

CPE 212 REVIEW GUIDE

JOSHUA BAYS

UNIVERSITY OF ALABAMA IN HUNTSVILLE

FALL 2024

ABOUT THIS DOCUMENT

- This document is intended to serve as a useful reference and study tool for the CPE 212 final exam.
- The code snippets presented in this document are designed to be basic representations of different concepts, and may differ from the methods presented in lectures.
- Any suggestions for additions or changes can be made to Joshua Bays by emailing jb0401@uah.edu or by sending a message via CanvasW
- This document and all code are under the GPLv3 license, copying, sharing, modifying copies, and sharing modified copies are permitted.

TABLE OF CONTENTS

1	Classes	9	Reusable Code
2	Pointers	10	Recursion
3	Exception Handling	11	Trees
4	Software Engineering	12	Heaps
5	Testing	13	Graphs
6	Stacks	14	Searching
7	Lists	15	Sorting
8	Queues	16	STL

CLASSES

- Class: Custom model of abstract data type (ADT)
- Object: Instance of a class
- Member types
 - ▶ Public: Can be accessed directly from outside of the class
 - ▶ Private: Can be accessed directly only from within the class
 - ▶ Protected: Can be inherited within derived classes
- Member functions
 - ▶ Constructor: Initialize an object
 - ▶ Transformers: Change an object's state
 - ▶ Observers: Get (but not change) an object's state
 - ▶ Iterators: Process all components within an ADT
 - ▶ Destructor: Properly clean up an object (Ex: de-allocate memory)
- Friend function: Function that can access private class members, but is not a member function (used outside of the class)

- Inheritance: Reuse existing class code for another class
- Multiple inheritance: Inheriting code from multiple classes
- Parent/Base class: The class being inherited from
- Derived class: A class that inherits from another class
- Virtual function: A member function that can be redefined by an inherited class
- Friend function: Non-member function that can access private members

Inheritance

```
class baseClass{
    public:
        baseClass();
        virtual int getX();
        int getY();
    private:
    protected:
        int x, y;
};

class derivedClass : public baseClass{
    /* Using the public keyword allows all public members to be inherited */
    public:
        derivedClass();
        int getX();
    private:
    protected:
};

baseClass::baseClass(){ x = 2; y = 2; }
int baseClass::getX(){ printf("Getting x\n"); return x; }
int baseClass::getY(){ return y; }

/* Redefine the member functions */
derivedClass::derivedClass(){ x = 3; y = 3; }
int derivedClass::getX(){ baseClass::getX(); return -1*x; }
```

Inheritance (cont.)

```
int main(){
    baseClass b;
    derivedClass d;
    printf("%i %i\n", b.getX(), b.getY());
    printf("%i %i\n", d.getX(), d.getY());
    return 0;
}
```

Output

Getting x

2 2

Getting x

-3 3

POINTERS

- Pointer: Variable that stores the memory address of another variable
- Dereferencing: Access the value stored in the location stored in the pointer
- Static allocation: Memory allocated at compile time
- Dynamic allocation: Memory allocated during program runtime
- Heap: Free memory for dynamic allocation

- Memory leak: Memory dynamically allocated but not deallocated
- Garbage: Locations in memory that can not be accessed any more
- Inaccessible object: Dynamically allocated variable without a pointer
- Dangling pointer: A pointer that points to memory that has been deallocated

- Use the new keyword to allocate memory in C++
- Use the delete keyword on a pointer to deallocate memory in C++

Static allocation

```
int main(){
    int x = 0;
    int *xPtr = &x; /* Create a pointer for x */
    printf("x is %i and stored at %X\n", x, xPtr);
    *xPtr = 2; /* Dereference the pointer to change x */
    printf("x is %i and stored at %X\n", x, xPtr);
    return 0;
}
```

Output

x is 0 and stored at 9B7CEBAC

x is 2 and stored at 9B7CEBAC

Dynamic Allocation

```
int main(){
    int* ptr;
    ptr = new int;
    *ptr = 10;
    printf("%i is stored at %X\n", *ptr, ptr);
    delete ptr; /* Deallocate the memory */
    ptr = NULL; /* Set the pointer to NULL because it does not point to any memory */
    return 0;
}
```

