Searching and Sorting
Searching in a List

- **Search Problem.** Given a list $L$ of length $n$ and an item $e$, is item $e$ present in $L$?

- Function `linearSearch(L, e)`

- Examples

  >>> linearSearch([12, 16, 23, 2, 7], 16)
  True

  >>> linearSearch(['hello', 'world', 'silly'], 'hi')
  False

  >>> linearSearch(['a', 'e', 'i', 'o', 'u'], 'u')
  True
Searching in an Unsorted List

- **Search Problem.** Given a list $L$ of length $n$ and an item $e$, is item $e$ present in $L$?
- In the worst case need to look through the entire list
- $O(n)$ algorithm

```python
def linearSearch(L, e):
    n = len(L)
    for item in L:
        if item == e:
            return True
    return False
```
What if the list is sorted?
Example: Dictionary

- How do we look up a word in a dictionary?
- Words are listed in alphabetical order
Searching in a Dictionary

• How do we look up a word in a dictionary?
• Words are listed in alphabetical order
Search Algorithm

- Look at the middle page of the dictionary for our query word
- If we find our query, great!
- Otherwise:
  - If our query is later in alphabetical order to the words on the page, look for the query between the middle page and the last page
  - If our query is earlier in alphabetical order, look for the query between the middle page and the first page
How Good is This?

- **Goal**: Analyze how many pages we need to look at to look a word up in the dictionary
- Want the worst case: it’s possible that I’m looking for a word that’s right on the middle page
- Each time we look at the “middle” page remaining, the number of pages we need to look at is divided by 2 (Actually slightly better since we can rule out the middle page itself)
- A 1024-page dictionary requires at most 11 lookups: 1024 pages, < 512, <256, <128, <64, <32, <16, <8, <4, <2, <1 page.
- Just needed to look at 11 pages out of 1024!
- Can we generalize this for an $n$ page dictionary?
Review: Logarithm

- Logarithms are the inverse function to exponentiation.

- Thus, $\log_b n$ describes the exponent to which base $b$ must be raised to produce $n$.

- That is, $b^{\log_b n} = n$.

- Another way to look at logarithms:
  - $\log_b n$ is, essentially, the number of times $n$ must be divided by $b$ to reduce it to 1.

- For us, important takeaway:
  - How many times can we divide $n$ by 2 until we get down to 1
  - $\approx \log_2 n$.
Log Summary

• Logarithms are defined by the relationship \( n = b^{\log_b n} \)

• The value \( \log_b n \) is, essentially, the number of times \( n \) must be divided by \( b \) to reduce it to 1

• In CS, we often set \( b = 2 \) as we often design solutions where we divide the problem size in half

• We ignore base in Big Oh notation because they affect the value by a constant, that is,
\[
\log_a n = \frac{\log_b n}{\log_b a} = \log_b n \cdot \log_a b = c \log_b n
\]

• Thus we write \( O(\log n) \) to describe logarithmic growth with respect to the input
Binary Search Algorithm

0

m = n//2

n = len(L)

sorted list L

If e == L[m], return True
Binary Search Algorithm

```
n = len(L)
m = n//2

If e < L[m], then need to search in L[:m]
If e == L[m], return True
```
Binary Search Algorithm

\[ n = \text{len}(L) \]

\[ m = \frac{n}{2} \]

- If \( e > L[m] \), then need to search in \([m+1:]\)
- If \( e < L[m] \), then need to search in \([:m]\)
- If \( e = L[m] \), return True

Sorted list \( L \)

\( 0 \quad m = \frac{n}{2} \quad n = \text{len}(L) \)
Binary Search Algorithm

- Base cases:
  - Positive: \( \text{if } e = L[m], \text{ return True} \)
  - Negative: \( \text{if } \text{len}(L) = 0, \text{ return False} \)
Binary Search Code

- Let's **implement this algorithm**
- See Jupyter Notebook

```python
n = len(L)
m = n//2

If e < L[m], then need to search in L[:m]
If e == L[m], return True
If e > L[m], then need to search in L[m+1:]
```
Analysis of Binary Search
Within a recursive call (function frame):

