# CS213 2024 Tutorial Solutions

CS213/293 UG TAs

2024

## Contents

Tutorial 1 2

# *Tutorial 1*

1. *The following is the code for insertion sort. Compute the exact worst-case running time of the code in terms of n and the cost of doing various machine operations.*

**Algorithm 1:** Insertion Sort Algorithm

```
   Data: Array A of length n
1  for j ← 1 to n − 1 do
2  |  key ← A[j];
3  |  i ← j − 1;
4  |  while i ≥ 0 do
5  |  |  if A[i] > key then
6  |  |  |  A[i + 1] ← A[i];
7  |  |  end
8  |  |  else
9  |  |  |  break;
10 |  |  end
11 |  |  i ← i − 1;
12 |  end
13 |  A[i + 1] ← key;
14 end
```

**Solution:** To compute the worst-case running time, we must decide the code flow leading to the worst case. When the if-condition evaluates to false, the control breaks out of the inner loop. The worst case would have the if-condition never evaluates false. This will happen when the input is in a strictly decreasing order.

Counting the operations for the outer iterator $j$ (which goes from 1 to $n-1$). Consider for each outer loop iteration:

1. Consider the outer loop without the inner loop. Then it has 1 Comparison (Condition in for loop), 1 Increment (1 Assignment + 1 Arithmetic) (Updation in for loop), 3 Assignments (Line 2, 3, 11) and 2 Memory Accesses (A[j] and A[i+1]) and 1 Jump. Hence $(C + 2Ar + 4As + 2M + J)$
2. while loop runs for $j$ times (as $i$ goes from $j-1$ to 0). Without the if-else, the while loop still has to do 1 Comparison (while condition), 1 Decrement (1 Assignment + 1 Arithmetic) ($i \leftarrow i-1$), 1 Jump, every iteration. Hence $(C + As + Ar + J) * j$
3. The if condition is checked in all iterations of the while loop. It involves 1 Comparison (if) and 1 Memory Access (A[i] in the if condition). if block is executed for all iterations of while in the worst case. if block has 2 Memory Accesses and an Assignment and then a Jump happens to outside the if-else statement. Hence $(C + 3M + As + J) * j$

The total cost would then be (terms are for outer loop, inner loop, if and else):

$$\sum_{j=1}^{n-1} ((C + 2Ar + 4As + 2M + J) + (C + As + Ar + J) * j + (C + 3M + As + J) * j)$$

$$= (C + 2Ar + 4As + 2M + J) * (n-1) + (C + As + Ar + J) * (n * (n-1)/2)$$
$$+ (C + 3M + As + J) * (n * (n-1)/2)$$

This means Insertion Sort is $\mathcal{O}(n^2)$.

($C$ is Comparison, $As$ is Assignment, $Ar$ is Arithmetic, $M$ is Memory Access, $J$ is Jump)

Note: Some intermediate operations might have been omitted but they do not affect the overall asymptotic performance of the algorithm. The time taken for comparisons, jumps, arithmetic operations, and memory accesses are assumed to be constants for a given machine and architecture.

2. *What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?*

**Solution:** Note that time complexity is measured as a function of the input size since we aim to determine how the time the algorithm takes scales with the input size.

**Binary Addition**
Assume two numbers A and B. In binary notation, their lengths (number of bits) are m and n. Then the time complexity of binary addition would be $\mathcal{O}(m+n)$. This is because we can start from the right end and add (keeping carry in mind) from right to left. Each bit requires an O(1) computation since there are only 8 combinations (2 each for bit 1, bit 2, and carry). Since the length of a number $N$ in bits is $\log N$, the time complexity is $\mathcal{O}(\log A + \log B) = \mathcal{O}(\log(AB)) = \mathcal{O}(m+n)$.

**Binary Multiplication**
Similar to above, but here the difference would be that each bit of the larger number (assumed to be A without loss of generality) would need to be multiplied by the smaller number, and the result would need to be added. Each bit of the larger number would take $\mathcal{O}(n)$ computations, and then m such numbers would be added. So the time complexity would be $\mathcal{O}(m \times \mathcal{O}(n)) = \mathcal{O}(mn)$[1]. Following the definition above, the time complexity is $\mathcal{O}(\log A \times \log B) = \mathcal{O}(mn)$.

Side note: How do we know if this is the most efficient algorithm? Turns out, this is NOT the most efficient algorithm. The most efficient algorithm (in terms of asymptotic time complexity) has a time complexity of $\mathcal{O}(n \log n)$ where $n$ is the maximum number of bits in the 2 numbers.

