**.
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:

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

```cpp
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.

```cpp
// 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

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

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

This implementation means we must change 2 files if an employee's 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::`

```cpp
// 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.
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*.

```cpp
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**: 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, ...

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

– 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 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
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!

```cpp
cout << all[0]->getHoursWorkedPerWeek() << endl;  // 40
cout << all[1]->getHoursWorkedPerWeek() << endl;  // 20
```
• With **templates**, we create one class that works with any type parameter:

```cpp
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.
  
  ```cpp
  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?
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.

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

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

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

- Be careful to not cast a variable to something it is not!
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;
    }
};
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;
    }
};
• 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.
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?

```java
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?

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

A. Snow 2  
B. Rain 2  
C. Sleet 2  
D. COMPILER ERROR
Mystery with type cast

```java
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?

\[
\text{Snow}^* \ \text{var4} = \text{new Rain}() ; \\
((\text{Rain}^*) \ \text{var4}) \rightarrow \text{method1}() ;
\]

A. Snow 1  
B. Rain 1  
C. Sleet 1  
D. COMPILER ERROR
Example 5

• Q: What is the output of the following call?

\[
\text{Snow}^* \text{ var5} = \text{new Fog}();
((\text{Sleet}^*) \text{ var5})\rightarrow\text{method1}();
\]

A. Snow 1

B. Sleet 1

C. Fog 1

D. COMPILER ERROR
Example 6

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

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

• What is the output?

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

• 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?

```cpp
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
• 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
• **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 ...
• **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:**

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>18</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>12</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>
// 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 $O(N^2)$.

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>47</td>
</tr>
<tr>
<td>8000</td>
<td>234</td>
</tr>
<tr>
<td>16000</td>
<td>657</td>
</tr>
<tr>
<td>32000</td>
<td>2562</td>
</tr>
<tr>
<td>64000</td>
<td>10265</td>
</tr>
<tr>
<td>128000</td>
<td>41141</td>
</tr>
<tr>
<td>256000</td>
<td>164985</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>15</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 1</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 2</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 3</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 5</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>17</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>pass 6</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>pass 7</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>
// 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**: 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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>

The diagram illustrates the merge sort process with a binary tree. The values are split and merged until they are sorted.
# Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14 from left</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>23 from right</td>
<td>14 23</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>32 from left</td>
<td>14 23 32</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>41 from right</td>
<td>14 23 32 41</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>58 from right</td>
<td>14 23 32 41 58</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>67 from left</td>
<td>14 23 32 41 58 67</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>76 from left</td>
<td>14 23 32 41 58 67 76</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>85 from right</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td></td>
<td>i</td>
</tr>
</tbody>
</table>
// 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);
  }
}
// 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?
  – $O(N \log N)$.

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

Input size (N)
More runtime intuition

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

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- width = $N$
- height = $\log_2 N$
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?

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>8</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>
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 \( a[i] \geq \text{pivot} \).
  - Starting from \( j = N-1 \), find an element \( a[j] \leq \text{pivot} \).
  - 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.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccccc}
8 & i & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & 6 \\
2 & i & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & j & 8 \\
5 & i & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & j & 9 \\
6 & 9 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & 6 & 9 \\
\end{array}
\]
### Quick sort example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>65</td>
<td>23</td>
<td>81</td>
<td>43</td>
<td>92</td>
<td>39</td>
<td>57</td>
<td>16</td>
<td>75</td>
<td>32</td>
</tr>
</tbody>
</table>

- **choose pivot=65**
- **swap pivot (65) to end**
- **swap 81, 16**
- **swap 57, 92**
- **swap pivot back in**

---

**recursively quicksort each half**

```
<table>
<thead>
<tr>
<th>32</th>
<th>23</th>
<th>16</th>
<th>43</th>
<th>57</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>23</td>
<td>16</td>
<td>43</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>32</td>
<td>23</td>
<td>16</td>
<td>43</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>32</td>
<td>23</td>
<td>16</td>
<td>43</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>32</td>
<td>23</td>
<td>16</td>
<td>43</td>
<td>57</td>
<td>39</td>
</tr>
</tbody>
</table>
```

- **pivot=32**
- **swap to end**
- **swap 39, 16**
- **swap 32 back in**

```
<table>
<thead>
<tr>
<th>81</th>
<th>75</th>
<th>92</th>
</tr>
</thead>
<tbody>
<tr>
<td>81</td>
<td>75</td>
<td>92</td>
</tr>
<tr>
<td>81</td>
<td>75</td>
<td>92</td>
</tr>
<tr>
<td>81</td>
<td>75</td>
<td>92</td>
</tr>
<tr>
<td>81</td>
<td>75</td>
<td>92</td>
</tr>
</tbody>
</table>
```

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

---
Quick sort code

```c
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);
}
```
// 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

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
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