CS 134: Sorting
Announcements & Logistics

• **Lab 9 Boggle**
  • Work on Boggle again in lab this week today/tomorrow
  • **All three parts** are due Wed/Thur at 10 pm
• **HW 9** will be released on Wed, due next Mon @ 10 pm (last one!)
• Last lab (**Lab 10**) will be a very short Java program
• We will discuss Java in last few lectures after we wrap up sorting today

Do You Have Any Questions?
Last Time: Efficiency & Searching

- Measured efficiency as number of steps taken by algorithm on worst-case inputs of a given size
- Introduced Big-O notation which captures the rate at which the number of steps taken by the algorithm grows w.r.t. size of input $n$, "as $n$ gets large"
- Compared array lists vs linked lists
- Compared linear vs binary search
Today: Searching and Sorting

- Wrap up our discussion of binary search including a runtime analysis
- Discuss some classic sorting algorithms:
  - *Selection sorting* in $O(n^2)$ time
  - A brief (high level) discussion of how we can improve it to $O(n \log n)$
  - Overview of recursive *merge sort* algorithm
Review: Logarithms

• Logarithms are the inverse function to exponentiation

• $\log_2 n$ describes the exponent to which 2 must be raised to produce $n$

• That is, $2^{\log_2 n} = n$

• Alternatively:
  
  • $\log_2 n$ (essentially) describes the number of times $n$ must be divided by 2 to reduce it to below 1

• For us, here’s the important takeaway:
  
  • How many times can we divide $n$ by 2 until we get down to 1
  
  • $\approx \log_2 n$
Review: Binary Search

- Base cases? When are we done?
  - If list is too small (or empty) to continue searching
  - If item we’re searching for is the middle element

```python
def binarySearch(aList, item):
    # Assume aList is sorted.
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False
    # base case 2
    elif item == aList[mid]:
        return True
    # recursive cases...
```
Review: Binary Search

- Recursive case:
  - Recurse on left side if item is smaller than middle
  - Recurse on right side if item is larger than middle

If item < L[mid], then need to search in L[:mid]
Review: Binary Search

- Recursive case:
  - Recurse on left side if item is smaller than middle
  - Recurse on right side if item is larger than middle

Review: Binary Search

mid = n//2

If item > L[mid], then need to search in L[mid+1:]
def binarySearch(aList, item):
    
    """Assume aList is sorted."""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False
    # base case 2
    elif item == aList[mid]:
        return True
    # recurse on left
    elif item < aList[mid]:
        return binarySearch(aList[:mid], item)
    # recurse on right
    else:
        return binarySearch(aList[mid + 1:], item)

Technically, there is one small problem with our implementation. List splicing is actually O(n)! See Jupyter for improvement.
Review: Binary Search

- **Binary search**: recursive search algorithm to search in a *sorted array list*
  - Similar to how we search for a word in a (physical) dictionary
  - Takes $O(\log n)$ time since we are reducing half of the search space on each step: $n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \cdots \rightarrow n/2^i = 1$
  - Much more efficient than a linear search
- **Note**: $\log n$ grows much more slowly compared to $n$ as $n$ gets large

But how expensive is sorting??

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

Selection Sort
Sorting

- **Problem:** Given a sequence of unordered elements, we need to sort the elements in ascending order.
- There are many ways to solve this problem!
- Built-in sorting functions/methods in Python
  - `sorted()` function that returns a new sorted list
  - `sort()` method that mutates and sorts the list it’s called on
- **Today:** how do we design our own sorting algorithm?
- **Question:** What is the best (most efficient) way to sort $n$ items?
- We will use Big-O to find out!
Selection Sort

• A possible approach to sorting elements in a list/array:
  • Find the smallest element and move (swap) it to the first position
  • Repeat: find the second-smallest element and move it to the second position, and so on
Selection Sort

- A possible approach to sorting elements in a list/array:
  - Find the smallest element and move (swap) it to the first position
  - Repeat: find the second-smallest element and move it to the second position, and so on
Selection Sort

- A possible approach to sorting elements in a list/array:
  - Find the smallest element and move (swap) it to the first position
  - Repeat: 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 and repeat.
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

- Find the smallest element and move it to the first position and repeat.
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

- Find the smallest element and move it to the first position and repeat
Selection Sort

• Generalize: For each index \( i \) in the list \( L \), we need to find the \texttt{min} item in \( L[i:] \) so we can replace \( L[i] \) with that item.

