Lab 8
Strategy 3 for Memory Allocation
- How amortization works! -
How amortization works!

- The table shown on the last slide illustrates the effect of using the third strategy (when allocating memory for our array) on the time efficiency of the append function.

- Remember, in the third strategy, we allocate memory for our array by calling `realloc( 2 * reserved )`, hence doubling the size of our array every time our array runs out of space.
This table explains why we say that the time efficiency of Strategy 3 memory allocation is **amortized O(1)**

- Column 1 (without a column heading) depicts our array
- Column 2 lists the size of array (often expressed as \( n \) in such complexity analysis) stored in variable `reserved`
  - The size of array starts at 10 and doubles on each row
- Column 3 computes the time efficiency of appending one more point to array
  - For example, when the size of array is 10, i.e., when the array has already 10 cells (have a look at the diagram in the first column), appending each of the first 10 points to the array takes \( O(1) \) to do (i.e., constant time)
  - The 11\(^{th} \) point (on the next row) will, however, take \( O(n) \) time since a call to `realloc` will need to be made and the worst case scenario of `realloc` is to allocate a new memory space of 20 cells for the whole array (`realloc` may not be able to simply add 10 cells to the already allocated 10 cells in the array) and copy all 20 points to this new memory location -> \( O(n) \)
  - Once 20 cells have been allocated and the 11\(^{th} \) point (point 10) appended to array, each of the other 9 points (11\(^{th} \) to 19\(^{th} \)) is appended to array in \( O(1) \) time
- Column 4 computes the ratio of the number of points appended in \( O(1) \) over the total number of points appended to array
  - As you can see this ratio, after moving slightly away from 1 (0.95 for \( n = 20 \), 0.95 for \( n = 40 \)), starts raising towards 1 again (0.9625 for \( n = 80 \), 0.975 for \( n = 160 \)) as the number of points appended to the array increases
- Conclusion: as size of array increases to infinity, the number of append's done in \( O(1) \) increases (demonstrated by the ratio moving towards 1)
## Lab 8 Demo

Amortization: How come 3rd strategy is O(1)?

<table>
<thead>
<tr>
<th>n (reserved)</th>
<th>time efficiency of appending 1 pt.</th>
<th>( \frac{10 \cdot O(1)}{10 \cdot O(1)} = 1.0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>( \frac{10 \cdot O(1)}{20} ) = 0.95</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>( \frac{19 \cdot O(1)}{20} ) = 0.95</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>( \frac{38 \cdot O(1)}{40} ) = 0.95</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>( \frac{77 \cdot O(1)}{80} ) = 0.9625</td>
</tr>
<tr>
<td>160</td>
<td></td>
<td>( \frac{156 \cdot O(1)}{160} ) = 0.975</td>
</tr>
</tbody>
</table>

As \( n \to \infty \), the number of operations increases and the ratio \( \frac{n}{n+1} \) approaches 1.

The 3rd strategy is efficient because the cost of operations is amortized over time.

### 3rd strategy

- The strategy keeps the array size \( O(n) \)
- As \( n \) increases, the ratio \( \frac{n}{n+1} \) approaches 1, indicating efficiency.

The strategy is called the "sliding window" and is useful in scenarios where the size of the array needs to be adjusted dynamically.