- Constant number of steps (independent on $n$) and including at most one recursive call
- Total number of steps: $O(\# \text{ of recursive calls})$

```python
def binarySearch(L, e):
    if len(L) == 0:
        return False
    else:  # recursive case
        if L[mid] == e:
            return True
        elif e <= L[mid]:
            return binarySearch(L[:mid], e)
        else:
            return binarySearch(L[mid+1:], e)
```
Binary Search Analysis

- How many recursive calls? How many timing can we halve $L$ until either we find the element or $L$ has size $< 1$
- Size goes down by half in each recursive call:

\[ n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \cdots \rightarrow n/2^i = 1 \]

```python
def binarySearch(L, e):
    if len(L) == 0:
        return False
    else:  # recursive case
        if L[mid] == e:
            return True
        elif e <= L[mid]:
            return binarySearch(L[:mid], e)
        else:
            return binarySearch(L[mid+1:], e)
```
Binary Search Analysis

- Number of recursive calls at most $\log_2 n + 1$
- Overall binary search is a $O(\log n)$ algorithm
- Grows very slowly with respect to $n$

$\log_2 (1 \text{ billion}) \sim 30$
Sorting Algorithms
The Sorting Problem

- **Problem.** Given a list of elements, sort those elements in ascending order.
- There are many ways to solve this problem!
- Built-in sorting functions in Python
  - `sorted`: returns new sorted list
  - `sort()`: destructive sort that sorts the list its called on
- Today: how do we design our own sorting algorithm
- **Question.** What is the best way to sort $n$ items?
- We will use Big Oh to find out!
Selection Sort

• Find the smallest element and move it to the first position.
• Find the second-smallest element and move it to the second position, and so on
Selection Sort

• Find the smallest element and move it to the first position.
• Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

• Find the smallest element and move it to the first position.
• Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

• Find the smallest element and move it to the first position.
• Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on.
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position, and so on
Selection Sort

- For each index $i$ in the list $L$, we need to find the min item in $L[i+1:]$, if its smaller than $L[i]$, swap $L[i]$ with that item
Selection Sort

- For each index $i$ in the list $L$, we need to find the min item in $L[i+1:]$ so we can replace $L[i]$ with that item

- In fact we need to find the position $\text{minPosition}$ of the item that is minimum in $L[i+1:]$

- Reminder: how to swap values of variables $a$ and $b$?
  - Using tuple assignment in Python:
    \[
    a, b = b, a
    \]

- Let's implement this algorithm
Selection Sort: Analysis
Selection Sort Analysis

- For \( i = 0 \), inner loop runs \( n - 1 \) items
- For \( i = 1 \), inner loop runs \( n - 2 \) times
- ...
- For \( i = n - 1 \), inner loop runs 0 times

```python
def selectionSort(L):
    """Destructive sort of list L, returns sorted list."""
    n = len(L)
    for i in range(n):
        minPosition = i
        for j in range(i+1, n):
            if L[minPosition] > L[j]:
                minPosition = j
        L[i], L[minPosition] = L[minPosition], L[i]
    return L
```
Selection Sort Analysis

- Within the inner loop we have $O(1)$ steps (constant)
- Overall number of steps $(n - 1) + (n - 2) + \cdots + 0 \leq n + (n - 1) + (n - 2) + \cdots + 1$
- What is this sum?

```python
def selectionSort(L):
    """Destructive sort of list L, returns sorted list."""
    n = len(L)
    for i in range(n):
        minPosition = i
        for j in range(i+1, n):
            if L[minPosition] > L[j]:
                minPosition = j
        L[i], L[minPosition] = L[minPosition], L[i]
    return L
```
Selection Sort Analysis

\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
Guassian Summation

\[ S = n + (n - 1) + (n - 2) + \cdots + 2 + 1 \]
\[ S = 1 + 2 + \cdots + (n - 2) + (n - 1) + n \]

\[ 2S = (n + 1) + (n + 1) + \cdots + (n + 1) + (n + 1) + (n + 1) \]
\[ 2S = (n + 1) \cdot n \]
\[ S = (n + 1) \cdot n \cdot 1/2 \]
Selection Sort Analysis

• Total number of steps

\[ O(n(n + 1)/2) \]

\[ = O(n(n + 1)) \]

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

\[ = O(n^2) \]

• Is this the best we can do for sorting?

• No! Can sort in \( O(n \log n) \) time!

• Next lecture: merge sort, a faster recursive sorting algorithm that is optimal
Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and