CS 112 – Introduction to Computing II
Wayne Snyder
Computer Science Department
Boston University

Today: Administrivia and Motivation
Administrative Matters:
Review of course design and course policies
Motivation: Two Algorithms for Searching An Array
Sequential Search and Binary Search compared
Next Time: From Python to Java
Reading assignment will be posted on Piazza!

Motivation: Two Algorithms for Searching an Array
This course is about learning to be computer scientists and programmers. Our two main goals are

Learn object-oriented programming using Java;
Develop the tools to think scientifically about the design and analysis of data structures and algorithms.

The second goal is by far the most important, and we will get a sense for what the science of algorithms is today; next time we will start to study Java in detail...
Motivation: Two Algorithms for Searching an Array

How do we think scientifically about the programs we write? Mostly this is by analyzing how they use resources (time, space, hardware, power, other algorithms). We will focus in this class on understanding the running time of algorithms.

As an example, let's consider an unsorted list of integers:

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
78 & 25 & 2 & 15 & 26 & 38 & 7 & 45 & 12 & 19 \\
\end{array}
\]

How would we determine if a given integer, say 19, is in the list?

This problem is called "sequential search" or "linear search."

Let's consider a Python implementation......

Sequential Search in Python:

```python
>>> def seqSearch(A,n):
...     for i in range(len(A)):
...         if A[i] == n:
...             print 'Found: ', n;
...             return;
...     print 'Not found: ', n;
...

>>> A = [78, 25, 2, 15, 26, 38, 7, 45, 12, 19];
>>> seqSearch(A,19);
Found 19
>>> seqSearch(A,29);
Not found: 29
```
Motivation: Two Algorithms for Searching an Array

How would we analyze this algorithm?

We are basically interested in how long it takes to find an arbitrary member of the list......

0 1 2 3 4 5 6 7 8 9

| 78 | 25 | 2 | 15 | 26 | 38 | 7 | 45 | 12 | 19 |

Here are the kinds of questions we want to answer:

How many “basic operations” (e.g., comparing one integer to another) does it take to find the integer (or not), expressed as a function of N = number of data items.

In the worst case?
In the best case?
In the average case?

1+2+...+N = N(N+1)/2

N = number of data items = 10

Motivation: Two Algorithms for Searching an Array

How would we analyze this algorithm mathematically?

We are basically interested in how long it takes to find an arbitrary member of the list......

0 1 2 3 4 5 6 7 8 9

| 78 | 25 | 2 | 15 | 26 | 38 | 7 | 45 | 12 | 19 |

Here are the kinds of questions we want to answer:

How many “basic operations” (e.g., comparing one integer to another) does it take to find the integer (or not), expressed as a function of N = number of data items.

In the worst case?  N  10
In the best case?    1  1
In the average case? (N+1)/2  5.5
Motivation: Two Algorithms for Searching an Array

Digression on summing the series $1 + 2 + \ldots + N$:

I really like the proof of

$$\sum_{i=1}^{N} i = \frac{n(n+1)}{2}$$

in which $1 + 2 + \ldots + (n-1) + n$ is written forwards then backwards and summed. It is claimed that Gauss had come up with this when he was just a child, although contended.

The proof:

Let

$$S = 1 + 2 + \ldots + (n-1) + n.$$  

Clearly,

$$S = n + (n-1) + \ldots + 2 + 1.$$  

Sum to get

$$2S = (n+1) + (n+1) + \ldots + (n+1) + (n+1).$$ 

Hence,

$$2S = n(n+1),$$  

and

$$S = \frac{n(n+1)}{2}.$$ 

Now let's consider how things change when we sort the list into ascending order:

```
0  1  2  3  4  5  6  7  8  9
```

```
 2  7 12 15 19 25 26 38 45 78
```

Now how would we determine if a given integer, say 15, is in the array?

The best way to do this is called "binary search."

Again let's consider the Python implementation......
Motivation: Two Algorithms for Searching an Array

```python
>>> def binSearch (A,n):
...     return binSearchAux(A,n,0,len(A)-1)
...
>>> def binSearchAux(A,n,left,right):
...     if right < left:
...         return False;
...     else:
...         mid = (left+right) // 2;
...         if n == A[mid]:
...             return True;
...         elif n < A[mid]:
...             return binSearchAux(A,n,left,mid-1);
...         else:
...             return binSearchAux(A,n,mid+1,right);
...
>>> A = [2, 7, 12, 15, 19, 25, 26, 38, 45, 78 ];

>>> binarySearch(A,15);
True

>>> binarySearch(A,29);
False
```

How would we analyze this algorithm mathematically?

Again, we want to count the number of "basic operations" such as comparisons:

```

<table>
<thead>
<tr>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td>25</td>
<td>26</td>
<td>38</td>
<td>45</td>
<td>78</td>
</tr>
</tbody>
</table>
```

In the worst case?
In the best case?
In the average case?
Motivation: Two Algorithms for Searching an Array

How would we analyze this algorithm mathematically?

Again, we want to count the number of “basic operations” such as comparisons:

<table>
<thead>
<tr>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>12</td>
<td>19</td>
<td>25</td>
<td>26</td>
<td>38</td>
<td>45</td>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>

In the worst case? \( \text{floor}(\log_2(N)) + 1 \)
In the best case? 1
In the average case? approx. \( 0.87 \times \log_2(N) \)

**RECALL:**

\[ \log_A B = C \]
iff
\[ A^C = B \]

floor(M) = largest integer \( \leq M \)

So for what \( X \) does \( N = 2^X \)? Clearly, \( X = \log_2(N) \).
The answer to the question “how many times can I divide \( N \) by 2 before I get 1” is “approximately \( \log_2(N) \).”

A more precise analysis gives us floor(\( \log_2(N) \)) + 1, but we will be satisfied with the approximate answer of \( \log_2(N) \) in this class......

---

How to derive the \( \log(N) \) bound on the worst case:

Let us count the **approximate size of the sublist** to be searched in the worst case at each call of the function:

Original List: \( N/1 = N/2^0 \)
After 1 comparison: \( N/2 = N/2^1 \)
After 2 comparisons: \( N/4 = N/2^2 \)
.....

After \( X \) comparisons: \( N/N = 1 = N/2^x \)

So for what \( X \) does \( N = 2^X \)? Clearly, \( X = \log_2(N) \).
The answer to the question “how many times can I divide \( N \) by 2 before I get 1” is “approximately \( \log_2(N) \).”

A more precise analysis gives us floor(\( \log_2(N) \)) + 1, but we will be satisfied with the approximate answer of \( \log_2(N) \) in this class......
Motivation: Two Algorithms for Searching an Array

How would we analyze this algorithm experimentally?

Again, we want to count the number of “basic operations” such as comparisons, but in this case we will run the program with sample data (randomly generated lists) and actually count the number of comparisons.

Hopefully, our mathematical and experimental results are consistent!!
Experiments with Sequential and Binary Search

Worst = floor(\(\log_2(N)\)) + 1
Avg = \(\sim 0.87 \times \log_2(N)\)

The Punchline:
Understanding data structures and algorithms scientifically can make the difference between a good solution and a bad solution, or between success and failure for a programming task!
It also will enable you to survive your first "coding interview"!