Problem 1. Consider the following egg dropping puzzle.

Suppose that you have $n$ eggs and an $s$ floor building. You want to know what is the highest floor you can drop an egg from without it breaking. You may assume that

- an egg that survives a fall is undamaged and may be reused,
- a broken egg is discarded,
- each egg is equally durable (until broken),
- if an egg breaks on floor $a$, it will break on any floor $b > a$,
- if an egg survives a fall from floor $a$, it will survive a fall from any floor $b < a$, and
- the egg may break on every floor or it may not break on any floor.

We describe three approaches to determining which floor the egg first breaks on below.

- You could test every floor from the lowest to the highest, though this will take the maximum number of trials (without repeats). This is, however, the minimum number of trials required if provided with only one egg.

- If you have 2 eggs, you could test the middle floor and perform a binary search upwards until it breaks. Once it does, you can test the floors between where it broke and when you last verified it did not break (floor 0 if it breaks on the first test).

- Alternatively, you could perform a proper binary search and hope you don’t run out of eggs early. This approach would not always succeed, so it fails the worst-case scenario requirement.

To develop an algorithm to optimize this, define the following function $N$.

$N(e, f) :=$ The minimum number of trails given $e$ eggs to test $f$ consecutive floors in the worst-case scenario.

To clarify, the worst-case scenario means you must determine which floor the egg first breaks when dropped from without fail regardless of the outcome. This means that when you test floor $k$, you would assume the worse outcome between the egg breaking and the egg not breaking occurs for your strategy.

a) Describe a dynamic programming approach to the problem (in English).
b) Give a recursive definition for $N(e, f)$ using mathematical notation. Don’t forget to build your base cases into it.

c) Give a dynamic programming algorithm to determine $N(e, f)$ for any $e > 0$ and $f > 0$.

d) What is the runtime of your algorithm?

**Problem 2.** A stack has two operations it supports in $O(1)$ time: push and pop. A queue has two supported operations as well:

- enqueue, wherein a value is pushed onto the end of the queue, and
- dequeue, wherein a value is popped from the front of the queue.

These two data structures are similar enough that we can use one to implement the other.

a) Describe (in English) how to implement a queue using two stacks. Your implementation should be able to support a constant amortized time enqueue and dequeue.

b) Show that your double stack queue implementation has $O(1)$ amortized time enqueue and dequeue operations using the aggregate method.

c) Show that your double stack queue implementation has $O(1)$ amortized time enqueue and dequeue operations using the accounting method.

d) Show that your double stack queue implementation has $O(1)$ amortized time enqueue and dequeue operations using the potential method.

**Problem 3.** Amortized time is not exclusively used to analyze data structures. An online algorithm is an algorithm that receives its input in serial. Otherwise put, it does not have its entire input immediately available and must be ready to receive more indefinitely.

An example of an online algorithm is insertion sort. It sorts by considering one new element at a time of an increasingly large list. Providing it with more data merely requires it to carry on with business as usual. We give an online version of the algorithm below.

**Algorithm 1:** OnlineInsertion($A, B$)

**Input:** A sorted doubly linked list $A$ a doubly linked list $B$

**Output:** A sorted doubly linked list $C$ containing the elements of $A$ and $B$

```plaintext
foreach $b \in B$ do
  Place $b$ at the end of $A$
  foreach $a \in A$ from back to front do
    if $b < a$ then
      Swap $b$ and $a$
    else
      Terminate the $a$ loop early
  return $A$
```

a) Determine the worst-case runtime of OnlineInsertion in terms of $A$ and $B$. The bound you give should be asymptotically tight (i.e. if it runs in $n^3 + 2n$ time, you should state $O(n^3)$, not $O(n!)$).
You may find the following identity for arithmetic sums useful.

\[ f + (f + d) + (f + 2d) + \ldots + (f + (n-1)d) = \frac{n(f + l)}{2} \]

Here \( n \) is the number of terms, \( f \) is the first term, and \( l = f + (n-1)d \) is the last term of the series.

b) Determine the asymptotically tight worst-case amortized time of OnlineInsertion over a sequence of \( n \) operations starting from \( A = \emptyset \) and adding \( B_1, \ldots, B_n \), where for each \( 1 \leq i \leq n, |B_i| = 1 \). This corresponds to the case when we restrict ourselves to adding one new input at a time.

You answer should be given in terms of \( A \), the \( B_i \)'s, and \( n \). You may or may not need every variable.

c) Determine the asymptotically tight worst-case amortized time of OnlineInsertion over a sequence of \( n \) operations starting from \( A = \emptyset \) and adding \( B_1, \ldots, B_n \).

You answer should be given in terms of \( A \), the \( B_i \)'s, and \( n \). You may or may not need every variable. You may also wish to introduce auxiliary variables expressing some useful combination of these variables.

(Hint: every \( |B_i| \leq |B_m| \) for some \( 1 \leq m \leq n \); equality is achieved in the worst-case)

d) Using your answers to the two previous parts, argue for one of the following:

- it is better to provide only one input at a time,
- it is better to batch inputs together into larger sets fewer in number, or
- neither is better than the other.