GENERAL NOTES

ANDREW S. HARD[1]

August 25, 2016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 *Department of Physics, University of Wisconsin, Madison, United States of America*

1 INTRODUCTION

As a fifth year graduate student, I have come to the realization that significant portions of my skillset and knowledge base are highly specialized. I intend to pursue a career outside of my field of study (experimental high-energy particle physics). In order to enhance my future job prospects, I decided that it would be useful to review basic concepts in computer science, statistics, mathematics, and machine learning. Most of the derivations and explanations will be borrowed from other sources. A list of references is currently being compiled.

2 DATA STRUCTURES

2.1 Java Basics

In Java, there are **primitive types** and **reference types**. Primitive types are like `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, `byte`. Reference types are **arrays** and **classes**.

When you declare a variable with a reference type, you get space for a reference or **pointer** to that type, not for the type itself. Use `new` to get space for the type itself. Non-primitive types are really pointers. Assigning 1 variable to another can cause **aliasing** (two or more different names reference the same object).

For each **primitive type** there is a "box class" or "wrapper class". For instance, the `Integer` class contains one `int` value. Java **autoboxing** converts between the primitive and wrapper classes as appropriate.

Java also has **visibility modifiers**, which specify how code can be accessed.

- *public* accessible to any code
- *private* accessible to code within same class
- *protected* accessible to code within same class or subclasses
- *public* accessible to code within same package (directory)

Numerical comparisons can be performed using the "<", ">", "=" signs for `int`, `float`, `double` primitives. However, the **Comparable** interface should be used for objects in Java (`String`, `Integer`, `Double`). The `compareTo` method returns negative value if "less than", zero if "equal to", and positive if "greater than".

2.2 Complexity

Complexity is a way to gauge the performance of an algorithm: how much of the computing resources are used when a program executes. **Complexity** measures how the resource requirements scale as the problem gets larger. The time requirement is basically proportional to the number of basic operations such as '+', '*', one assignment, one logical test, one read, or one write. The **problem size** is related to the number of important factors. In sorting, for example, this could be the number of elements to sort.

We are usually interested in the worst case number of operations, not the exact number of operations, which can vary. The average and best cases do provide useful information, however. We use the *big-O* notation to write complexity: $O(N)$ is linear complexity, $O(N^2)$ is quadratic complexity, etc.

Amortized complexity is the total expense per operation, but is evaluated over many operations. This is useful when operations are occasionally much more expensive than on average. For instance, insertions in a hash table typically take constant time. However, if the array implementation needs to be expanded, the operation might take $O(N)$ time for a single operation. Amortized complexity is just a way of describing the worst case on-average behavior.

2.3 Abstract Data Types

Abstract Data Types (**ADTs**) separate the *specification* of a data structure from the *implementation* of that data structure. The specification refers to the methods and operations that can be performed on it, while the implementation refers to how those features are programmed. The advantage to using ADTs is that code is more reusable: the implementation of certain code can be changed without changing the programs that use the code. An ADT corresponds to a class or many implementing classes. Operations on an ADT are the class's **public methods**. Example operations on an ADT include: initialize, add data, access data, remove data.

ADTs have 2 parts.

- public/external - conceptual picture & conceptual operations
- private/internal - the representation (how the structure is stored) and the implementation of operations (code)

To implement an interface in Java: (1) include "`implements InterfaceName`" in the class declaration, changing `InterfaceName` as appropriate, (2) define a public method for the class for each method signature in the interface.

To specify that something is a subclass, use `extends`. This says that the current class definition inherits from the specified class. For instance, `MyException` inherits from the `Exception` class in the example below.

```
public class MyException extends Exception {...};
```

2.4 Exceptions

Errors can arise due to *user error* (e.g. wrong inputs) or *programmer error* (e.g. buggy code). It is desirable to pass the error up from low-level methods to a level where it can be handled properly. In C++, methods usually just return a special value to indicate an error. Requires calling code to check for error. In Java, expections allow error testing in the code.

