ADVANCED ANALYSIS: 
AMORTIZATION AND RECURRANCE RELATIONS

• amortized time complexity
• accounting method
• Java vectors
• Recurrence Relations
Amortized Running Time

• Amortized running time considers interactions between operations by studying the total running time of a series of operations.

• **Example:** a **Clearable Stack**: supports the usual stack methods plus operation

  ```
  clearStack(): Empty the stack by removing all its elements
  Input: None; Output: None
  ```

  *clearStack* takes $O(n)$ time in the worst case

• **Proposition:** A series of $n$ operations on an initially empty clearable stack implemented with an array takes overall $O(n)$ time

• **Justification:**
  - Let $M_0..., M_{n-1}$ be the series of operations and $M_{i0}..., M_{ik-1}$ be the $k$-th *clearStack* operations in the series
  - We define $i_{-1} = -1$
  - The run time of operation $M_{ij}$ is $O(i_j - i_{j-1})$ since at most $i_j - i_{j-1}$ elements can be on the stack
Amortized Running Time (cont)

- Thus the running time of all the clearStack operations is

\[ O\left(\sum_{j=0}^{k-1} (i_j - i_{j-1})\right) \]

which is a telescoping sum.
- So the run time is \( O(n) \)

• **Definition:** the *amortized running time* of an operation within a series of operations is the worst-case running time of the entire series of operations divided by the number of operations.
Accounting Method

• The **accounting method** performs an amortization analysis with a system of credits and debits.

• Let’s view the computer as a vending machine that requires one cyber-dollar for a constant amount of computing time.

• An operation consists of a series of constant-time **primitive operations** that cost one cyber-dollar each.

• We will overcharge an operation that executes few primitives and use the profit to pay for operations that execute many primitives.

• We will need to set up a scheme for charging operations. This is known as the amortization scheme.
Amortization Scheme Example for a ClearableStack

• Assume one cyber-dollar is enough to pay for the push, pop, top, size, or isEmpty and for the time spent by the clearStack to dereference one element.

• We will charge 2 cyber-dollars though.

• So we undercharge clearStack but overcharge the other operations. When a clearStack operation is executed, the cyber-dollars stored in the stack are used to pay for derefencing the items.
Java Vectors

- The `java.util.Vector` class provides a convenient expandable data type in Java.

- A vector is a wrapper around an array that holds a variable called `capacityIncrement`. When the user inserts the \( n+1 \)\textsuperscript{st} element into a vector of size \( n \), the size of the array is increased by `capacityIncrement` if it is positive, or doubled if `capacityIncrement` is 0.

- Consider the case of `capacityIncrement` = 0:
  - Copying an array into a larger array takes \( O(n) \) time, but this only happens for \( \log(n) \) insertions.
  - Each insertion has \( O(1) \) amortized running time
Java Vectors (contd.)

- **Justification:**
  The array doubles in size with the insertion of every $2^i$th element ($1^{st}$, $2^{nd}$, $4^{th}$, etc.)

  - **Worst case:** we insert exactly $n = 2^i$ elements, so the last operation involves copying the entire array over again.
  
  We have $n$ insertions, and $n$ elements copied in the last insertion. We also have $i-1$ previous expansions of the array, which perform the following number of element-copy operations:

  \[
  \sum_{k=1}^{i-1} 2^k = 2^i - 1 = n - 1
  \]

  - The overall time complexity is proportional to $3n-1$, which is $O(n)$

- **But what if the `capacityIncrement` is, say, 3?**
  Do we still have the same amortization?

  - **No!** Copying an array into a larger array is $O(n)$, but this happens once every $n/capacityIncrement$ insertions.
  
  - Each insertion is amortized to $O(n)$
Java Vectors (contd.)

• **Justification:** \( c = \text{capacityIncrement} \)
  
  Let us assume that the original vector size is 0. The vector increases in size by the insertion of every \((ic)^{th}\) element \((1^{st}, c^{th}, 2c^{th}, \text{etc.})\)

  **Worst case:** we insert exactly \( n = ic \) elements, so the last operation involves copying the entire array.
  
  We have \( n \) insertions, and \( n \) elements copied on the last insertion.
  
  We also have \( i-1 \) other array copies, for a total of:

  \[
  \sum_{k=0}^{i-1} ck = c \sum_{k=0}^{i-1} k = c \frac{i(i - 1)}{2}
  \]

  previous element copies.

• The overall time complexity is proportional to 
  \( n(n-1/(2c)) \), which is \( O(n^2) \)
Recurrence Relations
The Pizza Slicing Problem

How many pieces of pizza can you get with N straight cuts?

1 cut 2 slices
2 cuts 4 slices
3 cuts 6 slices

... N cuts 2N slices

But ... who said you should cut through the center every time?
A Better Slicing Method ...

When cutting, intersect all previous cuts and avoid previous intersection points!

4 cuts
11 slices!!

5 cuts
16 slices!!
So ... How Many Pieces?

The $N$-th cut creates $N$ new pieces.
Hence, the total number of pieces given by $N$ cuts, denoted $P(N)$, is given by the following two rules:

- $P(1) = 2$
- $P(N) = P(N-1) + N$

Recursive definition of $P(N)$!
Recurrence Relations

- The pizza-cutting problem is an example of recurrence relation, where a function $f(N)$ is recursively defined.