Output

10 is stored at 1349DEB0

EXCEPTION HANDLING

- Robustness: How well a program can recover from an error
- Error types
 - ▶ Unexpected user input
 - ▶ Hardware issues
 - ▶ Software issues
- Ways to handle errors
 - ▶ Print an error message
 - ▶ Return an unusual value (Ex: -1)
 - ▶ Use a status variable as an error flag
 - ▶ Use assertions to prevent further code execution
 - ▶ Exception handling

- Exception: Unexpected event that requires special processing
- Exception handler: Code designed to address a specific exception

EXCEPTION HANDLING - TRY/THROW/CATCH

- Try: Execute code that may cause an exception within its own block
- Throw: If an error is detected terminate the program or execute code to address the exception by “throwing” an error
- Catch: Address the exception based on the type of error provided by the throw statement

Basic exception handling

```
int main(){
    try{
        throw 2;
        printf("Print me\n");
    }
    catch(int x){ printf("Error of type int!\n"); return 1; }
    catch(...){ printf("Error of a different type!\n"); return 1; }
    return 0;
}
```

Output

Error of type int!

SOFTWARE ENGINEERING

- Attributes of good software
 - ▶ Works
 - ▶ Can be easily modified
 - ▶ Is reusable
 - ▶ Is completed within time and budget requirements
- Software engineering: The proper application of the principles of design, production, and maintenance of software
 - ▶ Technical challenges
 - ▶ Project management
- Defects in code
 - ▶ About 1 error is created for every 10 lines of code
 - ▶ 75% of a code's cost is in maintenance of that code
- Software process: The process by which software is developed and maintained

- Requirements: High-level description of the product
- Specification: Detailed description containing functional requirements and constraints
- Design: Architectural (high-level) and detailed (low-level) design of the product
- Implementation: Converting the design into code
- Testing/Verification: Finding and fixing errors and demonstrating that the product works correctly
- Postdelivery/Maintenance: Correct errors found by users and enhancing functionality

- Waterfall process: Each step of the process is an input for the next step
- Agile process: Emphasizing individuals/interactions and working software over specific processes in order to enable quick changing and customer collaboration
- Scrum: Work is designed to be done in short periods called "sprints," with daily work being determined by the needs of the current sprint

- Testing: Trying to discover errors within a program
- Debugging: Removing known errors from a program
- Driver: A program specifically designed to test a part of code
- Stub: Dummy code designed to simulate real-life use cases
- Assertion: A statement that is either true or false
- Precondition: An assertion that must be true in order for a postcondition to be returned
- Postcondition: An assertion that is expected from a certain precondition

- Deskchecking: Informal checking by the developer
- Unit testing: Formally testing individual parts of a program by themselves
- Integration testing: Formally and systematically testing a part of a program within the larger code base
- Acceptance testing: Testing the program with real data in its real environment
- Regression testing: Testing a program following modifications
- Black-box testing: Testing a program by its inputs and outputs
- Clear-box testing: Testing a program utilizing knowledge of its structure

- Verification: The program works properly
- Validation: The program satisfies the needs of the problem

- When creating interfaces, checking can exist in either in the interface implementation, or within client code (Varies by occasion)

TESTING

STACKS

- Stack: Specially organized list
 - ▶ LIFO structure (Last In, First Out)
 - ▶ Data entry is only through the top of the stack
- Basic stack operations:
 - ▶ Push: Add an item from the top of the stack
 - ▶ Pop: Remove the top item from the stack
 - ▶ Top: Observe the top item from the stack
 - ▶ IsEmpty: Returns if the stack has no elements on it
 - ▶ IsFull: Returns if the stack is at its maximum capacity
 - ▶ MakeEmpty: Remove all elements from the stack
- Different methods exist to implement stacks
 - ▶ Array-based: Less memory used, but harder to resize
 - ▶ Linked node-based: Easier to resize, but more memory used