The idea is to **throw** an **exception** when an error is detected. The code causing the error stops executing immediately and control is transferred to the **catch clause** for that type of exception of the first enclosing **try block** that has such a clause. The try block might be in the current method or in a method that called the current one. The exception is passed up the call chain, all the way up to `main` or the Java virtual machine if necessary (at which point the program stops and an error message appears). The code below provides an example.

```
try {
  // code that might cause exceptions.
} catch( ExceptionType1 id1 ) {
  // statement to handle this exception type.
} catch( ExceptionType2 id2 ) {
  // statement to handle this exception type.
} finally {
  // statement to execute every time this try block executes.
}
```

Each `catch` clause specifies one type of exception and provides a name for it, like method objects. The `finally` clause is optional. It *always* executes if the `try` block is entered. Note: you can have zero catch clauses and still have a finally clause.

Java's standard libraries have several built-in exceptions: `ArithmeticException`, `ClassCastException`, `IndexOutOfBoundsException`, `NullPointerException`, `FileNotFoundException`...

All exceptions are either **checked exceptions** or **unchecked exceptions**. If a method includes code that could cause a checked exception to be thrown: (1) the exception must be declared in the method header using a **throws** clause, or (2) the code that might cause the exception to be thrown must be in a try block with a catch clause for that exception. Otherwise you will get a compiler error. Unchecked exceptions do not need to be listed in the throws clause of a method. The code below provides an example.

```
  public static void main( string[] args ) throws FileNotFoundException,
EOFException {
    // those two exceptions might be thrown here.
  }
```

Unchecked exceptions are subclasses of `RuntimeException`, while checked exceptions are subclasses of `Exception` only, not `RuntimeException`. Most built-in Java exceptions are unchecked, with the exception of I/O exceptions, which are checked. user defined exceptions should usually be checked.

Java exceptions are objects, and as such a new exception is made by defining a new instantiable class. It must be a subclass of `Throwable`, usually a subclass of `Exception`. An example is given below.

```
public class EmptyStackException extends Exception {
  public EmptyStackException() { super(); }
  public EmptyStackException(string message) { super(message); }
};
```

At the point in code where an error is detected, the exception is thrown using a `throw` statement.

```
if (...)  { throw new EmptyStackException(); }
```

2.5 List Structure

2.5.1 *Basic List*

A **List** structure is an ordered collection of items of some element type *E*. One advantage of lists over arrays is that they are variable size. A disadvantage is that you cannot have an empty list with non-zero size. Lists only hold objects, whereas arrays hold any type (even primitive types such as `int`, `char`).

The table below provides the public interface for the `ListADT`.

`void add(int pos, E item);`	Add an item at a specified position
`void add(E item);`	Append item to end of list
`boolean contains(E item);`	Check if list contains item
`int size();`	Get the number of items in the list
`boolean isEmpty();`	Check if list contains zero items
`E get(int pos);`	Get the item at the specified position
`E remove(int pos);`	Remove, return item at specified pos.

To instantiate `ListADT`, for a list of `Integer` type items named `number` using the `ArrayList` class:

```
ListADT<Integer> numbers = new ArrayList<Integer>();.
```

`Iterator` is a defined interface in `java.util`. Iterating accesses each item in a list in turn. The interface is `Iterable`. Every java class that implements Iterable (from `java.lang` provides an `iterator` method that returns an `Iterator` object for that collection. For example, the `List<E>` interface has an `iterator()` method:

```
Iterator<string> itr = words.iterator(); //words is a List<String>
```

The easiest way to implement an iterator for a class is usually to define a new class. For example, in the `SimpleArrayList` class, the `iterator()` method should return a new `SimpleArrayListIterator` object.