  (Base Case) \[ f(1) = 2 \]

  (Recursive Case) \[ f(N) = f(N-1) + N \quad \text{for } N \geq 2 \]

- The standard method for solving recurrence relations, called “unfolding”, makes repeated substitutions applying the recursive rule until the base case is reached.

\[
\begin{align*}
 f(N) &= f(N-1) + N \\
 f(N) &= f(N-2) + (N-1) + N \\
 f(N) &= f(N-3) + (N-2) + (N-1) + N \\
 & \quad \vdots \\
 f(N) &= f(N-i) + (N-i+1) + \ldots + (N-1) + N \\
\end{align*}
\]

The base case is reached when $i = N - 1$

\[
\begin{align*}
 f(N) &= 2 + 2 + 3 + \ldots + (N-2) + (N-1) + N \\
 f(N) &= N \frac{(N+1)}{2} + 1 = O(N^2)
\end{align*}
\]
Towers of Hanoi

Goal: transfer all $N$ disks from peg A to peg C

Rules:
- move one disk at a time
- never place larger disk above smaller one

Recursive solution:
- transfer $N - 1$ disks from A to B
- move largest disk from A to C
- transfer $N - 1$ disks from B to C

Total number of moves:
- $T(N) = 2 \cdot T(N - 1) + 1$
Solution of the Recurrence for Towers of Hanoi

Recurrence relation:

- \( T(N) = 2 \cdot T(N - 1) + 1 \)
- \( T(1) = 1 \)

Solution by unfolding:

\[
T(N) = 2 \cdot (2 \cdot T(N - 2) + 1) + 1 = 4 \cdot T(N - 2) + 2 + 1 = 4 \cdot (2 \cdot T(N - 3) + 1) + 2 + 1 = 8 \cdot T(N - 3) + 4 + 2 + 1 = \ldots
\]

\[
= 2^i \cdot T(N - i) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0
\]

the expansion stops when \( i = N - 1 \)

\[
T(N) = 2^{N-1} + 2^{N-2} + 2^{N-3} + \ldots + 2^1 + 2^0
\]

This is a geometric sum, so that we have:

\[
T(N) = 2^N - 1 = O(2^N)
\]
Another Recurrence

\[ T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{for } N \geq 2 \]

\[ T(1) = 1 \]

\[
T(N) = 2 \left( 2T\left(\frac{N}{4}\right) + \frac{N}{2} \right) + N
= 4T\left(\frac{N}{4}\right) + 2N
= 4 \left( 2T\left(\frac{N}{8}\right) + \frac{N}{4} \right) + 2N
= 8T\left(\frac{N}{8}\right) + 3N
= \ldots
= 2^i T\left(\frac{N}{2^i}\right) + iN
\]

The expansion stops for \( i = \log N \), so that

\[ T(N) = N + N \log N \]
Solving Recurrences by “Guess and Prove”

\[ T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{for} \quad N \geq 2 \]
\[ T(1) = 1 \]

Step 1: Take a wild guess that

\[ T(N) = N + N \log N \]

Step 2: Prove it by induction:

**Basis**

\[ T(1) = 1 + \log 1 = 1 \]

**Inductive Step**

\[ T(N) = 2T\left(\frac{N}{2}\right) + N = 2\left(\frac{N}{2} + \frac{N}{2} \log \frac{N}{2}\right) + N \]

\[ T(N) = N + N(\log N - 1) + N = N + N \log N \]
A More Difficult Example

\[ T(N) = 2T(\sqrt{N}) + 1 \quad T(2) = 0 \]

\[
2T(N^{1/2}) + 1 \\
2(2T(N^{1/4}) + 1) + 1 \\
4T(N^{1/4}) + 1 + 2 \\
8T(N^{1/8}) + 1 + 2 + 4 \\
\ldots \\
2^i T\left(\frac{1}{N^{2^i}}\right) + 2^0 + 2^1 + \ldots + 2^i - 1
\]

The expansion stops for \( N^{2^i} = 2 \)
i.e., \( i = \log\log N \)

\[ T(N) = 2^0 + 2^1 + \ldots + 2^{\log\log N - 1} = \log N. \]
Proofs by Induction

We want to show that property $P$ is true for all integers $n \geq n_0$

**Basis:**
prove that $P$ is true for $n_0$.

**Inductive Step:**
prove that

if $P$ is true for all $k$ such that $n_0 \leq k \leq n - 1$

then $P$ is also true for $n$
An Example of Proof by Induction

\[ S(n) = \sum_{i=1}^{n} i = n\frac{(n+1)}{2} \quad \text{for } n \geq 1 \]

**Basis:**

\[ S(1) = 1\frac{(1+1)}{2} = 1 \quad \text{Easy, Right?} \]

**Inductive Step:**

Assume \( S(k) = k\frac{(k+1)}{2} \) for \( 1 \leq k \leq n-1 \)

\[ S(n) = \sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n = S(n-1) + n \]

\[ = (n-1)\frac{(n-1+1)}{2} + n = \frac{(n^2 - n + 2n)}{2} \]

\[ = n\frac{(n+1)}{2} \]