**Unary Addition**
The unary addition of A and B is just their concatenation. This means the result would have $A+B$ number of 1's. Iterating over the numbers linearly would give a time complexity of $\mathcal{O}(A+B)$ which is linear in input size.

3. *Given $f(n) = a_0 n^0 + \ldots + a_d n^d$ and $g(n) = b_0 n^0 + \ldots + b_e n^e$ with $d > e$, then show that $f(n) \notin \mathcal{O}(g(n))$*

**Solution:** Let us begin by assuming the proposition is False, ergo, $f(n) \in \mathcal{O}(g(n))$. By definition, then, there exists a constant c such that there exists another constant $n_0$ such that

$$\forall n \geq n_0, f(n) \leq cg(n)$$

. Hence, we have

$$\forall n \geq n_0, a_0 n^0 + \ldots + a_d n^d \leq c b_0 n^0 + \ldots + b_e n^e$$

$$\forall n \geq n_0, \sum_{i=0}^{e} (a_i - c b_i) n^i + a_{i+1} n^{i+1} + \ldots + a_d n^d \leq 0$$

By definition of limit

$$\lim_{n\to\infty} \sum_{i=0}^{e} (a_i - cb_i)n^i + a_{i+1}n^{i+1} + \ldots + a_d n^d \leq 0$$

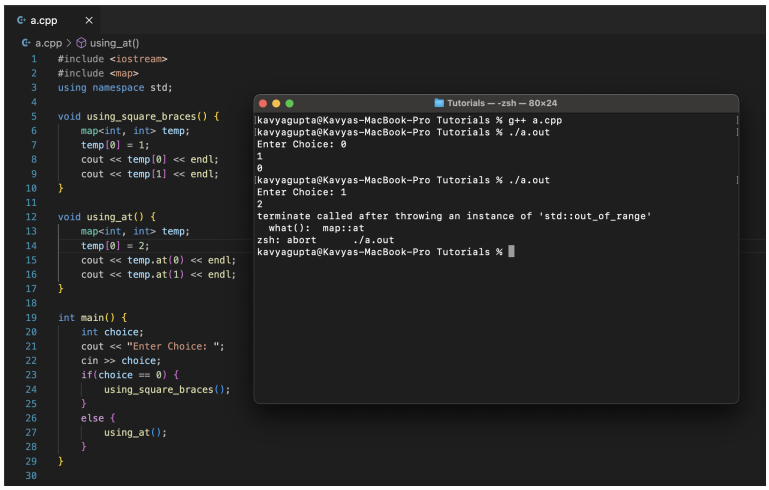
$$\implies a_d \leq 0$$

Assuming $a_d > 0$ (since we are dealing with functions mapping from $\mathbb{N}$ to $\mathbb{N}$), this results in a contradiction; thus, our original proposition is proved.

4. *What is the difference between "at" and "..[..]" accesses in C++ maps?*

**Solution:** Both accesses will first search for the given key in the map but will behave differently when the key is not present. The $\mathrm{at}$ method will throw an exception if the key is not found in the map, but the $[\,]$ operator will insert a new element with the key and a default-initialized value for the mapped type and will return that default value.

Look at the following illustration:-




5. *C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are*

   - $\mathrm{auto\_ptr}$
   - $\mathrm{unique\_ptr}$
   - $\mathrm{shared\_ptr}$
   - $\mathrm{weak\_ptr}$

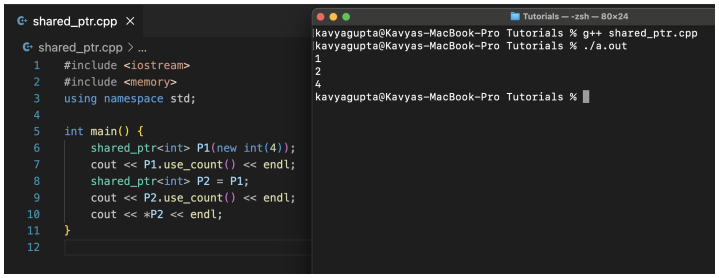
   *Write programs that illustrate the differences among the above smart pointers.*

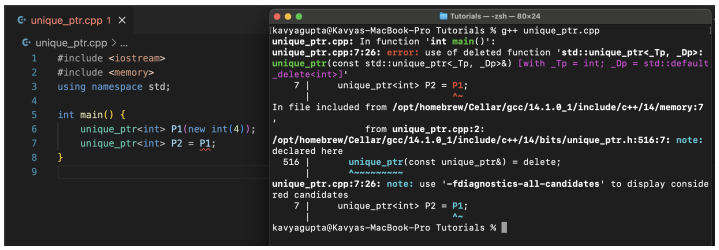
**Solution:** Memory allocated in heap (using $\mathrm{new}$ or $\mathrm{malloc}$) if not de-allocated can lead to memory leaks. Smart pointers in C++ deal with this issue. Broadly, they are classes that store reference counts to the memory they point. When a smart pointer is created, reference count to that memory is increased by one. Whenever a smart pointer (pointing to the same memory) goes out of scope, its destructor is automatically executed, which reduces the reference count of that memory by one and when it hits zero, that memory is deallocated.