Here's a simple overview of an iterator class:

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class SimpleArrayListIterator implements Iterator<Object>
{
   // methods:
   public boolean hasNext(){...}
   public Object next(){...}
   public void remove(){...}
};
```

2.5.2 *Linked Lists*

Linked lists are a type of list in which each item in the list contains the item data as well as a reference to the next element in the list. The list is built using `Listnode` objects that store data as well as the next listnode. The `getNext()` method returns the next item in the list, which can also be set using the `setNext(Listnode<E> n)` method.

Linked lists often have a **header node** – a dummy node at the front of the list. Header nodes do NOT contain data, the only include a reference for the next item. A **tail pointer** references the last listnode.

There are many variants of linked lists, including doubly linked lists (which have a reference to the previous and next elements) and circularly linked lists (where the tail node `next` is linked to the first node).

2.6 Stacks and Queues

Stacks and queues are like lists, but with more limited operations. They can be implemented using an array or linked list.

2.6.1 *Stacks*

A **stack** is a stack of items. Only the top item can be accessed. Items are added to the top of a stack. It is a basic **LIFO** structure: a last-in-first-out data type. A basic stack interface is provided below.

```
public interface StackADT<E> {
  boolean isEmty(); //return true iff empty
  void push(E obj); // add object to top of stack
  E pop() throws EmptyStackException; //remove
  E peek() throws EmptyStackException; // see
};
```

2.6.2 *Queue*

A **queue** is a structure similar to a line or queue. Items go in one side and out the other. It is a basic **FIFO** structure: a first-in-first-out data type. Items are added to the *rear* of the queue and removed from the *front* of the queue.

```
public interface QueueADT<E> {
  boolean isEmty(); //return true iff empty
  void enqueue(E obj); // add object to end of queue
  E dequeue() throws EmptyQueueException; //remove from front
};
```

2.7 Recursion

A method is **recursive** if it can call itself either directly or indirectly. Recursive code is often *simpler*, but not faster or more efficient. There are a few rules associated with recursion.

- *Recursion rule 1*: recursive methods must have a **base case** - a condition under which no recursive call is made. This prevents infinite recursion.
- *Recursion rule 2*: every recursive method **make progress** towards a base case.

At runtime, a stack of activation records (AR) is maintained (one for each active method). This is also named the **call stack**. It includes the method's parameters, local variables, and return address. When a method is called, its AR is pushed onto the stack. When too many methods are called, a **stack overflow** error is produced.

A data structure is recursive if it can be defined in terms of itself.

2.8 Searching in an Array

There are two basic approaches to searching in an array: **sequential** search and **binary** search. Sequential search involves looking at each value in turn. It is faster for sorted arrays O(N). Binary search (for sorted array) involves looking at the middle item, comparing with the value of interest, then eliminating half of the array from the search O(log(N)).

2.9 Tree Structure

So far, the structures discussed have been **linear data structures**. These incude lists, stacks, queues, and other structures in which items are stored with one item following another. **Trees** are an example of a **nonlinear data structure**. More than one item can follow from a single item. Furthermore, the number of following items can vary from one item to another. Trees are useful because they can provide *fast access* to information in a database.

Some tree terminology is provided in the table below.

Node	A single element of the tree
Edge	Arrow connecting two nodes
Root	Top-most node of the tree (has no **parent**)
Leaf	bottom nodes (has no **children**)
Ancestors	The preceding nodes in a tree
Descendants	The succeeding nodes in the tree
Path	A sequence of conected nodes
Path Length	Number of nodes in the path
Height	Length of *longest* path from root to a leaf
Depth	Path length from a given node back to root

2.9.1 *Binary Trees*

Binary trees are a special type of tree in which (1) each node has 0, 1, or 2 children and (2) each child is either a *right* or *left* child.

Programming a binary tree usually requires a tree class and a tree node class. The `BinaryTreenode` class only requires 3 fields: `data`, `leftChild`, and `rightChild`. A generic tree, by comparison, could just use a list of nodes for the children.

There are several ways to traverse a tree, as shown in the table below. V=view/visit, L=go to left child, R=go to right child.

Name	*for binary tree*	*for other tree*
pre-order	VLR	VC
post-order	LRV	CV
level-order	left to right from top to bottom	
in-order	LVR	(binary only)

2.9.2 *Binary Search Trees*

A **binary search tree** (BST) is a type of binary tree that stores sorted values. For every node *n* with a **key value** (and maybe data):

- all keys in *n*'s left sub-tree are less than the key in *n*,
- all keys in *n*'s right sub-tree are greater than the key in *n*.

A binary search tree has efficient implementations of the insert, contains, remove, and print methods. Example structures for the node and tree classes are provided below.

