CS2040S Tutorial 2

Group T40

Week 4

Picture of the Day

Problem 1: Time Complexity Analysis

Problem 1: Time Complexity Analysis

Analyse the following code snippets and find the best asymptotic bound for time complexity of the following functions with respect to n

Problem 1a

```java
public int niceFunction(int n) {
  for (int i = 0; i < n; i++) {
    System.out.println("I am nice!");
  }
  return 42;
}
```

Solution 1a

```java
public int niceFunction(int n) {
  for (int i = 0; i < n; i++) { // O(n)
    System.out.println("I am nice!");
  }
  return 42;
}
```

Time complexity: $\mathcal{O}(n)$

Problem 1b

```java
public int meanFunction(int n) {
  if (n == 0) return 0;
  return 2 * meanFunction(n / 2) + niceFunction(n);
}
```

Solution 1b

```
public int meanFunction(int n) { // T(n)
  if (n == 0) return 0;
  return 2 * meanFunction(n / 2) + niceFunction(n); // 1 + T(n / 2) + O(n)
}
```

Suppose the running time for `meanFunction` is $T(n)$.

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(n/2) + \mathcal{O}(n), & \text{otherwise} \end{cases}$$

Solution 1b (cont.)

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(n/2) + \mathcal{O}(n), & \text{otherwise} \end{cases}$$

We can try to solve the equation by substitution

$$\begin{aligned} T(n) &\le T(n/2) + cn \\ &\le T(n/4) + cn/2 + cn \\ &\dots \\ &\le T(0) + cn\underbrace{(... + 1/2 + 1)}_{\text{how many times?}} \end{aligned}$$

Solution 1b (cont.)

The number of terms in summation = the number of divide-by-2 needed from n to 0 (floored division). For simplicity, we assume we divide it until 1.

Suppose the number of steps is k, we need to solve

$$\frac{n}{2^k} \leq 1$$

Multiply by $2^k > 0$ and take log on both sides:

$$\lg n \leq k$$

So we need at least $\lg n$ terms.

Solution 1b (cont.)

$$
\begin{aligned}
T(n) &\leq T(0) + cn \sum_{k=0}^{\lg n - 1} \frac{1}{2^k} \\
&\leq 1 + cn \cdot \frac{1((1/2)^{\lg n} - 1)}{1/2 - 1} \\
&\leq 1 + cn \cdot \frac{2^{-(\lg n)} - 1}{-1/2} \\
&\leq 1 + cn \cdot 2(1 - n^{-1}) \\
&\leq 1 + 2cn - 2c \\
&\leq 2cn + 1 - 2c \\
&= \mathcal{O}(n)
\end{aligned}
$$

Solution 1b (cont.)

- Less tedious approach: draw recursion tree

Problem 1c

```java
public int strangerFunction(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      System.out.println("Execute order?");
    }
  }

  return 66;
}
```

Solution 1c

```java
public int strangerFunction(int n) {
  for (int i = 0; i < n; i++) { // O(n)
    for (int j = 0; j < i; j++) { // i instructions every loop
      System.out.println("Execute order?");
    }
  }

  return 66;
}
```

Simply $\sum_{i=1}^{n} i = n(n+1)/2 = \mathcal{O}(n^2)$

Problem 1d

```
public int suspiciousFunction(int n) {
  if (n == 0) return 2040;

  int a = suspiciousFunction(n / 2);
  int b = suspiciousFunction(n / 2);

  return a + b + niceFunction(n);
}
```

Solution 1d

```java
public int suspiciousFunction(int n) { // T(n)
  if (n == 0) return 2040;

  int a = suspiciousFunction(n / 2); // T(n/2)
  int b = suspiciousFunction(n / 2); // T(n/2)

  return a + b + niceFunction(n); // 2 + O(n) = O(n)
}
```

Solve for $T(n) = 2T(n/2) + \mathcal{O}(n)$

$T(n) = \mathcal{O}(n \lg n)$

Details: see tutorial solution

Problem 1e

```java
public int badFunction(int n) {
  if (n <= 0) return 2040;
  if (n == 1) return 2040;

  return badFunction(n - 1) + badFunction(n - 2) + 0;
}
```

Solution 1e

```
public int badFunction(int n) { // T(n)
  if (n <= 0) return 2040;
  if (n == 1) return 2040;

  return badFunction(n - 1) + badFunction(n - 2) + 0; // T(n) + T(n - 1) + 1
}
```

Solve for $T(n) = T(n-1) + T(n-2) + 1$

Result: $\mathcal{O}(\phi^n)$

Note: Not the same as Fibonacci sequence! Details on tutorial solution

Problem 1f

```
public int metalGearFunction(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 1; j < i; j *= 2) {
      System.out.println("!");
    }
  }

  return 0;
}
```

Solution 1f

```
public int metalGearFunction(int n) {
  for (int i = 0; i < n; i++) { // O(n)
    for (int j = 1; j < i; j *= 2) { // O(lg i)
      System.out.println("!");
    }
  }

  return 0;
}
```

Each loop takes ~$\lg(i+1)$ steps.