Basic stack code (Linked list)

```
class StackFull{}; /* Error class */
class StackEmpty{}; /* Error class */

class StackNode{
public:
    StackNode();
    int data;
    StackNode *next;
private:
};

class Stack{
public:
    Stack();
    bool Push(int data);
    bool Pop();
    int Top();
    bool IsEmpty();
    bool IsFull();
    void MakeEmpty();
    void Print();
    ~Stack();
private:
    StackNode *topPtr;
    int size;
    int maxSize;
};
```

Basic stack code (cont.)

```
StackNode::StackNode(){  
    next = NULL;  
}  
  
Stack::Stack(){  
    topPtr = NULL;  
    size = 0;  
    maxSize = 3;  
}
```


Basic stack code (cont.)

```
bool Stack::Push(int data){
    if(IsFull()){
        throw StackFull();
    }

    size++;
    if(IsEmpty()){
        topPtr = new StackNode();
        topPtr->data = data;
        topPtr->next = NULL;
        return true;
    }

    StackNode *p = new StackNode();
    p->data = data;
    p->next = topPtr;
    topPtr = p;
    return true;
}
```

Basic stack code (cont.)

```
bool Stack::Pop(){
    if(IsEmpty()){ throw StackEmpty(); }

    size--;
    StackNode *p = topPtr;
    topPtr = topPtr->next;
    delete p;
    return true;
}

int Stack::Top(){
    if(IsEmpty()){ throw StackEmpty(); }

    return topPtr->data;
}

bool Stack::IsEmpty(){
    return topPtr == NULL;
}

bool Stack::IsFull(){
    return size >= maxSize;
}
```

Basic stack code (cont.)

```
void Stack::MakeEmpty(){
    while (!IsEmpty()){
        Pop();
    }
}

void Stack::Print(){
    if (IsEmpty()){
        printf("Empty\n");
        return;
    }

    printf("Top\n");
    StackNode *p = topPtr;
    while (p->next != NULL){
        printf("%i\n", p->data);
        p = p->next;
    }
    printf("%i\n", p->data);
    printf("Bottom\n\n");
}

Stack::~Stack(){
    MakeEmpty();
}
```

LISTS

- List: Linear collection of homogeneous items
 - ▶ Can be sorted or unsorted
- Basic list operations:
 - ▶ IsEmpty: Returns if the list has no elements in it
 - ▶ IsFull: Returns if the list is at its maximum capacity
 - ▶ Length: Returns the amount of elements in the list
 - ▶ Insert: Add an item to the list (May implement sorting)
 - ▶ Delete: Delete an item from the list
 - ▶ IsPresent: Check if an item exists in the list

QUEUES

REUSABLE CODE

- Generic programming: Allowing multiple types to be used as parameters by using a template
 - ▶ Template: Code that gets expanded at compile time with the item types implemented within that code
- Uses
 - ▶ Code can be reused without function overloading
 - ▶ Code can be reused for multiple cases

Basic template code (Function template)

```
template <typename type>
type add_squares(type x, type y){
    return x*x + y*y;
}

int main(){
    printf("%i\n", add_squares<int>(2, 3));
    printf("%f\n", add_squares<float>(2.5, 3.5));
    return 0;
}
```

Output

13

18.500000

Basic template code (Function template)

```
template <typename type> class numContainer{
private: type value;
public:
    numContainer(){ value = 0; }
    ~numContainer(){}
    void add(type x){ value += x; }
    type getValue(){ return value; }
};

int main(){
    numContainer<int> i; i.add(2);
    printf("The value is %i\n", i.getValue());

    numContainer<float> f; f.add(2.5);
    printf("The value is %f\n", f.getValue());

    return 0;
}
```

Output

The value is 2

The value is 2.500000

- Function overloading: Using the same function name for different parameter types (not return types)
- Operator overloading: Extending an operator's functionality to work with custom data types and objects
 - ▶ The operators `.` `*` `::` `?:` cannot be overloaded

