CS134: Sorting
Announcements & Logistics

• **Lab 9 Boggle**
  • Work on Boggle again in lab this week today/tomorrow
  • **All three parts** are due Wed/Thur at 11 pm
• **HW 9** will be released on Wed, due next Mon @ 11 pm
• Check calendar for updated office hours this week
• Last lab (**Lab 10**) will be a 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 wrt 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: 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
- Much more efficient than a **linear search**
- **Note**: log $n$ grows much more slowly compared to $n$ as $n$ gets large

![Graph showing time complexity of algorithms](image)
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.
    If item is in aList, return True;
    else return False.""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False

    # base case 2
    elif item == aList[mid]:
        return True
```
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

If item > L[mid], then need to search in L[mid+1:]
def binarySearch(aList, item):
    """Assume aList is sorted. If item is in aList, return True; else return False."""
    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)

There is one small problem with our implementation. List splicing is O(n)! See Jupyter for improvement.
Analysis of Binary Search

• Within a recursive call in our improved approach:
  • Constant number of steps (independent of \( n \)): just 1 comparison
  • Therefore total number of steps: \( O(\# \text{ of recursive calls}) \)
• Size of list gets cut in half in each recursive call:

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

• This is an \( O(\log n) \) time
• Really small even for large \( n \!\)

\[
\log_2 (1 \text{ billion}) \sim 30
\]
Sorting
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 its 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 \textbf{min} item in $L[i:]$ so we can replace $L[i]$ with that item.

- In fact we need to find the position $\text{minIndex}$ 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: $\text{temp} = a; a = b; b = \text{temp}$

- Let's implement this algorithm!
Selection Sort Code

```python
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 el with ith el
        myList[i], myList[minIndex] = myList[minIndex], myList[i]

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

```
Out[6]: [1, 2, 3, 4, 5, 7, 9, 11, 12, 13, 14]
```
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
# 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 el with ith el
    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 el with ith el
    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:

\[ O(n(n + 1)/2) = O(n(n + 1)) = O(n^2 + n) = O(n^2) \]
Towards an $O(n \log n)$ Algorithm

- There are many other natural sorting algorithms that compare and rearrange elements in a slightly different way, but they are still $O(n^2)$ steps.
  - Any algorithm that takes $k$ steps to move each item $k$ positions to its final position will take at least $O(n^2)$ steps as every element can be $O(n)$ away from its position in the worst case.
  - To do better than much better than $n^2$, we need to be able to move an item to its final position in significantly less steps.
  - Turns out we can sort in $O(n \log n)$ time if we are bit more clever, which is the best possible: **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!)

```
0

L

m = n//2

n = len(L)

12  2  9  4  11  13  1  7  14  5  13
```
Selection vs Merge Sort in Practice

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

In [21]:
```python
wordList = []
with open('prideandprejudice.txt') as book:
    for line in book:
        line = line.strip().split()
        wordList.extend(line)
print(len(wordList))
```
122089

In [25]:
```python
miniList = wordList[:500]
medList = wordList[:7000]
```
Selection vs Merge Sort in Practice

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

```
In [35]: timedSorting(miniList)

Selection sort takes {} secs 0.016601085662841797
Merge sort takes {} secs 0.0012111663818359375

In [36]: timedSorting(medList)

Selection sort takes {} secs 1.614171028137207
Merge sort takes {} secs 0.014803886413574219

In [37]: timedSorting(wordList)

Selection sort takes {} secs 590.5920398235321
Merge sort takes {} secs 0.39650511741638184
```

~10 mins vs 1/3 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!
- More about this in CS136!
Extra Slides
Merging Sorted Lists

- **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

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 \( c \)
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 \( c \)
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""
    # initialize variables
    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

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort Example

12  2  9  4  11

2  12

4  9  11

3  1  7

14  5  13

1  3  5  7  13  14

1  2  3  4  5  7  9  11  12  13  14