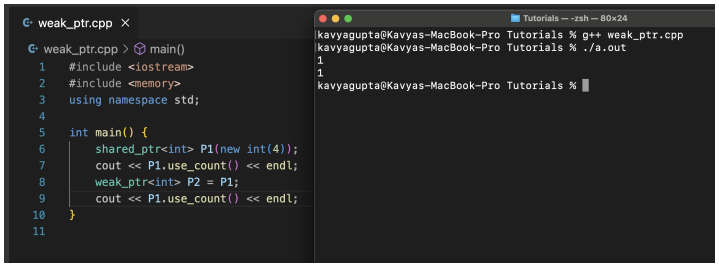
- shared_ptr allows multiple references to a memory location (or an object)
- weak_ptr allows one to refer to an object without having the reference counted
- unique_ptr is like shared_ptr but it does not allow a programmer to have two references to a memory location
- auto_ptr is just the deprecated version of unique_ptr

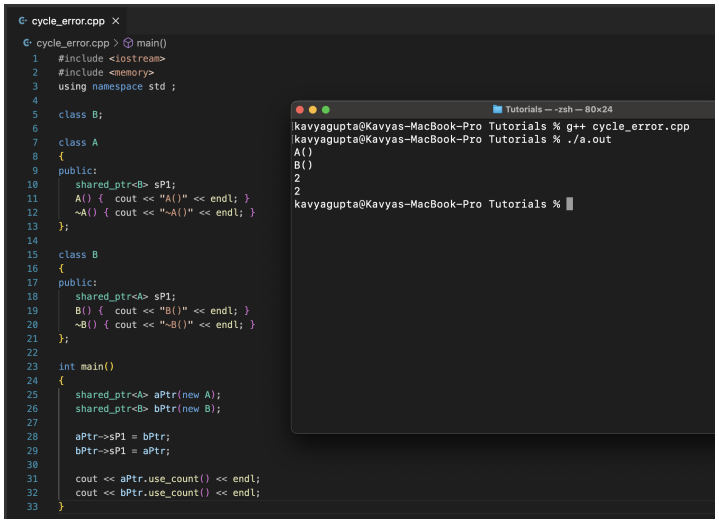
The use for weak_ptr is to avoid the circular dependency created when two or more object pointing to each other using shared_ptr.

You can see the reference count of a shared_ptr using its use_count() function.



6. *Why do the following three writes cause compilation errors in the C++20 compiler?*

```
class Node {
    public :
    Node(): value(0) {}
    const Node& foo(const Node* x) const {
        value = 3; // Not allowed because of ______________
        x[0].value = 4; // Not allowed because of ______________
        return x[0];
    }
    int value;
};

int main () {
    Node x[3], y ;
    auto &z = y.foo(x);
    z.value = 5; // Not allowed because of ______________
}
```

**Solution:** The line 'value = 3;' is not allowed since the method 'foo' is marked **const** at the end. Using **const** after the function signature and its parameters for a class method implies that the class members cannot be changed and the object itself is constant within the bounds of the function. As a result, an error is raised.

The line 'x[0]. value = 4;' raises an error since the parameter 'x' is of type '**const** Node*', meaning that different values can be assigned to the pointer itself, but the subscripts of the pointer cannot be assigned, since it points to constant members of class 'Node'. Likewise, '*x.value = 3;' is equally illegal. Note that 'Node **const** *' also does the same thing, whereas 'Node * **const**' means that the subscripts of the pointer can be assigned since it does not point to constant members of class 'Node' but the pointer itself cannot be assigned to point to a different Node or array of Nodes. '**const** Node* **const**' or 'Node **const** * **const**' imply that the pointer cannot be assigned AND subscripts or dereferences cannot be assigned.

The last line 'z.value = 5;' is illegal since the datatype of 'z', as determined from the method 'foo' is '**const** Node &' NOT 'Node &'. The implication is that 'z' refers to a Node, which is constant and hence cannot be assigned. Note that assignments to class members are as bad as assignments to the class object itself regarding constant objects in C++.