CS134 Lecture 32: Sorting
Announcements & Logistics

• **HW 10** will be released today, due Mon @ 10 pm
  • Last HW on efficiency and Big Oh
• **Lab 8** graded feedback will be returned soon
• **Lab 10** will be released today
  • Very short lab on searching and sorting (today's lecture)
  • No prelab
  • Individual lab but can discuss strategies with lab mate
• **CS134 Scheduled Final:** **Friday, May 17, 9:30 AM**
  • Room: **TCL 123**

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: captures the rate at which the number of steps taken by the algorithm grows with respect to the size of input $n$, "as $n$ gets large".
- Compared array lists vs linked lists.
- Compared linear vs binary search.

![Graph showing time vs number of elements with different Big-O complexities: $O(1)$, $O(n)$, $O(n^2)$, $O(\log n)$.]
Today: Searching and Sorting

• Review recursive implementation of binary search
• Discuss some classic sorting algorithms:
  • *Selection sort*
  • *Merge sort*
Binary Search

- The recursive search algorithm we described to search in a sorted array is called **binary search**
- It can be much more efficient than a **linear search**
  - Takes $\approx \log n$ lookups if we can index into sequence efficiently
- Which data structure supports fast access/indexing?
  - Accessing an item at index $i$ in an array requires constant time
  - Accessing an item at index $i$ in a LinkedList can require traversing the whole list (even if it is sorted!): linear time
- To get a more efficient search algorithm, it is not only important to use the right algorithm, we need to use the right data structure as well!
Binary Search

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

\[ \text{mid} = \frac{n}{2} \]
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 \( \text{item} < \text{a_lst}[\text{mid}] \), then need to search in \( \text{a_lst}[:\text{mid}] \)
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 > a_lst[mid], then need to search in a_lst[mid+1:]

mid = n//2
def binary_search(seq, item):
    """Assume seq is sorted. If item is in seq, return True; else return False."""

    n = len(seq)
    # base case 1
    if n == 0:
        return False

    mid = n // 2
    mid_elem = seq[mid]
    # base case 2
    if item == mid_elem:
        return True
    # recurse on left
    elif item < mid_elem:
        left = seq[:mid]
        return binary_search(left, item)
    # recurse on right
    else:
        right = seq[mid+1:]
        return binary_search(right, item)

Technically, there is one small problem with our implementation. List splicing is actually O(n)!
def binary_search_helper(seq, item, start, end):
    '''Recursive helper function used in binary search'''

    # base case 1
    if start > end:
        return False

    mid = (start + end) // 2
    mid_elem = seq[mid]

    if item == mid_elem:
        return True

    # recurse on left
    elif item < mid_elem:
        return binary_search_helper(seq, item, start, mid-1)

    # recurse on right
    else:
        return binary_search_helper(seq, item, mid+1, end)

def binary_search_improved(seq, item):
    return binary_search_helper(seq, item, 0, len(seq)-1)
More on Big Oh
Big-O Notation

• Tells you how fast an algorithm is / the run-time of algorithms
  • But not in seconds!
• Tells you how fast the algorithm grows in number of operations

\( O(\log n) \)

"Big O" Number of Operations
Understanding Big-O

- **Notation:** $n$ often denotes the number of elements (size)

- **Constant time** or $O(1)$: when an operation does not depend on the number of elements, e.g.

  - Addition/subtraction/multiplication of two values, or defining a variable etc are all constant time

- **Linear time** or $O(n)$: when an operation requires time proportional to the number of elements, e.g.:

  ```python
  for item in seq:
    <do something>
  ```

- **Quadratic time** or $O(n^2)$: nested loops are often quadratic, e.g.,

  ```python
  for i in range(n):
    for j in range(n):
      <do something>
  ```
Notation: \( n \) often denotes the number of elements (size)

Our goal: understand efficiency of some algorithms at a high level

Big-O: Common Functions

- \( O(1) \)
- \( O(n) \)
- \( O(n^2) \)
- \( O(\log n) \)
- \( O(1) \)
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()` list method that mutates and sorts the list

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

- 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 (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 (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 (swap) it to the first position.
- *Repeat*: find the second-smallest element and move it to the second position, and so on.
- The **gold** bars represent the sorted portion of the list.
Selection Sort

- 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.
- The gold bars represent the sorted portion of the list.
Selection Sort

- 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.
- The gold bars represent the sorted portion of the list.
Selection Sort

- 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
- The gold bars represent the sorted portion of the list.
Selection Sort

- 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.
- The *gold* bars represent the sorted portion of the list.
Selection Sort

- 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
- The gold bars represent the sorted portion of the list.
Selection Sort

- 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
- The **gold** bars represent the sorted portion of the list.
Selection Sort

- 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
- The gold bars represent the sorted portion of the list.

And now we're finally done!
Selection Sort

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

• In fact we need to find the position $\text{min_index}$ of the item that is the minimum in $\text{lst}[i:]$.

