

CS 311 F01 Data Structures and Algorithms, Spring 2008  
**Midterm Exam Solutions**

The Midterm Exam was given in class on Friday, March 21, 2008.

1. [4 pts] **Parameter Passing in C++.** In the table below, the names of the three ways to pass parameters in C++ are written in the first row. In the rest of the table, circle “YES” or “NO” to answer the questions for each parameter-passed method.

Does this method ...	BY VALUE	BY REFERENCE	BY REFERENCE-TO-CONST
... make a copy of the object passed?	<input checked="" type="checkbox"/> YES / NO	YES / <input checked="" type="checkbox"/> NO	YES / <input checked="" type="checkbox"/> NO
... allow const objects to be passed?	<input checked="" type="checkbox"/> YES / NO	YES / <input checked="" type="checkbox"/> NO	<input checked="" type="checkbox"/> YES / NO
... allow for polymorphism/virtual dispatch?	YES / <input checked="" type="checkbox"/> NO	<input checked="" type="checkbox"/> YES / NO	<input checked="" type="checkbox"/> YES / NO
... allow implicit type conversions to be performed?	<input checked="" type="checkbox"/> YES / NO	YES / <input checked="" type="checkbox"/> NO	<input checked="" type="checkbox"/> YES / NO

2. [6 pts total] **Properties of Algorithms.**

2a. [2 pts] We are interested in “efficiency” of algorithms. What does this mean in a general sense? When we talk about efficiency, we are usually interested in something very specific; what is it? *Note: There are two questions here.*

An algorithm is *efficient*, in the general sense, if it uses few resources (time, memory, disk space, network bandwidth, semaphores, etc.). The most important resource is usually time. Thus, in practice, efficient = fast.

2b. [2 pts] What does it mean for an algorithm (or method or idea) to be “scalable”?

An algorithm is *scalable* if it works well when used to solve increasingly large problems.

2c. [2 pts] What does it mean for an algorithm to be “stable”?

An algorithm that reorders data items (for example, a sorting algorithm) is *stable* if it does not reorder items unnecessarily. In particular, a sorting algorithm is stable if it does not change the order of equivalent items.

3. [6 pts total] **Using Iterators.** Suppose we have variables `v1` and `v2`, declared as follows:

```
const std::vector<int> v1(SIZE);
std::vector<int> v2(SIZE);
```

3a. [3 pts] Write an **iterator-based loop** that finds the sum of the items in `v1`, and stores the result in the variable `sum`. *Note that `v1` is const.*

```
int sum = 0;
std::vector<int>::const_iterator cit;
for (cit = v1.begin();
     cit != v1.end();
     ++cit)
{
    sum += *cit;
}
```

*Note: We must use a `const_iterator`, since `v1` is const.*

3b. [3 pts] Using one of the Standard Template Library algorithms, write a **single function call** that copies the data in `v1` to `v2`. Note that they are already the same size.

```
std::copy(v1.begin(), v1.end(), v2.begin());
```

4. [7 pts] **Writing an Operator.** Suppose you are writing a class `Info`. Suppose that `Info` has a member function `print` (already written), which outputs the object to a given stream. Function `print` is prototyped as follows:

```
void Info::print(std::ostream & out) const;
```

Write a **global function** implementing `operator<<` (stream insertion) for `Info`. This should output the object in the same format as function `print` does. You may assume that any necessary system headers have been included. Other than the existence of function `print`, make no assumptions about the structure or member functions of class `Info`.

```
std::ostream & operator<<(std::ostream & out,
                        const Info & x)
{
    x.print(out);
    return out;
}
```

5. [10 pts total] **Recursion vs. Iteration.**

5a. [4 pts] Recursion has a number of disadvantages, as contrasted with iteration. Give at least two of these.

Here are three:

- Function-call overhead.
- Repeated memory allocation on the system Stack, which does not allow for error handling and recovery, and often results in increased memory usage.
- Many recursive algorithms are inherently inefficient.

5b. [2 pts] Under what circumstances is it **very easy** to convert recursive code to iterative form?

When the code is tail-recursive.

5c. [4 pts] Briefly explain how the conversion in the previous part could be done.

Surround the function body with a “while (true)” loop. Replace the tail-recursive call with code to set the parameters to their new values.

6. [4 pts total] **Order of Functions.** Write the order of each function, using big- $O$ .