```
class BSTnode<K> {
  private K key;
  private BSTnode<K> left, right;
  public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {...} ;
  //accessors:
  public K getKey(){...}
  public BSTnode<K> getLeft(){...}
  public BSTnode<K> getRight(){...}
  //mutators:
  void setKey(K key){...}
  void setLeft(BSTnode<K> left){...}
  void setRight(BSTnode<K> right){...}
};

public class BST<K extends Comparable<K> > {
  private BSTnode<K> root;// root of BST
  public void insert(K key){...}
  public void delete(K key){...}
  public boolean lookup(K key){...}
  public void print(PrintStream p){...}
};
```

Of course, it is also possible to design a BST that stores data along with each key.

The `lookup(K key)` method is an example of a recursive algorithm. The base cases are an empty tree, which returns false, or value is in the root node, which returns true. Check the root, left, and right subtrees. The worst-case lookup for a BST is $O(N)$, and corresponds to a stalky shape. Best case performance for a BST lookup is $O(\log(N))$ for a full, balanced tree.

Question: how do you **delete** a node in a tree with children? And with what do you replace the deleted node? Possibilities:

- the largest value in the left subtree (**in-order predecessor**).
- the smallest value in the right subtree (**in-order successor**).

The Java standard library implements a BST with the `TreeSet` and `HashSet` classes. Use `Map` and `TreeMap` to associate data with each key.

A few additional tree types are listed below.

- **Full tree**: all the leaves have the same depth.
- **Complete tree**: full to depth H-1, depth H filled from left to right.

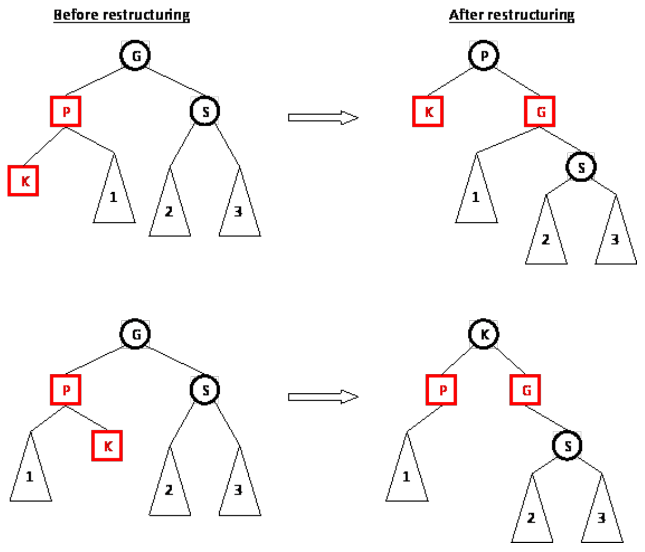

Figure 1: An example of a **tri-node restructuring** procedure for a red-black tree (c.o. James D. Skrentny, University of Wisconsin CS367).

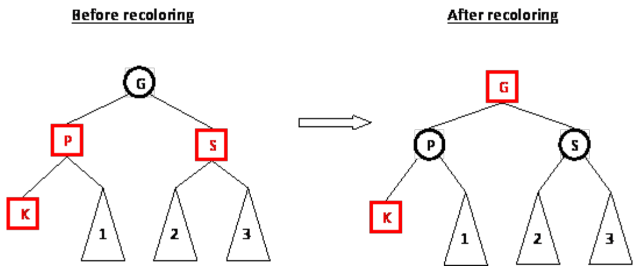

Figure 2: An example of a **recoloring** procedure for a red-black tree (c.o. James D. Skrentny, University of Wisconsin CS367).

- **Height-balanced tree**: difference in heights of *left* and *right* subtrees is at most 1.
- **Balanced tree**: has a height of $O(\log(N))$, where N=# nodes.

2.9.3 *Red-Black Trees*