Time complexity: $\sum_{i=1}^{n} \lg i = \lg n! \leq \lg n^n = \mathcal{O}(n \lg n)$

Problem 1g

```java
public String simpleFunction(int n) {
  String s = "";
  for (int i = 0; i < n; i++) {
    s += "?";
  }

  return s;
}
```

Solution 1g

```
public String simpleFunction(int n) {
  String s = "";
  for (int i = 0; i < n; i++) { // O(n)
    s += "?"; // O(len(s))
  }

  return s;
}
```

Note that `+=` operation on `String` s takes $\mathcal{O}(len(s))$ time. Hence, it is $\mathcal{O}(n^2)$

For faster appending, you may want to look at `StringBuilder` that appends in $\mathcal{O}(1)$ time

Reflection on Problem 1

- Analysis of recursive function
  - Assume the time need to execute input of size n is $T(n)$
  - Form recursive formula and solve it
- Seem harmless, but costly :(
  - If memoization is possible, do it!
- Recursion tree to gain intuition
- Know your library

Problem 2: Sorting Review

Problem 2a

How would you implement insertion sort recursively? Analyse the time complexity by formulating a recurrence relation.

Solution 2a

Let `insertionSort(A, n)` be an algorithm that sorts first `n` elements of array `A`.

- What is the base case?
  - When `n` is 0, no need to sort
- What is the recursive property?
  - Take `n`-th element, sort the rest, insert it

Solution 2a (cont.)

```java
public void insertionSort(int[] A, int n) {
    if (n == 0) { // Base case
        return;
    }

    insertionSort(A, n - 1); // Recurse!

    int cur = n;

    while (cur > 0 && A[cur] < A[cur - 1]) {
        swap(A[cur], A[cur - 1]);
        cur--;
    }
}
```

Solution 2a (cont.)

```java
public void insertionSort(int[] A, int n) { // T(n)
    if (n == 0) {
        return; // O(1)
    }

    insertionSort(A, n - 1); // T(n - 1)

    int cur = n;

    while (cur > 0 && A[cur] < A[cur - 1]) { // O(n)
        swap(A[cur], A[cur - 1]);
        cur--;
    }
}
```

Solution 2a (cont.)

Suppose that `insertionSort(A, n)` runs in $T(n)$.
Then we have $T(n) = T(n-1) + O(n)$.
Solving $T(n)$ gives $T(n) = O(n^2)$

Problem 2b

Consider an array of pairs (a,b). Your goal is to sort them by a in ascending order first, and then by b in ascending order. For example, [(2, 1), (1, 4), (1, 3)] should be sorted into [(1,3), (1,4), (2,1)].

You are given 2 sorting functions, which are a MergeSort and a SelectionSort. You can use each sort at most once. How would you sort the pairs? Assume you can only sort by one field at a time.

Solution 2b

- Insertion sort with key = b and value = a. After that, merge sort with key = a and value = b.
- After the insertion sort, value of b will be nondecreasing.
- Merge sort is stable, hence for the same value a, the corresponding values b is nondecreasing.

Problem 2c

Implement Merge Sort *iteratively*.

Solution 2c

- Merge every chunk of k elements, where k iterates from 2, 4, 8, and so on.
- Similar as recursive merge sort.
- Since k is always two times larger than its previous value, it will iterate about $\lg n$ times.
- In each iteration, we run through the whole array, i.e. n
- Total time complexity: $\mathcal{O}(n \lg n)$
- Requires $\mathcal{O}(n)$ additional space

Problem 3: Queues and Stacks Review

Recall the Stack and Queue Abstract Data Types (ADTs) that we have seen in CS1101S. Just a quick recap, a Stack is a "LIFO" (Last In First Out) collection of elements that supports the following operations:

- `push`
- `pop`
- `peek`

Problem 3: Queues and Stacks Review (Cont.)

