CS213 2024 Tutorial Solutions

CS213/293 UG TAs

2024

Contents

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

$$\forall n \geq n_0, a_0 n^0 + \ldots + a_d n^d \leq cb_0 n^0 + \ldots + b_e n^e$$

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

By definition of limit

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

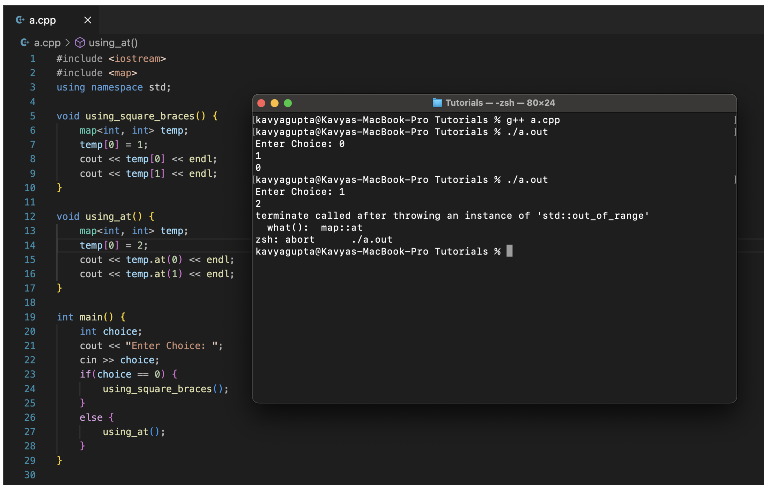
$$\implies a_d \leq 0$$

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 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*

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

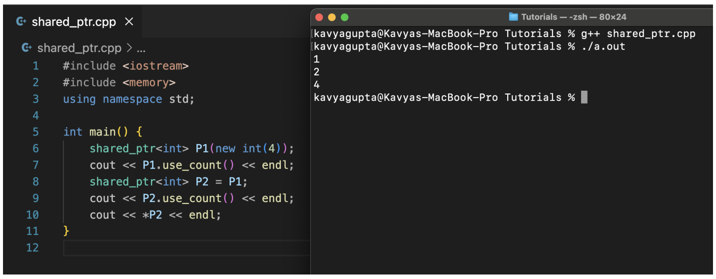
Write programs that illustrate the differences among the above smart pointers.

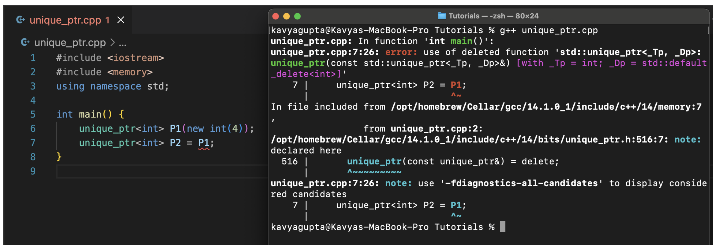
Solution: Memory allocated in heap (using new or malloc) if not de-allocated can lead to memory leaks. Smart pointers in C++ deal with this issue. Broadly, they are classes that store reference counts to the memory they point. When a smart pointer is created, reference count to that memory is increased 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.

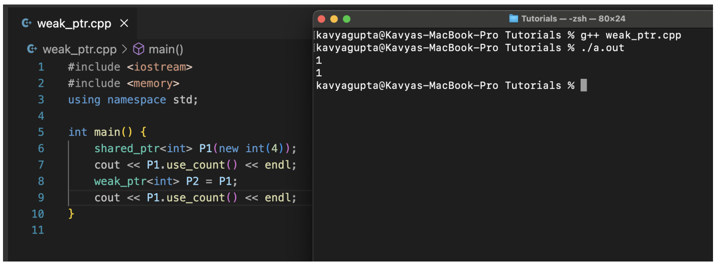
- shared_ptr allows multiple references to a memory location (or an object)
- weak_ptr allows one to refer to an object without having the reference counted
- unique_ptr is like shared_ptr but it does not allow a programmer to have two references to a memory location
- auto_ptr is just the deprecated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