Red-black trees (RBT) are a type of balanced tree. In order to preserve balance, the `insert` and `delete` operations may restructure the tree. Each node of the tree has a **color** (red or black), and 3 properties hold:

- **Root property**: the root of the tree is black.
- **Red property**: children of a red node are black.
- **Black property**: for each node with at least 1 null child, the number of black nodes on the path from the root to the null child is the same.

We want to perform insert, lookup, delete, and print operations on this tree. Lookup and print for the RBT are identical to those of the BST. The lookup, insert, delete operations have $O(\log(N))$ complexity for the worst-case.

Here's an overview of the RBT insert algorithm for node with key K.

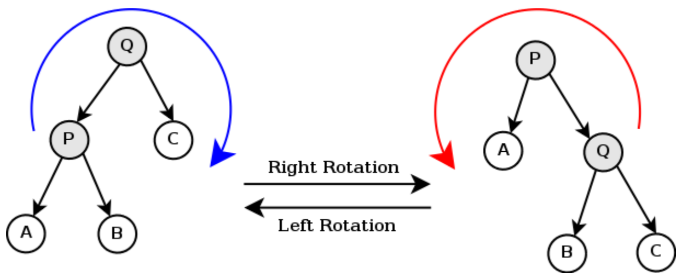

Figure 3: An example of a tree rotation (By Ramasamy at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4043513).

- Use the BST insert algorithm to add K to the tree.
- Color the node containing K **red**.
- Restore the RBT properties if necessary.

Insertion can lead to a red property violation **RPV**. Here's how to deal with that.

- *Case 1*: if K's parent P is black, then do nothing (no RPV).
- *Case 2a*: if P's sibling is black or null, do a **trinode restructuring**.
- *Case 2b*: if P's sibling S is red, do a **recoloring** of P, S, G (unless G is the root).

2.9.4 *Balanced Search Trees*

There are other types of search trees that preserve the O(log(N)) tree height that produces efficient operations for N nodes. Other than red-black trees there are also AVL and BTree objects.

An **AVL** is a height-balanced binary search tree. It was the first such structure to be invented, and is named after its Soviet inventors.

- Keeps a balance value for each node (-1, 0, +1).
- Detect problem when balance becomes ± 2.
- Fix by **rotation**.

BTrees relax the binary tree structure by allowing 2, 3, 4 descendant nodes (the number of allowed nodes is called the order). BTrees are also referred to sometimes as 2-3 trees or 2-3-4 trees. When 4 nodes are detected, they are split.

2.10 Priority Queues

Priority queues are a data structure where priorities are put into the object and highest priorities are removed from the object. Here are the basic priority queue operations:

```
boolean isEmpty()
void insert(Comparable p);
void insert(Comparable p);
Comparable removeMax();
```