REUSABLE CODE - CODE

Basic function overloading code

```
int add_squares(int x, int y){  
    printf("int function\n");  
    return x*x + y*y;  
}  
  
float add_squares(float x, float y){  
    printf("float function\n");  
    return x*x + y*y;  
}  
  
int main(){  
    printf("%i\n", add_squares(2, 3));  
    printf("%f\n", add_squares(2.5f, 3.5f));  
}
```

Output

int function

13

float function

18.500000

Basic operator overloading code

```
class weighted_value{
public:
    float value;
    float weight;
    weighted_value(float v, float w){ value = v; weight = w; }
};

float operator + (const weighted_value x, const weighted_value y){
    return x.value*x.weight + y.value*y.weight;
}

int main(){
    weighted_value x(3, 1.5);
    weighted_value y(2.0, 3.0);

    printf("x + y is %f\n", x + y);
    return 0;
}
```

Output

x + y is 10.500000

RECURSION

TREES

HEAPS

- Heap: A complete binary tree that has either the greatest value (max heap) or the least greatest value (min heap) as the root node
- Basic heap operations
 - ▶ Heapify: Rearrange the heap to maintain its order of max heap or min heap
 - ▶ Insert: Add an item to the heap and possibly heapify
 - ▶ Delete: Remove the root node, make the last node the root, and heapify
- Heaps are typically implemented as arrays
 - ▶ root: `array[0]`
 - ▶ parent of *i*th node: `array[(i-1)/2]`
 - ▶ left child of *i*th node: `array[(i*2)+1]`
 - ▶ right child of *i*th node: `array[(i*2)+2]`
- Uses
 - ▶ Priority queues
 - ▶ Sorting algorithms (heap sort)

HEAPS - COMPLEXITY TABLE

Data structure	Insertion	Deletion	Search	Indexing
BST (Average case)	$O(\log n)$	$O(\log n)$	$O(\log n)$	
BST (Worst case)	$O(n)$	$O(n)$	$O(n)$	
Heap	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$
Array	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Linked List	$O(1)$ (at head)	$O(1)$ (at head)	$O(n)$	$O(n)$

GRAPHS

SEARCHING

- Searching: Finding a specific item from a set of data
 - ▶ Efficiency: Program performance is improved
 - ▶ Data retrieval: Specific data is quickly found in a large dataset
 - ▶ Problem solving: Data needs to be found in order to solve problems
- Different methods of searching
 - ▶ Linear search
 - ▶ Binary search
 - ▶ Hashing

- Linear search: Sequentially search through a set of data until the value is found
- Complexity:
 - ▶ Best case: $O(1)$ (The first element)
 - ▶ Worst case: $O(n)$ (The last element)
- Use cases:
 - ▶ Small dataset
 - ▶ Unordered datasets
 - ▶ Linked lists

Basic linear search code

```
int main(){
    int arr[6] = {-1, 7, 12, 17, 3, 4};
    int s = 12;
    for(int i = 0; i < 6; i++){
        if(arr[i] == s){
            printf("%i is at index %i\n", s, i);
            break;
        }
    }
}
```

Output

12 is at index 2

- Binary search: Continually divide the search area in half, comparing the middle value to the target value
- Complexity
 - ▶ Best case: $O(1)$ (The first element)
 - ▶ Worst case: $O(\log n)$ (The last element)
- Use cases:
 - ▶ Data must be sorted
 - ▶ Random access should be a constant time function

Basic binary search code

```
int main(){
    int arr[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
    int s = 5;
    int high = 15; int low = 0;
    int mid;
    while(1){
        if (high >= low){
            mid = low + (high - low) / 2;
        }
        if (arr[mid] == s){
            printf("%i is at index %i\n", s, mid);
            break;
        }
        if (arr[mid] > s){
            high = mid - 1;
            continue;
        }
        low = mid + 1;
    }
}
```

Output

5 is at index 5

- Hashing: Data storage technique designed to allow $O(1)$ search time
 - ▶ Assign key-value pairs through a function to data inputs
 - ▶ Hash function used to store the element and to find if the element exists in the dataset
 - ▶ Tradeoff of memory space for access speed
- Collision: Repeated outputs for different inputs
 - ▶ Must be addressed with hash function (Ex: Offsetting the value)

