Analysis of Algorithms & Big-O
function find_least_important_seam(vals):

dirs = 2D array with same dimensions as vals

costs = 2D array with same dimensions as vals

costs[height-1] = vals[height-1] // initialize bottom row of costs

for row from height-2 to 0:
    for col from 0 to width-1:
        costs[row][col] = vals[row][col] +
            min(costs[row+1][col-1],
                costs[row+1][col],
                costs[row+1][col+1])
        dirs[row][col] = -1, 0, or 1 // depending on min

// Find least important start pixel
min_col = argmin(costs[0]) // Returns index of min in top row

// Create vertical seam of size ‘height’ by tracing from top
seam = []
seam[0] = min_col
for row from 0 to height-2:
    seam[row+1] = seam[row] + dirs[row][seam[row]]

return seam
How fast is this algorithm?

```javascript
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```
Let’s measure it

- Implement it (in Python)
- Run it
- Time it
Let’s measure it

- Implement it (in Python)
- Run it
- Time it
- Repeat for different input sizes
What might affect these measurements?
Let's try something else

- Have to look at the algorithm
- How long will it take?
- Depends on how long each operation takes
  - +
  - *
  - array[i]
- Let's assume each operation takes the same amount of time
How fast is this algorithm?

```plaintext
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- Do we count "return error"?
  - depends on whether input array is empty
  - if `array` is empty then `sum_array` takes 2 ops
  - if `array` is not empty then `sum_array` takes ??? ops
Run time depends on input

- Which inputs should we choose?
  - Best-case?
  - Worst-case?
  - Average-case?

- In general, worst-case
  - CS is an engineering discipline
  - If I’m building a bridge, don’t care about best-case weight tolerance!
How fast is this algorithm?

```javascript
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- How long in non-empty case?
  - Depends on loop length, which depends on array length
  - Call the running time on an array of length n \( T(n) \)
  - What's \( T(n) \)?
How fast is this algorithm?

```plaintext
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- $T(n) = 3n + 3$ ops
- Do we believe this number? What assumptions did we make?
- What if array accesses take twice as long as addition?
Could be any of these...

- $3n+3$
- $4n+3$
- $n+5$
- ...

- What can we say for sure?

Linear
Running Times

**Constant**
independent of input size

**Linear**
dePENDs on input size

**Quadratic**
dePENDs on square of input size
Constant Running Time

- How many operations are executed?
  - \( T(n) = 2 \) ops
  - What if array has 100 elements?
  - What if array has 100,000 elements?

Key observation:
- Running time does not depend on array size!
What’s the running time?

```javascript
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    // between any two elements in the list
    products = []
    for i in [0, array.length):
        for j in [0, array.length):
            products.append(array[i] * array[j])
    return products
```

Quadratic
function argmax(array)
    // Input: an array
    // Output: the index of the maximum value
    index = 0
    for i in [1, array.length):
        if array[i] > array[index]:
            index = i
    return index

1op
loop
3ops per loop
1op per loop
(sometimes)
1op

Linear
Q: how do we compare running times?
Which Algorithm is Better?

- Algorithm **A** takes $T_A(n) = 30n + 10$ ops
- Algorithm **B** takes $T_B(n) = 5n$ ops
Which Algorithm is Better?

- Alg A takes $T_A(n) = 5n + 1000$ ops
- Alg B takes $T_B(n) = 10n + 2$ ops
- It depends on $n$

$$rtime(A) < rtime(B) \iff 5n + 1000 < 10n + 2 \iff 5n > 998 \iff n > 199.6$$
Which Algorithm is Better?

- **Alg A** takes $T_A(n) = 1000n^2$ ops
- **Alg B** takes $T_B(n) = n^8$ ops
- It depends on $n$

\[
\text{rtime}(A) < \text{rtime}(B) \iff 1000n^2 < n^8 \\
\iff 1000n^2 - n^8 < 0 \\
\iff n^2(1000 - n^6) < 0 \\
\iff 1000 - n^6 < 0 \\
\iff n > 1000^{1/6} \\
\iff n > 3.16\
\]
What is Running Time?