```
    Comparable getMax();
```

To implement the priority queue efficiently, a new data structure called a heap will be implemented. A **heap** is a binary tree with two special properties.

- **Order property** for every node N, the value N is greater than or equal to the values in its children.
- **Shape property**
  - All leaves are at depth d or $(d-1)$.
  - All of the leaves at depth $(d-1)$ are to the right of the leaves at depth d.
  - There is at most 1 node with just 1 child. That child is the left child of its parent. It is the rightmost leaf at depth d.

The **root** of the heap is always in `array[1]`. If a node is in `array[k]`, the left child is in `array[2*k]`, the right child is in `array[2*k+1]`, and the parent is in `array[k/2]`.

Implementing **heap insert**:

- add new value at the end of the array (preserve shape),
- swap with parent until order property holds, or get to root.

Implementing **heap removeMax**:

- replace the value with the value at the end of the array (removing leaf),
- work way down the tree, swapping value with the larger child to restore the order property.

The insert and remove operations described above are for a **max heap**. Of course, it is also possible to have a **min heap** along the same lines.

2.11 Hashing

Hashing is a technique that is $O(1)$ average complexity for insert, lookup, and delete operations. The basic idea is to store values (*keys*) in an array, computing each key's position in the array as a function of its value. The array storing data is referred to as the **hash table**, while the function that maps keys to array indices is called the **hash function**. It is good to use efficient hash functions. For example, one can convert the key value to an integer modulo the table size.

One issue that arises with hashing is that multiple data items might map to the same array index. **Collisions** occur when key values map to the same index. There are a few tricks that help to avoid collisions. First, it is best to use a table that is 1.25 times the size of the expected number of items and then to expand the hash table as necessary. Second, it is good to have a *prime* table size, so that hashing functions that include multiplication by some factor don't send all keys to the same bucket after the hash function modulation. Ideally, you want the generator for the hash index to use a constant that is co-prime with the hash table length.

Buckets are used to handle keys that get mapped to the same array index. *LinkedList* buckets can be used for the buckets. This structure would give

$O(1)$ complexity for insertion but $O(N)$ worst-case complexity for other operations. Another option would be to use a BST bucket, which would give $O(\log(N))$ worst-case complexity when all keys are mapped to one index.

Java has a `hashCode()` method for every Object, including `Strings`.

Another way to handle collisions is through open addressing. If there is a collision, search for an unused element elsewhere in the table using a **probe sequence**. The list below uses H to represent the hash function result for an item k:

- *linear probing*: $H(k), H(k)+1, H(k)+2, \ldots$ % table size
- *quadratic probing*: $H(k), (H(k)+1)^2, (H(k)+2)^2, \ldots$ % table size
- *double hashing*: $H(k), (H(k)+\mathtt{step}), (H(k)+2\cdot\mathtt{step}), \ldots$ % table size

2.12 Graph Structure

Graphs are a generalization of the tree structure. They have **nodes** connected by **edges** (also called *vertices* or *arcs*). A nide can have any number of incoming or outgoing edges. There are directed and undirected graphs.

Undirected graphs have nodes which are connected by edges without any direction. Connected (adjacent) nodes are simply referred to as **neighbors**.

Directed graphs have neighbor nodes that are connected by edges with a **direction**. The source of the edge is referred to as the **predecessor** node, while the target of the edge is referred to as the **successor** node.

Paths can be taken through the nodes in a graph. A *cyclic path* is a path in which the same node occurs twice. An *acyclic path* path is a path in which no node is repeated. Note: a node can connect an edge to itself.

Here are some types of graphs:

- *directed acyclic graph (DAG)*: a directed graph that has NO cyclic paths.
- *Complete graph*: an undirected graph that has an edge between every pair of nodes, or a directed graph that has an edge from every node to every other node.
- *Weighted graph*: A graph with values associated with the edges.
- PtextitNetwork: A weighted di-graph (directed graph) with non-negative weights.

2.12.1 *Depth-First Search*