Basic hashing code

```
#define ArrSize 16

bool hash_insert(int x, int *arr){
    bool isFull = true;
    for(int i = 0; i < ArrSize; i++){
        if(arr[i] == -1){ isFull = false; break; }
    }
    if(isFull){ return false; }
    int index = x % ArrSize;
    while(arr[index] != -1){ index++; index %= ArrSize; }
    arr[index] = x;
    return true;
}

int hash_find(int x, int *arr){
    int index = x % ArrSize;
    while(arr[index] != x){
        index++;
        index %= ArrSize;
        if(index == x % ArrSize){ return -1; }
    }
    return index;
}
```

Basic hashing code (Cont.)

```
int main(){
    int arr[ArrSize];
    for(int i = 0; i < ArrSize; i++){ arr[i] = -1; }

    int x = 347;
    int y = 347 + ArrSize;
    hash_insert(x, arr);
    hash_insert(y, arr);
    printf("%i is at index %i\n", x, hash_find(x, arr));
    printf("%i is at index %i\n", y, hash_find(y, arr));
    return 0;
}
```

Output `fontsize=]code/hashing-output.txt`

SORTING

- Sorting: Organizing data based on its value
 - ▶ Usually based on numeric value or alphabetical value
- Efficiency
 - ▶ Speed: How many comparisons are made and how many swaps are required
 - ▶ Space: How much memory is required
 - ▶ More memory is usually traded for faster speed
- Divide and conquer: Method some algorithms use to sort smaller sections of data and merging them back together

- Selection sort: Continually swap the smallest/largest unsorted value with the first unsorted element
 - ▶ Completes redundant swaps
- Efficiency
 - ▶ Best case: $O(n^2)$
 - ▶ Worst case: $O(n^2)$
 - ▶ Average case: $O(n^2)$
 - ▶ $n - 1$ swaps will always be performed

SORTING - CODE

Basic selection sort code

```
int main(){
    srand(time(0));
    std::vector<int> arr;
    for(int i = 0; i < 10; i++){ arr.push_back(rand() % 100); }

    int swapIndex; int tmp; int min;
    for(int i = 0; i < arr.size(); i++){
        for(int j = 0; j < arr.size(); j++){ printf("%i ", arr[j]); }printf("\n");

        swapIndex = i;
        min = arr[i];
        for(int j = i + 1; j < arr.size(); j++){
            if(arr[j] < min){
                min = arr[j];
                swapIndex = j;
            }
        }
        tmp = arr[i];
        arr[i] = min;
        arr[swapIndex] = tmp;
    }

    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```

SORTING - CODE

Output

```
74 86 32 0 12 60 75 99 15 48
0 86 32 74 12 60 75 99 15 48
0 12 32 74 86 60 75 99 15 48
0 12 15 74 86 60 75 99 32 48
0 12 15 32 86 60 75 99 74 48
0 12 15 32 48 60 75 99 74 86
0 12 15 32 48 60 75 99 74 86
0 12 15 32 48 60 74 99 75 86
0 12 15 32 48 60 74 75 99 86
0 12 15 32 48 60 74 75 86 99
```

Sorted output:

```
0 12 15 32 48 60 74 75 86 99
```

- Bubble sort: Move the smallest/largest value to the front/end of the list
 - ▶ Compare each item to its immediate successor
 - ▶ The next smallest/largest value will be moved to its correct place each pass through
- Efficiency
 - ▶ Best case: $O(n)$, 0 swaps
 - ▶ Worst case: $O(n^2)$, $\frac{n^2}{2}$ swaps
 - ▶ Average case: $O(n^2)$, $(\frac{1}{2})(\frac{n^2}{2})$ swaps

Basic bubble sort code

```
int main(){
    srand(time(0));
    std::vector<int> arr;
    for(int i = 0; i < 10; i++){ arr.push_back(rand() % 100); }

    int tmp;
    for(int i = 0; i < arr.size(); i++){
        for(int j = 0; j < arr.size() - 1 - i; j++){ printf("%i ", arr[j]); }printf("\n");

        for(int j = 0; j < arr.size() - 1 - i; j++){
            if(arr[j] > arr[j+1]){
                tmp = arr[j+1];
                arr[j+1] = arr[j];
                arr[j] = tmp;
            }
        }
    }

