Maximum Number

- **Problem.** Given a set $S$ of numbers, find the largest number in the set.
- **Algorithm A:**
  1. Let $M$ be the first number in $S$.
  2. For each number $s$ in $S$:
     - If $s > M$, place the value of $s$ in $M$. Go back to the beginning of $S$ and start step 2 once again.
     - If we got to the end of $S$, return $M$ and stop.

$S = 1, 5, -3, 3.14, 100.1, 432, 123, 123.4$
Maximum Number

• **Problem.** Given a set $S$ of numbers, find the largest number in the set.

• **Algorithm B:**
  1. Let $M$ be the first number in $S$.
  2. Go over each number $s$ in $S$:
     • If $s > M$, place the value of $s$ in $M$.
     • Once we got to the end of $S$, return $M$ and stop.

$$S = 1, 5, -3, 3.14, 100.1, 432, 123, 123.4$$

Which Algorithm is More Efficient?

• **Algorithm A:**
  1. Let $M$ be the first number in $S$.
  2. Go over each number $s$ in $S$:
     • If $s > M$, place the value of $s$ in $M$. Go back to the beginning of $S$ and start step 2 once again.
     • If we got to the end of $S$, return $M$ and stop.

• **Algorithm B:**
  1. Let $M$ be the first number in $S$.
  2. Go over each number $s$ in $S$:
     • If $s > M$, place the value of $s$ in $M$.
     • Once we got to the end of $S$, return $M$ and stop.
Running Time

• Analyzing the **running time of an algorithm**:
  ◦ Each basic operation takes **1 unit of time**: addition, placing a value in a variable, checking the next number of the list, etc.
  ◦ **Obviously false!** We will see why this is reasonable to do.
  ◦ Running time is with respect to the **size of the input**: How long it takes to find max of \( n \) numbers.

Running Time: Example 1

• What happens for **input** \( n, n-1, n-2, \ldots, 1 \)?

  • **Algorithm A.**
    ◦ Put first element in \( M \): **1** time unit.
    ◦ Go over every element of the list: **\( n \)** units.
    ◦ Compare each element to \( M \): **\( n \)** units.
    ◦ Return \( M \): **1** unit.
    ◦ **Total: ** \( 2n + 2 \) time units.

  • **Algorithm B.**
    ◦ Exactly the same. **Total: ** \( 2n + 2 \) time units.
Running Time: Example 2

- What happens for input $1, 2, 3, \ldots, n$?
- **Algorithm A.**
  - Put first element in $M$: 1 time unit.
  - Compare $M$ to first element, compare $M$ to second element, place second element in $M$: 3 time units.

Running Time: Example 2

- What happens for input $1, 2, 3, \ldots, n$?
- **Algorithm A.**
  - Put first element in $M$: 1 time unit.
  - Go until element 2 and set $M = 2$: 3 units.
  - Go until element 3 and set $M = 3$: 4 units.
  - ...
  - Go until element $n$ and set $M = n$: $n + 1$ units.
  - Return $M$: 1 unit.
  - **Total:** $1 + 1 + 3 + 4 + 5 + \cdots + (n + 1) = \frac{n^2 + 3n}{2}$ time units.
Running Time: Example 2

- What happens for input \{1,2,3, \ldots, n\}?
- **Algorithm A.**
  - Total: \( 1 + 1 + 3 + 4 + 5 + \cdots + (n + 1) = \frac{n^2+3n}{2} \) time units.
- **Algorithm B.**
  - Similar to Example 1, which took \( 2n + 2 \). But with \( n - 1 \) new assignments to \( M \).
  - Total: \( 3n + 1 \) time units.
- In one example both algorithms behaved the same, but not in the other!