A **depth-first search**, or DFS, is a method for searching through the nodes in a graph. It usually is implemented with a *stack* or *recursive* structure. The idea is to start at some node n, then follow an edge of of n, then another edge, getting as far away from n as possible before visiting any more of n's successors.

- start with all nodes marked "unvisited"
- mark n "visited"
- recursively do a DFS from each of n's unvisited successors.

The time for this search is $O(N+E)$, where N=# nodes and E=# edges in the graph. A boolean in the node class could store visited information.

There are many applications of the depth-first search. It can answer questions like the following.

- Is a graph connected?
- Does the graph contain a cycle?
- Is there a path between two nodes?
- What nodes are reachable from the given node?
- Can the nodes be ordered so that, for every node j, j comes before all of its successors?

Depth-first search is similar to the *pre-order* traversal of the tree structure.

2.12.2 *Breadth-First Search*

A **breadth-first search**, or BFS, is a method for searching through the nodes in a graph. The idea is to visit all nodes at the same distance from the starting node before visiting farther nodes. It is usually implemented with a *queue* structure rather than through recursion.

- Start with all nodes "unvisited" and enqueue(n).
- Dequeue, mark node as visited, add the node's children to the queue if unvisited.
- Repeat previous line until queue is empty.

Breadth-first search is similar to the *level-order* traversal of the tree structure. It can be used to find the shortest path between points. *Djikstra's algorithm* is an example of this. It finds the shortest path through a weighted non-negative graph and has complexity $O(E\log(N))$.

2.13 Sorting

Sorting involves putting the values of an array into some order. **comparison sorts** work by comparing values. Simple algorithms are $O(N^2)$, though the best-possible performance is $O(N\log(N))$.

Name	Complexity
selection sort	w.c. $O(N^2)$
insertion sort	w.c. $O(N^2)$
merge sort	w.c. $O(N\log(N))$
quicksort	w.c. $O(N^2)$, average $O(N\log(N))$

Stable sorting algorithms also preseve the relative ordering for duplicate keys from the previous sorting.

2.13.1 *Selection Sort*

Selection sort is an $O(N^2)$ complexity sorting algorithm. The basic approach is to find the smallest value in an array `A` and then put it in `A[0]`. Then find the n^{th} smallest value in `A` and put it in `A[n-1]`.

- Use outer loop from `0` to `n-1`, letting `k` represent the index.
- Use a nested loop from `k+1` to `n-1` to find index of smallest value.
- Swap that value with `A[k]`.
- After i^{th} iteration, `A[0]` through `A[i-1]` are ordered.

Complexity = $(N-1)+(N-2)+...+1+0=O(N^2)$.

2.13.2 *Insertion Sort*

Insertion sort is another $O(N^2)$ complexity sorting algorithm. The basic approach is to put the first two items in correct relative order. Then insert the 3rd item in the correct place relative to the first two. Then insert the n^{th} item in the correct place relative to the previous $n-1$.

- Use outer loop from `k=1` to `k=n-1`.
- Use inner loop from `j=k-1` to `0` *as long as* `A[j] > A[k]`.
- Each time, shift higher numbers up (`A[j+1]=A[j]`) to make space for eventual insertion.
- Insert `A[k]` into final `A[j]`.
- After the i^{th} iteration, `A[0]` through `A[i-1]` are relatively ordered but are not in the final position.

2.13.3 *Merge Sort*

Merge sort is an $O(N\log(N))$ *divide and conquer* algorithm. It takes advantage of the fact that it is possible to merge two sorted arrays, each containing $N/2$ items, in $O(N)$ time. It works by simultaneously stepping through the two arrays and always choosing the smaller value to put in the final array.

- Divide the array into two halves.
- Recursively sort the left half.
- Recursively sort the right half.
- Merge the two sorted halves (requires temporary auxiliary array).

2.13.4 *Quick Sort*

Quick sort is another on average $O(N\log(N)$ divide and conquer sorting algorithm. Compared with merge sort, it does more work during the "divide" part in order to avoid work in the "combine" part. The idea is to start by *partitioning* the array using some *median* value (or *pivo* value). Then use 2 pointers at opposite ends of the array to perform swaps of values.

- Choose a pivot value – put pivot at end of array (swap with existing).
- Partition the array. Put all entries less than pivot in the left part and all entries greater than the pivot in the right part, with the pivot in the middle.
- Recursively, sort the values less than or equal to the pivot.
- Recursively, sort the values greater than or equal to the pivot.