Asymptotic worst-case running time

\[ \text{Number of elementary operations on worst-case input as a function of input size } n \text{ when } n \text{ tends to infinity} \]

In CS “running time” usually means asymptotic worst-case running time…but not always!

we will learn about other kinds of running times
Comparing running times

Comparing asymptotic running times

\[ T_A(n) \text{ is better than } T_B(n) \text{ if } \]
\[ \text{for large enough } n \]
\[ T_A(n) \text{ grows slower than } T_B(n) \]
Q: can we formalize all this mathematically?
Big-O

**Definition (Big-O):** $T_A(n)$ is $O(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$’s order of growth is at most $T_B(n)$’s order of growth
- Examples
  - $2n+10$ is $O(n)$
Big-O

- How do we find “the Big-O of something”?
  - Usually you “eyeball” it
  - Then you try to prove it
    - (most of the time in CS16 it will be simple enough that you don’t need to prove it)
Big-O Examples

\[ \text{Definition (Big-O): } T_A(n) \text{ is } O(T_B(n)) \text{ if there exists positive constants } c \text{ and } n_0 \text{ such that:} \\
T_A(n) \leq c \cdot T_B(n) \]
for all \( n \geq n_0 \)

- Guess that \( 2n+10 \) is \( O(n) \). Can we prove it?
  - If we set \( c=3 \), can we find an \( n_0 \) such that for all \( n \geq n_0 \), \( 2n+10 \leq 3 \cdot n \)?

\[ 2n+10 \leq 3n \iff -n + 10 \leq 0 \iff n \geq 10 \]
What is $n_0$?

We don’t care what happens here

We only care what happens here
More Big-O Examples

- $n^2$ is not $O(n)$. Why?
  - What do we have to show to prove that $n^2$ is $O(n)$?
  - We have to find a positive constant $c$,
  - and a positive constant $n_0$ such that
  - for all $n > n_0$, $n^2 \leq c \cdot n$
  - This is not possible!

$n^2 \leq c \cdot n \iff n \leq c$

Not possible when $n$ grows & $c$ is constant
Eyeballing Big-O

- If \( T(n) \) is a polynomial of degree \( d, T(n) = an^d + bn^{d-1} + \ldots + wn + z \)
  - then \( T(n) \) is \( O(n^d) \)

- In other words you can ignore
  - lower-order terms
  - constant factors

- Examples
  - \( 1000n^2 + 400n + 739 \) is \( O(n^2) \)
  - \( n^{80} + 43n^{72} + 5n + 1 \) is \( O(n^{80}) \)

- For the Big-O, use the smallest upper bound
  - \( 2n \) is \( O(n^{50}) \) but that’s not really a useful bound
  - instead it is better to say that \( 2n \) is \( O(n) \)
• $n^{10} + 2020$ is $O(n^{10})$ but also $O(n^{11}), \ldots, O(n^{50}), \ldots$
  • but better to say it is $O(n^{10})$
• There are at most 300 people in this room
  • there are also at most 1000, \ldots, 1M, \ldots
  • but telling me there are at most 300 is more “useful”
More Eyeballing Big-O

- Find Big-O of 3 algorithms
  - Runtime of first is $T(n) = 2$
  - Runtime of argmax is $T(n) = 4n + 2$
  - Runtime of possible_products is $T(n) = 4n^2 + n + 3$
- Replace constants with "c" (they are irrelevant as $n$ grows)
  - First: $T(n) = c$
  - Argmax: $T(n) = c_0n + c_1$
  - Possible_products: $T(n) = c_0n^2 + n + c_1$
More Eyeballing Big-O

- Discard lower-order terms & constants
  - first: $T(n) = c$ is $O(1)$
  - argmax: $T(n) = c_0n + c_1$ is $O(n)$
  - possible_products: $T(n) = c_0n^2 + n + c_1$ is $O(n^2)$
  - The convention for $T(n) = c$ is to write $O(1)$
## Running Times