• In fact we need to find the position \texttt{minIndex} of the item that is minimum in \( L[i:] \).

• \textbf{Reminder:} how to swap values of variables \( a \) and \( b \)?
  
  • Using tuple assignment in Python: \( a, b = b, a \)
  
  • Or using a temp variable: \( \texttt{temp} = a; \ a = b; \ b = \texttt{temp} \)

• Let's implement this algorithm! (We won't use recursion this time, although we could….)
def selectionSort(myList):
    """Selection sort of given list myList, mutates list and sorts using selection sort."""
    # find size
    n = len(myList)

    # traverse through all elements
    for i in range(n):

        # find min element in remaining unsorted list
        minIndex = i
        for j in range(i + 1, n):
            if myList[minIndex] > myList[j]:
                minIndex = j

        # swap min element with element at i
        myList[i], myList[minIndex] = myList[minIndex], myList[i]

>>> myList = [12, 2, 9, 4, 11, 3, 1, 7, 14, 5, 13]
>>> selectionSort(myList)
>>> print(myList)

[1, 2, 3, 4, 5, 7, 9, 11, 12, 13, 14]
Selection Sort Analysis

- For $i = 0$, inner loop checks $n - 1$ items
- For $i = 1$, inner loop checks $n - 2$ items
- ...
- For $i = n - 1$, inner loop checks 0 items

```python
# traverse through all elements
for i in range(n):
    # find min element in remaining unsorted list
    minIndex = i
    for j in range(i + 1, n):
        if myList[minIndex] > myList[j]:
            minIndex = j

    # swap min element with element at i
    myList[i], myList[minIndex] = myList[minIndex], myList[i]
```
Selection Sort Analysis

- Within the inner loop we have $O(1)$ steps - just 1 comparison (constant)
- Thus overall number of steps is sum of inner loop steps
  $(n - 1) + (n - 2) + \cdots + 0 \leq n + (n - 1) + (n - 2) + \cdots + 1$
- What is this sum? (Math 200??)

```python
# traverse through all elements
for i in range(n):

    # find min element in remaining unsorted list
    minIndex = i
    for j in range(i + 1, n):
        if myList[minIndex] > myList[j]:
            minIndex = j

    # swap min element with element at i
    myList[i], myList[minIndex] = myList[minIndex], myList[i]
```
Selection Sort Analysis

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

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

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

\[ 2S = (n + 1) \cdot n \]

\[ S = (n + 1) \cdot n \cdot 1/2 \]

• Total number of steps taken by selection sort is thus:

  • \( O(n(n + 1)/2) = O(n(n + 1)) = O(n^2 + n) = O(n^2) \)
Sorting

Merge Sort
Towards an $O(n \log n)$ Algorithm

- There are other sorting algorithms that compare and rearrange elements in different ways, but are still $O(n^2)$ steps
  - Any algorithm that takes $n$ steps to move each item $n$ positions (in the worst case) will take at least $O(n^2)$ steps
  - To do better than $n^2$, we need to move an item in fewer than $n$ steps
- We can sort in $O(n \log n)$ time if we are clever: **Merge sort algorithm** (Invented by John von Neumann in 1945)
Merge Sort: Basic Idea

- If we split the list in half, sorting the left and right half are smaller versions of the same problem.

**Algorithm:**

- **(Divide)** Recursively sort left and right half ($O(\log n)$)
- **(Conquer)** Merge the sorted halves into a single sorted list ($O(n)$)
- (More info in extra slides at the end of this lecture!)

---

L

$m = \lfloor n \rfloor / 2$

n = len(L)

<table>
<thead>
<tr>
<th>12</th>
<th>2</th>
<th>9</th>
<th>4</th>
<th>11</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>14</th>
<th>5</th>
<th>13</th>
</tr>
</thead>
</table>

L
Selection vs Merge Sort in Practice

- Selection sort is $O(n^2)$ and merge sort is $O(n \log n)$ time
- How different is the performance in practice?
- Example: `wordList` is 12,000 words from the book *Pride & Prejudice*
- `miniList` and `medList` are the first 500 and 7000 words respectively

```python
wordList = []
with open('prideandprejudice.txt') as book:
    for line in book:
        line = line.strip().split()
        wordList.extend(line)
print(len(wordList))

>>> miniList = wordList[:500]
>>> medList = wordList[:7000]

122089
```
Selection vs Merge Sort in Practice