Worst Case Analysis

- Analyzing the running time of an algorithm:
  - Consider the maximum number of steps that the algorithm requires for an input of size \( n \).
  - **Worst case analysis.**
- In our example, the worst case running times are
  - Algorithm A: \( \frac{n^2+3n}{2} \).
  - Algorithm B: \( 3n + 1 \).
Rate of Growth

- We have two algorithms for the same problem.
  - Alg C has worst case running time of $n^8$.
  - Alg D has worst case running time of $10^{10}n^2$.

- **Which algorithm is better?**
  - Alg D is better when $n \geq 47$.

- We care about **large** $n$.

Checking Some Values of $|V|$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$10^{10} \cdot n^2$</th>
<th>$n^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$10^{10}$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$2.5 \cdot 10^{11}$</td>
<td>390,625</td>
</tr>
<tr>
<td>10</td>
<td>$10^{12}$</td>
<td>$10^8$</td>
</tr>
<tr>
<td>50</td>
<td>$2.5 \cdot 10^{13}$</td>
<td>$\sim 3.9 \cdot 10^{13}$</td>
</tr>
<tr>
<td>100</td>
<td>$10^{14}$</td>
<td>$10^{16}$</td>
</tr>
<tr>
<td>10,000,000</td>
<td>$10^{24}$</td>
<td>$10^{56}$</td>
</tr>
</tbody>
</table>
What Can a Computer Do?

\[(2 \cdot 10^{20}) \cdot (3 \cdot 10^8) \cdot (14 \cdot 10^9) < 10^{38}.\]

Seconds per year  Age of universe

- Even if we had MUCH MUCH MUCH ... MUCH faster computers, we won’t be able to run programs with \(10^{56}\) steps.

- \(10^{10} n^2\) is so much better than \(n^8\)! 

---

**Top 10 positions of the 64th TOP500 in November 2019**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Rank (original)</th>
<th>Name</th>
<th>Model</th>
<th>Processor</th>
<th>Interconnect</th>
<th>Vendor</th>
<th>Site country, year</th>
<th>Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>148.600</td>
<td>Summit</td>
<td>IBM Power System AC922</td>
<td>POWER8, Tesla V100</td>
<td>InfiniBand EDR</td>
<td>IBM</td>
<td>Oak Ridge National Laboratory United States, 2018</td>
<td>Linux (RHSE)</td>
</tr>
<tr>
<td>2</td>
<td>94.944</td>
<td>Sierra</td>
<td>IBM Power System 9023LC</td>
<td>POWER8, Tesla V100</td>
<td>InfiniBand EDR</td>
<td>IBM</td>
<td>Lawrence Livermore National Laboratory United States, 2018</td>
<td>Linux (RHSE)</td>
</tr>
</tbody>
</table>
Asymptotic Running Time

- We care about the running time of programs that receive **large inputs**.
  - When analyzing running times, we first care about the **dependency in n**.
  - We will not care whether an algorithm requires $10n^2$ steps or $100n^2$ steps.
  - What about $n^2 + 1000n + 10^{10}$?
  - When $n$ is sufficiently large
    - $1000n + 10^{10} < n^2$.
  - So $n^2 + 1000n + 10^{10} < 2n^2$.

$O(\cdot)$-Notation

- Asymptotic notation:
  - Functions $f(n)$ and $g(n)$ represent running times.
  - $f(n) = O(g(n))$ means that exist $c, d > 0$ such that $f(n) \leq c \cdot g(n)$ for every $n \geq d$.
  - **Intuitively**, $f(n)$ does not grow faster than $g(n)$ when $n$ is large.
  - **Example**: $n^2 + 5 = O(n^3)$ because $n^2 + 5 \leq n^3$ for every $n \geq 3$.
  - In this case $c = 1$ and $d = 3$. 
The Heart of the Matter

\[ n^2 + 5 = O(n^3) \]

means that \( n^2 + 5 \) does not grow faster than \( n^3 \).