And a Queue is a "FIFO" (First In First Out) collection of elements that supports these operations:

- `enqueue`
- `dequeue`
- `peek`

Problem 3a

Implement Stack and Queue with fixed-size array in Java. Assume that the number of items never exceed the size of array.

Problem 3c

What sorts of problem handling do we need? (Applies for 3a and 3b)

Solution 3a & 3c

Stack

- Use a pointer, let it be `tail`. Initialize with `0`
- When `push`, add the element to where `tail` points to. Increase `tail` by 1
- When `pop`, decrease `tail` by 1
- When `peek`, look through element at `tail - 1`
- Error handling:
  - cannot `pop` and `peek` -- `tail == 0`
  - cannot `push` when it is full -- `tail == array.length`

Solution 3a & 3c

Queue

- Use two pointers, let it be `head` and `tail`
- `head` will be the pointer to to-be-popped element
- `tail` will be the pointer to to-be-inserted element
- Increase `head` when `dequeue`
- Increase `tail` when `enqueue`
- Circular array (e.g. `head = (head + 1) % A.length`)
- Error handling:
    - cannot `dequeue` and `peek` -- `head == tail && A[tail] == null`
    - cannot `enqueue` when it is full -- `head == tail && A[tail] != null`

Problem 3b

Implement Deque (double-ended queue) with fixed-size array in Java, which have the following operations:

- `enqueue_front`
- `dequeue_front`
- `enqueue_back`
- `dequeue_back`

Assume that the number of items never exceed the size of array.

Solution 3b

- Similar to the idea as before, we have `head` and `tail`.
- Here, `head = 0` and `tail = A.length - 1`. Hence, we have the invariant that when we run from `head` to `tail` (in left direction) circularly (excluding both `head` and `tail`), those are elements inside your deque.
- `enqueue_front` and `dequeue_front` should increase and decrease `head` by 1
- `enqueue_back` and `dequeue_back` should decrease and increase `tail` by 1
- Empty when `(tail + 1) % A.length == head` and both are empty
- Full when `(tail + 1) % A.length == head` and both are filled

Problem 3d

A set of parentheses is said to be balanced as long as every opening parenthesis `"("` is closed by a closing parenthesis `")"`. So for example, the strings `"()()"` and `"(())"` are balanced but the strings `")(())("` and `"(("` are not. Using a stack, determine whether a string of parentheses are balanced.

Solution 3d

- Push when encounter open brackets
- Pop when encounter closing brackets
- If stack is empty when popping, then it's not balanced
- If at the end it is not empty, then it's not balanced
- Invariant: When we success fully pop a bracket, we found a pair of balanced bracket

Problem 4: Stac and Cue

Problem 4: Stac and Cue

Abridged Problem Statement

Given N houses, each of which has height of H_i. Find the number of houses such that there exist a house to its left and right such that it has higher height than itself.

Leetcode 42. Trapping Rain Water

Solution 4

- Keep track with Stack that store nonincreasing height
- If current house is higher than the previous one, pop from stack -> the house is flooded
- Anything odd?
    - You cannot flood if the left-side is empty!

Test Cases

- `[5, 4, 3, 0, 5, 1, 6]` -- `[F, T, T, T, F, T, F]`
- `[5, 4, 5, 1, 6, 1, 6]` -- `[F, T, T, F, F, T, F]`

Pseudocode

```
int findFlooded(int n, int[] heights) {
    Stack<Integer> stack;

    int floodedCount = 0;
    int maxHeight = -1;

    for (int height: heights) {
        int popCount = 0;

        while (!stack.empty() && stack.top() < height) {
            stack.pop();
            popCount += 1;
        }

        if (maxHeight >= height) {
            floodedCount += popCount;
        }

        stack.push(height);
        maxHeight = Math.max(maxHeight, height)
    }

    return floodedCount;
}
```

Runtime Analysis

Each height is at pushed at least once and popped at most once. Each operation takes $\mathcal{O}(1)$ time. Hence it takes $\mathcal{O}(n)$.

Problem 5: Sorting with Queues (Optional)

Sort a queue using *another queue* with $O(1)$ additional space

Solution 5

- Use the same idea as problem 1c, i.e. iterative merge sort
- When we want to sort a chunk of size k
  - Dequeue $k/2$ elements and put it to the other queue (Call it Q_2)
  - Merging phase takes place in enqueuing in the original queue (Call it Q_1)
- Invariant (after dequeue $k/2$ elements):
  - The first $k/2$ elements in Q_1 are sorted
  - The $k/2$ elements in Q_2 are also sorted
  - After merge, the last inserted k elements in Q_1 are sorted