6a. [2 pts]

```
void func_a(int arr[], int n) // n is size of arr
{
    cout << "Item 0 = " << arr[0] << endl;
    cout << "Item 1 = " << arr[1] << endl;
}
```

**Order (big- $O$ ):**  $O(1)$

6b. [2 pts]

```
void func_b(int arr[], int n) // n is size of arr
{
    for (int a = 0; a < n; ++a)
        for (int b = 0; b < a; ++b)
            for (int c = 0; c < a; ++c)
                arr[c] += arr[b];
}
```

**Order (big- $O$ ):**  $O(n^3)$

7. [4 pts] One might think that, as computers get faster, we would stop worrying about efficient algorithms. And yet, your instructor mentioned this quote from Nick Trefethen.

The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less.

Explain the thinking behind this quote.

Algorithmic efficiency involves how much an algorithm slows down as the problem size becomes larger. Thus, efficient algorithms become more important as the problem size grows. And as computers become more powerful, the problems we want to solve with them become larger.

8. [10 pts total] **Order of Useful Operations.** In each part, indicate the order of the given operation, first using **big- $O$**  and then, based on this, **in words** (e.g., “exponential time”). Assume that a good algorithm is used—within the constraints given. Use  $n$  to denote the number of items in the list that the algorithm is given as input.

8a. [2 pts] Finding an item with a given value in a sorted array.

**Big- $O$ :**  $O(\log n)$ . [*Use Binary Search.*]

**In Words:** Logarithmic time.

8b. [2 pts] Printing the middle item in an array whose size is known.

**Big- $O$ :**  $O(1)$ .

**In Words:** Constant time.

8c. [2 pts] Printing the middle item in a Linked List whose size is known.

**Big- $O$ :**  $O(n)$ .

**In Words:** Linear time.

8d. [2 pts] Sorting an array with Quicksort.

**Big- $O$ :**  $O(n^2)$ . [*Remember: worst-case.*]

**In Words:** Quadratic time.

8e. [2 pts] Sorting a Linked List.

**Big- $O$ :**  $O(n \log n)$ . [*Use Merge Sort.*]

**In Words:** Log-linear time.

9. [12 pts total] **Impossible?** Your friend Egbert is known for making wild claims. In each part below, Egbert claims he has a new algorithm. Indicate whether each of Egbert's claims is **possible** or **impossible** (circle one). If it is possible, explain how to do it. If it is impossible, explain why.

9a. [4 pts] Egbert says, "I have a new algorithm that can find the largest number in a given unsorted array in logarithmic time."

POSSIBLE

**IMPOSSIBLE**

Since the array is unsorted, looking at some items gives us no information about the value of other items, and so we may need to look at every item in the array. Thus, in the worst case, this problem cannot be solved without reading all the input, which requires at least linear time.

9b. [4 pts] Egbert says, "I have a new algorithm that can sort a list in linear time, assuming that each item's position is no more than 1000 away from its position in the final sorted list."

**POSSIBLE**

IMPOSSIBLE

Egbert is claiming that his algorithm can sort a nearly-sorted list in linear time. We already know how to do this: use Insertion Sort.

9c. [4 pts] Egbert says, "I have a new algorithm that can sort any list in linear time, given only an appropriate comparison function."

POSSIBLE

**IMPOSSIBLE**

Since Egbert says, "any list", he is claiming that his algorithm solves the General Sorting Problem in linear time. But we know that the General Sorting Problem cannot be solved by an algorithm in any efficiency class faster than log-linear time.

10. [12 pts total] **Error Handling.** Consider the following function `setItem`, which is a member of a class `Ar`, an array-like container class.

```
// Ar::setItem
// Set item with the given index to the given value.
// Pre: None.
// Post: data_[index] == v.
void Ar::setItem(int index, const Ar::value_type & v)
{ data_[index] = v; }
```

If parameter `index` is out of range, this function will not work. We discussed three ways of dealing with problems like this. In each part below, deal with the problem in a different way. First, indicate generally your method of dealing with the problem, and then **exactly** how you would change the code and/or the comments. Do not rewrite the entire function; just indicate the changes. *Part a is partially done for you.*

10a. [4 pts]

**Method of dealing with the problem:**

*Signaling the client code when an error condition occurs.*

**Changes to the code and/or the comments:**

Change function body & add comments:

```
// Throws range_error if index is out of range.
if (index < 0 || index >= size())
    throw std::range_error("Ar: Index out of range");
data_[index] = v;
```

10b. [4 pts]

**Method of dealing with the problem:**

Handling out-of-range values in the function.

**Changes to the code and/or the comments:**

Change function body & add comments:

```
// Does nothing if index is out of range.
if (index < 0 || index >= size())
    return;
data_[index] = v;
```

*In all three parts, I assume the existence of a member function `size`, which returns the size of the array.*

10c. [4 pts]

**Method of dealing with the problem:**

Requiring the client to give an in-range value.

**Changes to the code and/or the comments:**

Change comments:

```
// Pre: 0 <= index < size().
```