(when \( n \) is large and up to constant factors.)

\[ O(\cdot) - \text{Notation} \]

- Asymptotic notation:
  - Functions \( f(n) \) and \( g(n) \) represent running times.
  - \( f(n) = O(g(n)) \) means that exist \( c, d > 0 \) such that \( f(n) \leq c \cdot g(n) \) for every \( n \geq d \).
  - Intuitively, \( f(n) \) does not grow faster than \( g(n) \) when \( n \) is large.
  - Example 2: It is not true that \( n^2 = O(n) \). For any \( c \) we have \( n^2 > cn \) for sufficiently large \( n \).
  - \( n^2 \) grows faster than \( n \).
\( O(\cdot) \)-Notation Exercises

- True or False?
  - \( n^3 + n^4 = O(n^4) \).

- \( n^3 + n^4 \leq n^4 + n^4 = 2n^4 \)
- True: \( n^3 + n^4 \) does not grow faster than \( n^4 \).
- Formally. Setting \( c = 2 \) and \( d = 1 \), we get that \( n^3 + n^4 \leq cn^4 \) for every \( n \geq d \).
Discuss in Groups

• Which of the following are true, and why:
  ◦ $1000 = O(n)$.
  ◦ $100n^2 = O(n^2)$.
  ◦ $n + n^2 + n^3 + n^4 = O(n^4)$.
  ◦ $n^2 \cdot n^3 = O(n^4)$.
  ◦ $n^5 - 2n^4 = O(n^4)$.

For true statements, find valid values for $c$ and $d$.

Back to Finding a Maximum

• Our algs for finding a maximum:
  ◦ Algorithm A: $\frac{n^2 + 3n}{2}$.
  ◦ Algorithm B: $3n + 1$.

• How would we write these running times using $O(\cdot)$-notation?
  ◦ Algorithm A: $O(n^2)$.
  ◦ Algorithm B: $O(n)$. 
Python Example

• What is the asymptotic running time?

```python
2
n = int(input("Please enter an integer: "))
3
answer = 1
4
for i in range(2,n+1):
5
    answer = answer * i
6
print(answer)
```

◦ Several operations occur only once.
◦ The loop has \( n - 1 \) iterations.
◦ Answer: \( O(n) \).

Python Example 2

• What is the asymptotic running time?

```python
2
def isPrime(num):
3    for i in range(2,num):
4        if num % i == 0:
5            return False
6        return True
7
8
9
n = int(input("Please enter an integer: "))
10
for j in range(2,n+1):
11    if isPrime(j):
12        print(j)
```

◦ Running \( \text{isPrime}(n) \) has running time \( O(n) \).
◦ Since \( \text{isPrime}() \) is called \( n - 1 \) times, the total running time is \( O(n^2) \).
More Details

• What is the **asymptotic running time**?

```python
2 def isPrime(num):
3     for i in range(2,num):
4         if num % i == 0:
5             return False
6     return True
7
8 n = int(input("Please enter an integer: "))
9 for j in range(2,n+1):
10    if isPrime(j):
11        print(j)
```

Why does running an $O(n)$ function $n$ times gives $O(n^2)$?

\[ n + n + \cdots + n = n \cdot n = n^2 \]

$n$ times

Recap: Running Time Analysis

• **Analyzing the running time of a program:**
  ◦ With respect to input of size $n$.
  ◦ **Worst case running time** over all inputs of size $n$.
  ◦ Assume that $n$ is very large.
  ◦ Asymptotic running time: $O(\cdot)$-notation.
Test Your Intuition

- **Problem.** Given a set $S$ of $n$ numbers, find the largest number in $S$.
  - We saw an algorithm that solves the problem with running time $O(n)$.
  - Explain why an asymptotically faster algorithm cannot exist for this problem. (for example, why can’t we have an algorithm with running time $O(n^{1/2})$).

