Two-Dimensional Arrays
Two-Dimensional Arrays

- A *one-dimensional array* stores a list of elements.
- A *two-dimensional array* can be thought of as a table of elements, with rows and columns.
Two-Dimensional Arrays

• In Java, a two-dimensional array is an *array of arrays*

```
int[][] matrix = new int[12][50];
```

• A two-dimensional array is declared by specifying the size of each dimension separately:

```
int[][] matrix = new int[12][50];
```
Two-Dimensional Arrays

- Declaration:
  
  \[ \text{int}[][] \ \text{matrix} = \text{new} \ \text{int}[12][50]; \]

- Referencing a single element:
  
  \[ \text{value} = \text{matrix}[3][6]; \]

- The array stored in one row can be specified using one index

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>int[][]</td>
<td>2D array of integers, or array of integer arrays</td>
</tr>
<tr>
<td>matrix[5]</td>
<td>int[]</td>
<td>array of integers</td>
</tr>
<tr>
<td>matrix[5][12]</td>
<td>int</td>
<td>integer</td>
</tr>
</tbody>
</table>
Looping Through a 2D Array

```java
int M = 10, N = 5;
double[][] a = new double[M][N];
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        a[i][j] = 0;
```

```java
int M = 10, N = 5;
double[][] a = new double[M][N];
for (int i = 0; i < a.length; i++)
    for (int j = 0; j < a[i].length; j++)
        a[i][j] = 0;
```
Ragged 2D Arrays

```java
int M = 9;
double[][] a = new double[M][];
for (int i = 0; i < M; i++) {
    a[i] = new double[M-i];
    for (int j = 0; j < a[i].length; j++)
        a[i][j] = 0.0;
}
```
Ragged 2D Arrays

```c
int scores[][] = { {44, 55, 66, 77}, 
                  {36}, 
                  {87, 97}, 
                  {68, 78, 88}  };
```
Asymptotic Analysis
Computational Complexity

- How many resources will it take to solve a problem of a given size?
  - time
  - space

- Expressed in terms of problem size (beyond some minimum)
  - how do requirements grow as size grows?

- Problem size (typically called n)
  - number of elements to be handled
  - size of thing to be operated on
The Goal of Asymptotic Analysis

- How to analyze the running time (aka computational complexity) of an algorithm in a theoretical model

- Using a theoretical model allows us to ignore:
  - Which computer are we using?
  - How good is our compiler at optimization?

- We define the running time of an algorithm with input size n as $T(n)$ and examine the rate of growth of $T(n)$ as $n$ grows larger and larger and larger.
Growth Functions

- **Constant**
  \[ T(n) = c \]
  
  ex: getting array element at known location
  any simple Java statement (e.g. assignment)

- **Linear**
  \[ T(n) = cn \text{ [+ possible lower order terms]} \]
  
  ex: finding particular element in array of size n
  (i.e. sequential search)
  trying on all of your n shirts
Growth Functions (cont.)

- Quadratic
  \[ T(n) = cn^2 \ [ + \text{possible lower order terms}] \]
  ex: sorting all the elements in an array (using insertion sort)
    trying all your n shirts with all your n pants

- Polynomial
  \[ T(n) = cn^k \ [ + \text{possible lower order terms}] \]
  ex: finding the largest element of a k-dimensional array
    looking for maximum substrings in array
Growth Functions (cont.)

- Exponential
  \[ T(n) = c^n \] [+ possible lower order terms]
  
ex: constructing all possible orders of array elements
  Towers of Hanoi \((2^n)\)
  Recursively calculating \(n^{th}\) Fibonacci number \((2^n)\)

- Logarithmic
  \[ T(n) = \lg n \] [+ possible lower order terms]
  
ex: finding a particular array element (binary search)
  any algorithm that continually divides a problem in half
A Graph of Growth Functions

![Graph of Growth Functions](image-url)

- T(n) vs Problem Size, n
- Functions: lg(n), n lg(n), n^2, n^3, 2^n
Expanded Scale

![Graph showing different growth rates: lg(n), n lg(n), n, n^2, n^3, and 2^n]
Asymptotic Analysis

How does the time (or space) requirement grow as the problem size grows really, really large?

- We are interested in “order of magnitude” growth rate

Simplifying Assumptions:

- We drop constant multipliers
  \[ T(n) = cn^2 \implies T(n) = n^2 \]
- Lower order terms don’t matter
  \[ T(n) = n^2 + n \implies T(n) = n^2 \]

We call the result “Big-Oh”
Analysis Cases

- What particular input (of given size) gives worst/best/average complexity?

**Best Case**: time for the absolutely best input for the algorithm

**Worst Case**: time for the absolutely worst input for the algorithm

**Average case** is the “run time efficiency” over all possible inputs

- Mileage example: how much gas does it take to go 20 miles?
  - **Worst case**: all uphill
  - **Best case**: all downhill, just coast
  - **Average case**: “average” terrain
Cases Example

- Consider sequential search on an unsorted array of length $n$, what is time complexity?
  
  - Best case:
  
  - Worst case:
  
  - Average case:
Analyzing the Complexity of Code

1. All simple procedural Java statements take $O(1)$

2. For consecutive statements, we add their complexities
   - Since lower-order terms don’t matter, this is the same as taking their max()

3. For nested statements (loops, function calls), we multiply their complexities
Example

- **Code:**

```c
a = b;
++sum;
int y = 42 % 16;
```

- **Complexity:**
Example

- **Code:**

```java
sum = 0;
for (i = 1; i <= n; i++)
    sum += n;
```

- **Complexity:**
Example

- Code:

```java
sum1 = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        sum1++;
```

- Complexity:
Example

- **Code:**
  ```
  sum2 = 0;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
      sum2++;
  ```

- **Complexity:**
Example

- Code:
  
  ```
  sum = 0;
  for (j = 1; j <= n; j++)
      for (i = 1; i <= j; i++)
          sum++;
  for (k = 0; k < n; k++)
      a[k] = k;
  ```

- Complexity:
Example

- **Code:**
  
  ```
  sum1 = 0;
  for (k = 1; k <= n; k *= 2)
    for (j = 1; j <= n; j++)
      sum1++;
  ```

- **Complexity:**