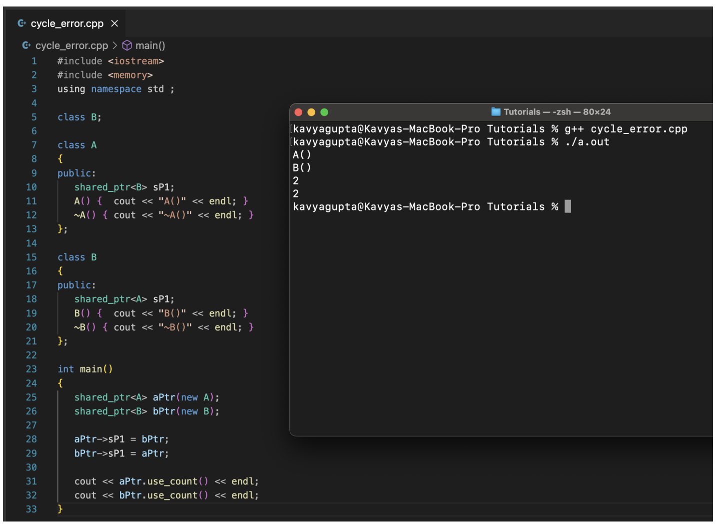
The use for weak_ptr is to avoid the circular dependency created when two or more object pointing to each other using shared_ptr.

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++.

Tutorial 2

---

1. *The span of a stock's price on i^{th} day is the maximum number of consecutive days (up to i^{th} day and including the i^{th} day) the price of the stock has been less than or equal to its price on day i.*

   Example: for the price sequence 2 4 6 5 9 10 of a stack, the span of prices is 1 2 3 1 5 6.

   Give a linear−time algorithm that computes s_i for a given price series.

> **Solution:** The idea here is to find out the latest (most recent) day till the i^{th} day where the price was greater than the price on day i. To do this, we will maintain a stack of indices, and for each index i, we will keep the stack in a state such that the following invariant holds for each day i:
>
> *If the stack is not empty, then the price on the day whose index is at the top of the stack is the most recent price that was strictly greater than the current price.*
>
> With this invariant, the updates to the stack and the stock span follow as:
>
> - we remove the top index in the stack till the price corresponding to the top index becomes strictly greater, or the stack becomes empty
> - if the stack is empty, the current price is the largest, so the span is the number of days so far
> - otherwise the span is the number of days since the index at the top of the stack
>
> (Such a stack is known as a "Monotonic Stack").
>
> The base case is that at day 1 the price is the only encountered price, and hence is the largest, so the span is 1.
>
> The time complexity is indeed $\mathcal{O}(n)$ (n being the number of days/size of array s) because each element is pushed **exactly once** in the stack and popped **atmost once**.
>
> Look at Algorithm 2 for the pseudo-code.

Algorithm 2: Stock Span Algorithm

Data: Array $price$ of length n and an empty stack St

for $i \leftarrow 0$ **to** $n-1$ **do**
    $s[i] \leftarrow i+1$ //If price[i] is $\geq$ all the previous prices, then $s[i]$ will be $i+1$
    while ***not*** $St.isEmpty()$ **do**
        if $price[St.top()] > price[i]$ **then**
            $s[i] \leftarrow i - St.top()$
            break
        end
        $St.pop()$
    end
    $St.push(i)$
end

---

2. *There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radii. Only two operations are allowed:*

   (i) serve the top dosa
   (ii) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack and flip it upside-down and put it back

Design a data structure to represent the tava, input a given tava, and to produce an output in sorted order. What is the time complexity of your algorithm?

This is also related to the train-shunting problem.

Solution: Read the first line of the question with the emphasis being on **stack**. We will use the array representation of a stack for the tava.