• \textbf{Reminder:} how to swap values of variables $a$ and $b$?
  - in-line swapping: $a, b = b, a$

• How do we implement this algorithm?
def selection_sort(my_lst):
    """Selection sort of a given mutable sequence my_lst, sorts my_lst by mutating it. Uses selection sort."""

    # find size
    n = len(my_lst)

    # traverse through all elements
    for i in range(n):
        # find min element in the sublist from index i+1 to end
        min_index = get_min_index(my_lst, i)

        # swap min element with current element at i
        my_lst[i], my_lst[min_index] = my_lst[min_index], my_lst[i]

You will work on this helper function in Lab 10
def selection_sort(my_lst):
    """Selection sort of a given mutable sequence my_lst, sorts my_lst by mutating it. Uses selection sort."""

    # find size
    n = len(my_lst)

    # traverse through all elements
    for i in range(n):
        # find min element in the sublist from index i+1 to end
        min_index = get_min_index(my_lst, i)

        # swap min element with current element at i
        my_lst[i], my_lst[min_index] = my_lst[min_index], my_lst[i]

Even without an implementation, can we guess how many steps does this function need to take?
Selection Sort Analysis

• The helper function `get_min_index` must iterate through index $i$ to $n$ to find the min item
  
  • When $i = 0$ this is $n$ steps
  
  • When $i = 1$ this is $n-1$ steps
  
  • When $i = 2$ this is $n-2$ steps
  
  • And so on, until $i = n-1$ this is 1 step
  
• 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? (You will see this in MATH 200 if you take it.)
Selection Sort Analysis: Visual

\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
Selection Sort Analysis: Algebraic

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

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

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

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

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

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

\[ O(n(n + 1)/2) = O(n(n + 1)) = O(n^2 + n) = O(n^2) \]
How Fast Is Selection Sort?

- Selection sort takes approximately $n^2$ steps!
More Efficient 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 ($O(\log n)$)
- *(Unite)* Merge the sorted halves into a single sorted list ($O(n)$)
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 $c$

### Diagram

- **List $a$**:
  - Elements: 2, 4, 9, 11, 12
  - Indicators: $i$

- **List $b$**:
  - Elements: 1, 3, 5, 7, 13, 14
  - Indicators: $j$

- **Merged List $c$**:
  - Elements: 1

---

**Notes**

- The process of merging two sorted lists results in a new sorted list.
- The comparison of elements from $a$ and $b$ determines the order in which they are added to $c$.
- The example shows how elements are appended based on the comparison result.
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}
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
\end{array}
```

```
\begin{array}{cccccc}
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
\end{array}
```

$k$ $i$ $j$ $c$
Merging Sorted Lists

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

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

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

\[\begin{array}{cccccc}
1 & 2 & 3 \\
\end{array}\]

\( k \) merged list \( c \)
Merging Sorted Lists

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

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

\[
\begin{array}{ccccccc}
| 2 | 4 | 9 | 11 | 12 |
\end{array}
\]

\[
\begin{array}{cccccccccc}
| 1 | 3 | 5 | 7 | 13 | 14 |
\end{array}
\]

\[
\begin{array}{cccccccccccc}
| 1 | 2 | 3 | 4 | 5 | 7 | 9 | 11 | 12 | 13 | 14 |
\end{array}
\]

merged list \( c \)
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
    len_a, len_b = len(a), len(b)
    c = []
    # traverse and populate new list
    while i < len_a and j < len_b:
        if a[i] <= b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1
    # handle remaining values
    if i < len_a:
        c.extend(a[i:]))
    elif j < len_b:
        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 merge_sort(lst):
    """Given a list lst, returns a new list that is lst sorted in ascending order."""
    n = len(lst)

    # base case
    if n == 0 or n == 1:
        return lst
    else:
        m = n//2 # middle

        # recurse on left & right half
        sort_lt = merge_sort(lst[:m])
        sort_rt = merge_sort(lst[m:]):

        # return merged list
        return merge(sort_lt, sort_rt)
```
Merge Sort Example

12  2  9  4  11
3   1  7  14  5  13

12  2  9  4  11
3   1  7  14  5  13

12  2
9   4  11
3   1  7
14  5  13

4  11
1   7
5   13
Merge Sort Example

Merge Sort Example

Merge Sort Example

Merge Sort Example
Merge Sort Analysis: Basic Idea

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

**Algorithm Analysis Rough Idea:**

- *(Divide)* Recursively sort left and right half: happens $\log n$ times.
- *(Unite)* Merge the sorted halves into a single sorted list: takes $O(n)$ times to merge two lists of $n$ items.

```
0
12  2  9  4  11  3  1  7  14  5  13
```

$m = n//2$

$n = \text{len}($lst$)$
Big Oh Comparisons

- Selection sort: $O(n^2)$
- Merge sort: $O(n \log n)$