A Search Problem

- **We have a list/array and wish to find a specific element in it.**
  - **Example.** A list of people in the US, each element contains the details of the relevant person.
  - Given an SSN, we wish to return the information of the relevant person.
  - Accessing one entry takes constant time.
Basic Search

- **Basic search algorithm:**
  - Receive SSN \( X \) as input.
  - For each element \( Y \) in list:
    - If \( X \) is the SSN of \( Y \), return information of element \( Y \) and stop program.
    - Print that there is no person with this SSN.

- **Running time:**
  - List consists of \( n \) elements.
  - Number of operations is \( O(n) \).

Binary Search

- Assume that the list is sorted according to the SSNs.
  - Can we use this to search faster?
  - List \( L \) of length \( n \).

- **Compare** input SSN \( X \) to SSN in \( L[n/2] \):
  - If the numbers are identical – done!
  - If \( X \) is smaller, look only in first half of \( L \).
  - If \( X \) is larger, look only in second half of \( L \).
  - Repeat the above. At every step we cut the length of the list by half!
Binary Search: Example

We are looking for 14 in list:

<table>
<thead>
<tr>
<th>Cell #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>22</td>
</tr>
</tbody>
</table>

We first compare 14 with the element in the middle cell (cell 7). Since 8 < 14:

<table>
<thead>
<tr>
<th>Cell #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>22</td>
</tr>
</tbody>
</table>

We next consider the middle element of the green part. There are two: cells 10 and 11.

We are looking for 14 in list:

<table>
<thead>
<tr>
<th>Cell #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>22</td>
</tr>
</tbody>
</table>

Bad luck: Say we chose cell 11. Since 15 > 14

<table>
<thead>
<tr>
<th>Cell #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>22</td>
</tr>
</tbody>
</table>

Cell 9: Since 11 < 14
Mario Explains Binary Search

Run Binary Search on Your Own

Run the algorithm to find 55 in list. What cells did you check?

<table>
<thead>
<tr>
<th>Cell</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>25</td>
<td>29</td>
<td>40</td>
<td>50</td>
<td>55</td>
<td>59</td>
<td>99</td>
</tr>
</tbody>
</table>
**Binary Search: Number of Steps**

- **List** \( L \) has \( n \) elements.
- At each step we cut the length in half.
  - Sometimes we move from \( n \) to \( \lfloor n/2 \rfloor \).
- **How many steps are there?**
  - **Worst case**: Number of times we need to divide \( n \) by 2 until we get 1.
  - This is \( \log_2 n \).
- There are \( \leq \log_2 n \) steps, each requiring constant time.
  - Running time: \( O(\log_2 n) \).

---

**\( \log n \) is so Much Better than \( n \)**

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log_2 n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>~3.3</td>
</tr>
<tr>
<td>100</td>
<td>~6.6</td>
</tr>
<tr>
<td>1,000,000</td>
<td>~20</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>~30</td>
</tr>
<tr>
<td>( 10^{20} )</td>
<td>~66.4</td>
</tr>
<tr>
<td>( 10^{100} )</td>
<td>~332</td>
</tr>
</tbody>
</table>
Discuss in Groups

- **True** or **False**:
  - $10^6 \cdot n \cdot \log_2 n = O(n^2)$
  - $(\log_2 n)^5 = O((\log_2 n)^3)$
  - $2^{\log_2 n} = O(n^2)$
  - $n^5 = O(2^{\log_2 n})$
  - $n^5 = O(n^{\log_2 n})$
  - $\log_2 n = O(\log_9 n)$

- **Recall**:
  - $a^{\log_a b} = b$.
  - $b = 2^{\log_2 b}$.

Asymptotic Behavior of Logarithms

- **True** or **False**:
  - $\log_2 n = O(\log_9 n)$.
- For any $a, b > 1$:
  - $\log_b n = \frac{\log_a n}{\log_a b} = O(\log_a n)$. 
A Moment of Nonsense

- Binary search in Flamenco dance: https://www.youtube.com/watch?v=iP897ZS5Nerk

The End