Our stack-tava has the following abstraction:

- Initialization and input: Initialize an array S of size n, to represent the stack. Using the given input, fill the array elements with the radii of dosas on the tava in the initial stack order, with the bottom dosa at index 0 and the top at $n-1$. Maintain a variable sz which contains the current size of the stack, and initialize it to n.
- Serving the top dosa: This method is essentially carrying out the "pop" operation on our stack S. Return the top dosa in the stack, $S[sz-1]$, and decrement sz by 1. Clearly, this method takes a constant $\mathcal{O}(1)$ time for every call.
- Flipping the top k dosas: This method is equivalent to reversing the slice of the array S from index $sz-k$ to $sz-1$. This is simple enough: initialize an index i to $sz-k$ and another index j to $sz-1$, swap elements at indices i and j, increment i and decrement j by 1, and repeat the swap and increment/decrement while $j > i$. This method takes $\mathcal{O}(k)$ time for every call.

Note that in this implementation, the serve operation is the only way by which the problem can be reduced in size (number of dosas reduced by 1). The flip operation, if used wisely, can give us access to dosas that can help us reduce the problem size.

With this implementation, the algorithm to serve the dosas in increasing order of the radii can be designed as follows. While the stack is not empty, repeat the following:

- Iterate over the array S and find the index of the minimum element in the array. Let this index be m. This is the position of the dosa having the smallest radius
- Flip over the top $sz-m$ dosas in the stack, using the flipping operation. This brings the smallest dosa from index m to index $sz-1$, id est, the top of the stack
- Serve the top dosa, using the serving operation. This pops the smallest dosa off the stack

It is easy to argue the correctness. The invariant is that we keep serving the minimum radii dosa from the remaining stack. The resulting order has to be sorted.

For computing the time complexity: in a stack of size z, finding the index of the minimum element takes $\mathcal{O}(z)$ time, flipping over the top $k \leq z$ elements takes $\mathcal{O}(k)$ time (and thus $\mathcal{O}(z)$ time) and serving off the top dosa takes $\mathcal{O}(1)$ time. Thus serving the smallest dosa takes $\mathcal{O}(z)$ time overall. This has to be repeated for all dosas, so the stack size z goes from n to 1, decrementing by 1 every time. Therefore the algorithm has a time complexity of $\mathcal{O}(n^2)$.

3. *(a) Do the analysis of performance of exponential growth if the growth factor is three instead of two? Does it give us better or worse performance than doubling policy?*

   (b) Can we do the similar analysis for growth factor 1.5?

Solution: Let us do the analysis for a general growth factor α. Suppose initially N = 1 and there are n = α^i consecutive pushes. So, total cost of expansion is (refer to slides for detailed explanation):

$$(\alpha+1)\cdot(\alpha^0+\alpha^1+\alpha^2...+\alpha^{i-1})$$
$$\frac{\alpha+1}{\alpha-1}\cdot(\alpha^i-1)$$
$$\frac{\alpha+1}{\alpha-1}\cdot(n-1)$$

For α equals to 3 cost of expansion is $2\cdot(n-1)$, it's better than doubling policy. For α equals 1.5 cost of expansion is $5\cdot(n-1)$, which is worse than doubling policy. Trade-off involved is extra memory allocation, maximum extra memory allocated is $\alpha-1$ times the requirement, so with increasing alpha extra memory allocation is increasing.

4. *Give an algorithm to reverse a linked list. You must use only three extra pointers.*

Solution: Let us initialise three pointers - *prev, curr*, and *next*, initialised to null, head and $head -> next$ respectively. If head is null, then it is an empty linked list and we return. Otherwise, we will be iterating over each element in the linked list. In each iteration, perform the following updates.

- $curr -> next =$ prev
- prev = curr
- curr = next
- next = $next -> next$

The loop terminates when $next$ is null, in which case, set head to curr.

5. *Give an algorithm to find the middle element of a singly linked list.*

Solution: The idea is as follows:

- Make 2 pointers middle and end initialized to head of linked list
- At each step, increment middle by one and end by 2
- Continue this process until end reaches the end of linked list
- return the middle element
- return null if the head itself was null (empty linked list)

Note: When there is an odd number of elements, the above algorithm returns the element at the median location. What happens when there is an even number of elements?