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>1.84 $\times$ 10$^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>3.40 $\times$ 10$^{38}$</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>16,777,216</td>
<td>1.15 $\times$ 10$^{77}$</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>134,217,728</td>
<td>1.34 $\times$ 10$^{154}$</td>
</tr>
</tbody>
</table>
Big-O

**Definition (Big-O):** \( T_A(n) \) is \( O(T_B(n)) \) if there exists positive constants \( c \) and \( n_0 \) such that:

\[
T_A(n) \leq c \cdot T_B(n)
\]

for all \( n \geq n_0 \)

- \( T_A(n) \)’s growth rate is upper bounded by \( T_B(n) \)’s growth rate
- But what if we need to express a lower bound?
  - we use Big-\( \Omega \) notation!
Big-Omega

**Definition (Big-Ω):** $T_A(n)$ is $\Omega(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \geq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$’s growth rate is lower bounded by $T_B(n)$’s growth rate
- What about an upper **and** a lower bound?
  - We use Big-$\Theta$ notation
Big-Theta

\[ T_A(n) \text{ is } \Theta(T_B(n)) \text{ if it is } O(T_B(n)) \text{ and } \Omega(T_B(n)). \]

- \( T_A(n) \)'s growth rate is the same as \( T_B(n) \)'s
## More Examples

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>Big-$O$</th>
<th>Another Big-$O$</th>
<th>Big-Ω</th>
<th>Big-$Θ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$an + b$</td>
<td>$O(n)$</td>
<td>$O(n^{100})$</td>
<td>Ω($n$)</td>
<td>Θ($n$)</td>
</tr>
<tr>
<td>$an^2 + bn + c$</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$</td>
<td>Ω($n$)</td>
<td>Θ($n^2$)</td>
</tr>
<tr>
<td>$a$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>Ω($1$)</td>
<td>Θ($1$)</td>
</tr>
<tr>
<td>$3^n + an^{40}$</td>
<td>$O(3^n)$</td>
<td>$O(50^n)$</td>
<td>Ω($n$)</td>
<td>Θ($3^n$)</td>
</tr>
<tr>
<td>$an + b \log n$</td>
<td>$O(n^2)$</td>
<td>$O(n\log n)$</td>
<td>Ω($\log n$)</td>
<td>Θ($n$)</td>
</tr>
</tbody>
</table>
Key takeaways

- We can analyze algorithm running time independent of implementation
- Important thing is behavior as input grows
- We’ll be doing a fair amount of running time analysis using Big-O notation and proofs
  - Big-O might seem complex at first
  - It’s just formalizing the intuition from earlier—constant factors don’t matter
How fast is this algorithm?

```python
function find_least_important_seam(vals):
    dirs = 2D array with same dimensions as vals
    costs = 2D array with same dimensions as vals
    costs[height-1] = vals[height-1] // initialize bottom row of costs

    for row from height-2 to 0:
        for col from 0 to width-1:
            costs[row][col] = vals[row][col] +
                min(costs[row+1][col-1],
                    costs[row+1][col],
                    costs[row+1][col+1])
            dirs[row][col] = -1, 0, or 1 // depending on min

    // Find least important start pixel
    min_col = argmin(costs[0]) // Returns index of min in top row

    // Create vertical seam of size ‘height’ by tracing from top
    seam = []
    seam[0] = min_col
    for row from 0 to height-2:
        seam[row+1] = seam[row] + dirs[row][seam[row]]

    return seam
```
Additional Readings

- To read more on asymptotic runtime and Big-O
  - Dasgupta et al. section 0.3 (pp. 15-17)
  - Roughgarden Part 1, Chap 2
References

- Slide #12
  - Usain Bolt (constant): 8-time Olympic gold medalist and greatest sprinter of all time
  - Sally Pearson (linear): 2012 Olympic world champion in 100m hurdles, 2011 and 2017 World Champion
  - Wilson Kipsang (quadratic): former marathon world-record holder, Olympic marathon bronze medalist
  - Eliud Kipchoge (quadratic): 2016 Olympic marathon gold medalist, greatest marathoner of the modern era