Note: a poorly chosen pivot can lead to $O(N^2)$ complexity. The **median of three** technique can be useful for choosing a pivot: pick median value from sampling beginning, middle, and end of array. Upper value placed at end of array, median placed next to end, low value placed at beginning of array.

Note that the quick sort algorithm does not require extra storage space, unlike the merge sort algorithm.

2.13.5 *Heap Sort*

Heap sort is an $O(N \log(N))$ complexity sorting algorithm. The idea is to insert each item into an initially empty heap. Then fill the array right-to-left as follows: while the heap is not empty, do one `removeMax` operation and put the returned value into the next position of the array.

2.13.6 *Radix Sort*

Radix sort is not a comparison sort. It is useful on sequences of *comparable* values, like sequences of characters or numbers. The time is $O((N + R) * L)$, where N is the number of sequences, R is the range of values ieach item could have, L is the maximum length of the sequences. The approach uses an array of queues.

- Process each sequence right-to-left (least to most significant digit).
- Each pass, values taken from original array and stored in a queue in the auxiliary array based on the value of the current digit. Note: the auxiliary array has length L, and the queues at each array position handle duplicates etx.
- Queues are dequeued back into the original array, ready for next pass.

The queue structure preserves previous ordering. It is often *better* than $O(N \log(N))$.

2.13.7 *Bubble Sort*

Bubble sort is a sorting algorithm of complexity $O(N^2)$. Each pass through the unsorted part "bubbles" the next smallest item from unsorted to the back of the sorted part.

- swap `A[j]` with `A[j-1]` if it is smaller, so that small values bubble all the way down.

2.14 Cache Algorithms

Cache algorithms are instructions that a computer can follow in order to maintain a cache of stored information. A **cache** is just a data storage system that can serve up data faster. When the cache is full, the algorithm must decide which item to discard in order to make room for the new item. A few examples of caching algorithms are LRU, MRU, and RR. The **least recently used** (LRU) algorithm discards the least recently used item in the cache when making space. The **most recently used** (MRU) algorithm discards the most-recently used item in the cache to make space. The **random replacement** (RR) just randomly selects a candidate item to discard.

2.15 Dynamic Programming

Dynamic programming is a method of solving problems. It is similar to a *divide and conquer* strategy, in that a large problem is solved by breaking it down into similar, smaller problems of the same type. However, dynamic programming is typically used with polynomial complexity in scenarios where straightforward divide and conquer or recursion would require exponential complexity.

2.16 Data Structures in C++

The following table provides a description of some basic C++ data structures.

Name	Description
`list`	Lists are sequence containers that allow constant time insertion and erase operations anywhere within the sequence, as well as iteration in both directions (doubly-linked). Similar to `forward_list` except that `forward_list` objects are single-linked lists, and can only be iterated forwards. They lack direct access to elements by their position, and require linear time in the distance between iterator position and target to lookup. They also consume extra memory to store linking information.
`vector`	A sequence container representing array that can change in size. Use dynamically allocated array to store elements. Memory reallocations should only happen at logarithmically growing intervals of size, so that insertion at the end of a vector can be provided with amortized constant time complexity.
`deque`	The double-ended queue is a sequence container with dynamic size that can be expanded or contracted on both ends. Usually implemented as some dynamic array. Unlike vectors, deques are not guaranteed to store all elements in contiguous storage locations. Any elements can be accessed in constant time. Worse than lists when frequent internal insertions or removals are needed.
`stack`	A LIFO structure, container adaptor. Standard container classes are `vector`, `dequeue`, `list`.
`map`	Slower than `unordered_map` for lookup, but faster at iterating over (ordered) elements. Often implemented as a binary search tree, with O(log(N)) lookup time.
`unordered_map`	Implemented as a hash map in C++. Constant time lookup for the most part, amortized linear time lookup.

Concerning exceptions, C++ allows the user to use `try`, the `throw`, and catch as in Java. The difference is that in the catch clause, you list the exception *type*:

```
try {
  throw 20;
} catch (int e) {
  cout << "An exception occurred, no.  " << e << endl;
}
```