6. *Given two stacks S1 and S2 (working in the LIFO method) as black boxes, with the regular methods: "Push", "Pop", and "isEmpty", you need to implement a Queue (specifically: Enqueue and Dequeue working in the FIFO method). Assume there are n Enqueue/Dequeue operations on your queue. The time complexity of a single method Enqueue or Dequeue may be linear in n, however the total time complexity of the n operations should also be $\Theta(n)$*

Solution:

Let us try to formulate a simple implementation of a queue using 2 stacks.

Label the stacks as A and B. We will use stack A to store elements of the queue sequentially, and stack B to implement the dequeue operation.

Enqueue: To insert a new element to the queue, we can just *push* the element to the top of stack A. Here, each *enqueue* operation requires $\mathcal{O}(1)$ time.

Dequeue: To remove an element from the queue in a FIFO manner, we must somehow delete the first element pushed to stack A. Since we cannot access any element of the stack other than the top one (last inserted), we will have to use stack B to implement this.

To get access to the first inserted element, we need to *pop* all the remaining elements from stack A and store them into stack B. As you can see in the figure 1, the order of elements is reversed while moving them to stack B. When we reach the first element of A, we can just *pop* it without then adding it to B.

We must now decide what to do with the elements of stack B. One idea is to simply use stack B for storing the queue in the place of stack A, but this doesn't work since the order of elements would change.

Another possible approach would be to again transfer all the elements saved in stack B to stack A and restore the queue. This would preserve the order of elements and allow new elements to be pushed to A while maintaining correctness. However, think of the worst case time complexity of this approach. For every *dequeue* operation, all the elements have to be shifted twice - once from A to B and then from B to A. This causes each *dequeue* operation to be of $\mathcal{O}(n)$ time complexity and the total time complexity of n operations to be $\mathcal{O}(n^2)$. (think when?)

The blowup in time complexity is caused due to elements being shifted back to A from B on every *dequeue*. Do we really need that? What happens if we just let the elements stay in B?

What happens now if we have 2 successive dequeue operations? Suppose we transferred the elements of queue from stack A to B in the first operation. For the 2nd dequeue operation, we can simply *pop* from stack B. Hence we don't need to re-transfer the elements. Now, if we have n operations, our complexity reduces to $\mathcal{O}(n)$.

To sum it up: Look at Algorithm 3 for the pseudo-code

Algorithm 3: Queue using Two Stacks

```
Function Enqueue(element):
    S1.Push(element)
end
Function Dequeue():
    if S1.isEmpty() and S2.isEmpty() then
        throw("Queue is empty.")
    end
    if S2.isEmpty() then
        while not S1.isEmpty() do
            S2.Push(S1.Pop())
        end
    end
    return S2.Pop()
end
```

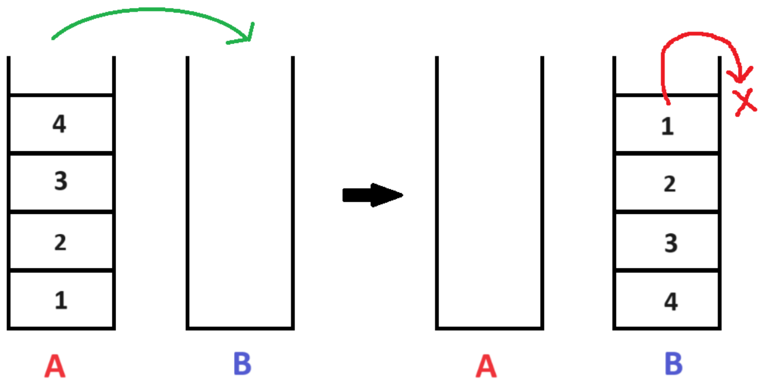


Figure 1: 2 Stacks making a Queue