    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```

SORTING - CODE

Output

```
41 67 15 2 73 90 8 26 89 57
41 15 2 67 73 8 26 89 57 90
15 2 41 67 8 26 73 57 89 90
2 15 41 8 26 67 57 73 89 90
2 15 8 26 41 57 67 73 89 90
2 8 15 26 41 57 67 73 89 90
2 8 15 26 41 57 67 73 89 90
2 8 15 26 41 57 67 73 89 90
2 8 15 26 41 57 67 73 89 90
2 8 15 26 41 57 67 73 89 90
```

Sorted output:

```
2 8 15 26 41 57 67 73 89 90
```

- Insertion sort: Move each item to its proper place in reference to its predecessors
 - ▶ Assumes the first item is sorted
- Efficiency
 - ▶ Best case: $O(n^2)$
 - ▶ Worst case: $O(n^2)$, $\frac{n^2}{2}$ swaps
 - ▶ Average case: $O(n^2)$, $(\frac{1}{2})(\frac{n^2}{2})$ swaps

Basic insertion sort code

```
int main(){
    srand(time(0));
    std::vector<int> arr;
    arr.push_back(0);
    for(int i = 0; i < 10; i++){ arr.push_back(rand() % 100); }

    int cmpIndex; int key;
    for(int i = 1; i < arr.size(); i++){
        for(int j = 0; j < arr.size(); j++){ printf("%i ", arr[j]); }printf("\n");

        key = arr[i];
        cmpIndex = i-1;
        while(cmpIndex >= 0 && arr[cmpIndex] > key){
            arr[cmpIndex+1] = arr[cmpIndex];
            cmpIndex--;
        }
        arr[cmpIndex+1] = key;
    }

    arr.erase(arr.begin());
    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```


SORTING - CODE

Output

```
0 58 28 3 45 34 74 77 72 83 48
0 58 28 3 45 34 74 77 72 83 48
0 28 58 3 45 34 74 77 72 83 48
0 3 28 58 45 34 74 77 72 83 48
0 3 28 45 58 34 74 77 72 83 48
0 3 28 34 45 58 74 77 72 83 48
0 3 28 34 45 58 74 77 72 83 48
0 3 28 34 45 58 74 77 72 83 48
0 3 28 34 45 58 72 74 77 83 48
0 3 28 34 45 58 72 74 77 83 48
```

Sorted output:

```
3 28 34 45 48 58 72 74 77 83
```

- Quick sort: Divide and conquer sorting algorithm by partitioning around a pivot and recursively sorting each pivot
 - ▶ Values less than the pivot go to one side, and values greater than the pivot go to the other side
 - ▶ Base case: A partition has one element
- Efficiency
 - ▶ Best case: $O(n \log n)$ (Each split generates equally-sized partitions)
 - ▶ Worst case: $O(n^2)$ (Mostly sorted)

Basic quick sort code

```
void quicksort(std::vector<int> &arr){
    if(arr.size() <= 1){ return; }
    int pivot = arr[0];
    std::vector<int> a1; std::vector<int> a2;
    for(int i = 1; i < arr.size(); i++){
        if(arr[i] <= pivot){ a1.push_back(arr[i]); }
        else{ a2.push_back(arr[i]); }
    }
    quicksort(a1); a1.push_back(pivot);
    quicksort(a2);
    arr.clear();
    for(int i = 0; i < a1.size(); i++){ arr.push_back(a1[i]); }
    for(int i = 0; i < a2.size(); i++){ arr.push_back(a2[i]); }
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");
}

int main(){
    srand(time(0));
    std::vector<int> arr;
    for(int i = 0; i < 15; i++){ arr.push_back(rand() % 100); }

    quicksort(arr);
    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```

SORTING - CODE

Output

12 17

12 17 18

11 12 17 18

33 33 35

33 33 35 67 78

11 12 17 18 28 33 33 35 67 78

81 95

81 95 98

80 81 95 98

11 12 17 18 28 33 33 35 67 78 79 80 81 95 98

Sorted output:

11 12 17 18 28 33 33 35 67 78 79 80 81 95 98

- Merge sort: Continually divide the dataset into smaller datasets and sorting and merging them back together
- Efficiency
 - ▶ Best case: $O(n \log n)$
 - ▶ Worst case: $O(n \log n)$

Basic merge sort code

```
void merge_sort(std::vector<int> &arr){
    if(arr.size() <= 1){ return; }
    std::vector<int> a1; std::vector<int> a2;
    for(int i = 0; i < floor(arr.size()/2); i++){ a1.push_back(arr[i]); }
    for(int i = floor(arr.size()/2); i < arr.size(); i++){ a2.push_back(arr[i]); }
    merge_sort(a1); merge_sort(a2); arr.clear(); int a1c = 0; int a2c = 0;
    while(arr.size() != a1.size() + a2.size()){
        if(a1c == a1.size()){ arr.push_back(a2[a2c]); a2c++; }
        else if(a2c == a2.size()){ arr.push_back(a1[a1c]); a1c++; }
        else if(a1[a1c] < a2[a2c]){ arr.push_back(a1[a1c]); a1c++; }
        else{ arr.push_back(a2[a2c]); a2c++; }
    }
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");
}

int main(){
    srand(time(0));
    std::vector<int> arr;
    for(int i = 0; i < 10; i++){ arr.push_back(rand() % 100); }
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    merge_sort(arr);
    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```

SORTING - CODE

Output

75 1 95 65 63 53 70 64 70 80

1 75

63 65

63 65 95

1 63 65 75 95

53 70

70 80

64 70 80

53 64 70 70 80

1 53 63 64 65 70 70 75 80 95

Sorted output:

1 53 63 64 65 70 70 75 80 95

- Sorting elements in a dataset based on its value within a known range (Ex: Leading digit)
- Efficiency
 - ▶ $O(kn)$, where k is the amount of times each data set is sorted

Basic radix sort code

```
int main(){
    srand(time(0));
    std::vector<int> arr;
    for(int i = 0; i < 25; i++){ arr.push_back(rand() % 100); }
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    std::vector<int> r[10][11];
    for(int i = 0; i < arr.size(); i++){ r[arr[i]/10 % 10][0].push_back(arr[i]); }
    for(int i = 0; i < 10; i++){
        for(int j = 0; j < r[i][0].size(); j++){
            printf("%i ", r[i][0][j]); r[i][(r[i][0][j] % 10 + 1)].push_back(r[i][0][j]);
        }
    }printf("\n");
    arr.clear();
    for(int i = 0; i < 10; i++){
        for(int j = 1; j < 11; j++){
            for(int k = 0; k < r[i][j].size(); k++){
                arr.push_back(r[i][j][k]);
            }
        }
    }

    printf("\nSorted output: \n");
    for(int i = 0; i < arr.size(); i++){ printf("%i ", arr[i]); }printf("\n");

    return 0;
}
```

Output

```
13 62 48 35 98 77 75 63 91 29 52 35 8 97 30 65 20 8 13 8 2 35 12 7 18  
8 8 8 2 7 13 13 12 18 29 20 35 35 30 35 48 52 62 63 65 77 75 98 91 97
```

Sorted output:

```
2 7 8 8 8 12 13 13 18 20 29 30 35 35 35 48 52 62 63 65 75 77 91 97 98
```

- Heap sort: Get and remove the maximum value from a sorted heap, then heapify the updated heap
- Efficiency
 - ▶ Heap construction: $O(n)$
 - ▶ Heapify once: $O(\log n)$
 - ▶ Complete sorting: $O(n \log n)$
 - ▶ Initial ordering does not affect efficiency

SORTING - EFFICIENCY TABLE

Efficiencies of sorting algorithms

Algorithm	Best	Average	Worst
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Radix	$O(nk)$	$O(nk)$	$O(nk)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

STL