- miniList: 500 words
- medList: 7000 words
- wordList: ~12000 words

```
timedSorting(miniList)

Selection sort takes 0.005692720413208008 secs
Merge sort takes 0.0005681514739990234 secs
```

```
timedSorting(medList)

Selection sort takes 1.0527238845825195 secs
Merge sort takes 0.009032011032104492 secs
```

```
timedSorting(wordList)

Selection sort takes 322.0893268585205 secs
Merge sort takes 0.1942448616027832 secs
```

~5 mins vs 1/5 sec!
Summary: Searching and Sorting

- We have seen algorithms that are
  - $O(\log n)$: binary search in a sorted list
  - $O(n)$: linear searching in an unsorted list
  - $O(n \log n)$: merge sort
  - $O(n^2)$: selection sort
- Important to think about efficiency when writing code!
The end!
Leftover Slides
Problem. Given two sorted lists $a$ and $b$, how quickly can we merge them into a single sorted list?
Is $a[i] \leq b[j]$?

- Yes, $a[i]$ appended to $c$
- No, $b[j]$ appended to $c$

Merging Sorted Lists

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>merged list c</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>k</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
Merging Sorted Lists

Is \( a[i] \leq b[j] \)?
- Yes, \( a[i] \) appended to \( c \)
- No, \( b[j] \) appended to \( c \)
Merging Sorted Lists

Is \( a[i] \leq b[j] \) ?

- Yes, \( a[i] \) appended to \( c \)
- No, \( b[j] \) appended to \( c \)

\[
\begin{array}{cccccc}
2 & 4 & 9 & 11 & 12 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 3 & 5 & 7 & 13 & 14 \\
\end{array}
\]

\[\text{merged list } c\]
Merging Sorted Lists

Is $a[i] \leq b[j]$ ?

- Yes, $a[i]$ appended to $c$
- No, $b[j]$ appended to $c$

![Diagram showing merging of sorted lists](image)
Merging Sorted Lists

Is $a[i] \leq b[j]$?

- Yes, $a[i]$ appended to $c$
- No, $b[j]$ appended to $c$

\[
\begin{array}{ccccccc}
2 & 4 & 9 & 11 & 12 \\
\hline
1 & 3 & 5 & 7 & 13 & 14
\end{array}
\]

\[
\begin{array}{ccccccc}
1 & 2 & 3 \\
\hline
merged list c
\end{array}
\]
Merging Sorted Lists

Is $a[i] \leq b[j]$?
- Yes, $a[i]$ appended to $c$
- No, $b[j]$ appended to $c$
Merging Sorted Lists

- Walk through lists $a$, $b$, $c$ maintaining current position of indices $i$, $j$, $k$
- Compare $a[i]$ and $b[j]$, whichever is smaller gets put in the spot of $c[k]$
- Merging two sorted lists into one is an $O(n)$ step algorithm!
- Can use this merge procedure to design our recursive merge sort algorithm!

```python
def merge(a, b):
    """Merges two sorted lists a and b, and returns new merged list c""
    i, j, k = 0, 0, 0
    lenA, lenB = len(a), len(b)
    c = []

    # traverse and populate new list
    while i < lenA and j < lenB:
        if a[i] <= b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1
            k += 1

    # handle remaining values
    if i < lenA:
        c.extend(a[i:])
    elif j < lenB:
        c.extend(b[j:])

    return c
```
Merge Sort Algorithm

- **Base case:** If list is empty or contains a single element: it is already sorted

- **Recursive case:**
  - Recursively sort left and right halves
  - Merge the sorted lists into a single list and return it

- **Question:**
  - Where is the sorting actually taking place?

```python
def mergeSort(L):
    """Given a list L, returns a new list that is L sorted in ascending order.""
    n = len(L)

    # base case
    if n == 0 or n == 1:
        return L

    else:
        m = n//2  # middle

        # recurse on left & right half
        sortLt = mergeSort(L[:m])
        sortRt = mergeSort(L[m:])

        # return merged list
        return merge(sortLt, sortRt)
```
Merge Sort Example
Merge Sort Example