SEARCHING AND SORTING ALGORITHMS

(download slides and .py files and follow along!)

6.0001 LECTURE 12
SEARCH ALGORITHMS

- search algorithm – method for finding an item or group of items with specific properties within a collection of items

- collection could be implicit
  - example – find square root as a search problem
    - exhaustive enumeration
    - bisection search
    - Newton-Raphson

- collection could be explicit
  - example – is a student record in a stored collection of data?
SEARCHING ALGORITHMS

- linear search
  - **brute force** search (aka British Museum algorithm)
  - list does not have to be sorted

- bisection search
  - list **MUST be sorted** to give correct answer
  - saw two different implementations of the algorithm
LINEAR SEARCH ON UNSORTED LIST: RECAP

```python
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

- must look through all elements to decide it’s not there
- \(O(len(L))\) for the loop * \(O(1)\) to test if \(e == L[i]\)
- overall complexity is \(O(n) – where n is len(L)\)

\(O(\text{len}(L))\) for the loop * \(O(1)\) to test if \(e == L[i]\)

 Assumes we can retrieve element of list in constant time

speed up a little by returning True here, but speed up doesn’t impact worst case
LINEAR SEARCH
ON SORTED LIST: RECAP

```python
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- \(O(\text{len}(L))\) for the loop * \(O(1)\) to test if \(e == L[i]\)
- overall complexity is \(O(n)\) – where \(n\) is \(\text{len}(L)\)
USE BISECTION SEARCH: RECAP

1. Pick an index, \( i \), that divides list in half
2. Ask if \( L[i] == e \)
3. If not, ask if \( L[i] \) is larger or smaller than \( e \)
4. Depending on answer, search left or right half of \( L \) for \( e \)

A new version of a divide-and-conquer algorithm

- Break into smaller version of problem (smaller list), plus some simple operations
- Answer to smaller version is answer to original problem
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high) // 2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid:  # nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
COMPLEXITY OF BISECTION SEARCH: RECAP

- `bisect_search2` and its helper
  - $O(\log n)$ bisection search calls
    - reduce size of problem by factor of 2 on each step
  - pass list and indices as parameters
  - list never copied, just re-passed as pointer
  - constant work inside function
  - $\Rightarrow O(\log n)$
SEARCHING A SORTED LIST
-- n is len(L)

- using **linear search**, search for an element is $O(n)$
- using **binary search**, can search for an element in $O(\log n)$
  - assumes the list is sorted!

- when does it make sense to **sort first then search**?
  - $\text{SORT} + O(\log n) < O(n) \Rightarrow \text{SORT} < O(n) - O(\log n)$
  - when sorting is less than $O(n)$

- **NEVER TRUE**!
  - to sort a collection of $n$ elements must look at each one at least once!
AMORTIZED COST
-- n is len(L)

- why bother sorting first?
- in some cases, may sort a list once then do many searches
- AMORTIZE cost of the sort over many searches
- SORT + \( K \cdot O(\log n) < K \cdot O(n) \)
  \( \rightarrow \) for large \( K \), SORT time becomes irrelevant, if cost of sorting is small enough
SORT ALGORITHMS

- Want to efficiently sort a list of entries (typically numbers)
- Will see a range of methods, including one that is quite efficient
MONKEY SORT

- aka bogosort, stupid sort, slowsort, permutation sort, shotgun sort

- to sort a deck of cards
  - throw them in the air
  - pick them up
  - are they sorted?
  - repeat if not sorted
COMPLEXITY OF BOGO SORT

```python
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
```

- best case: $O(n)$ where $n$ is $\text{len}(L)$ to check if sorted
- worst case: $O(?)$ it is **unbounded** if really unlucky
**BUBBLE SORT**

- **compare consecutive pairs** of elements
- **swap elements** in pair such that smaller is first
- when reach end of list, **start over** again
- stop when **no more swaps** have been made
- largest unsorted element always at end after pass, so at most n passes

CC-BY Hydrargyrum

https://commons.wikimedia.org/wiki/File:Bubble_sort_animation.gif
**COMPLEXITY OF BUBBLE SORT**

```python
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **$O(n^2)$ where $n$ is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes
SELECTION SORT

- first step
  - extract minimum element
  - swap it with element at index 0

- subsequent step
  - in remaining sublist, extract minimum element
  - swap it with the element at index 1

- keep the left portion of the list sorted
  - at i’th step, first i elements in list are sorted
  - all other elements are bigger than first i elements
ANALYZING SELECTION SORT

- loop invariant
  - given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix
  1. base case: prefix empty, suffix whole list – invariant true
  2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
  3. when exit, prefix is entire list, suffix empty, so sorted
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1

- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is $O(n^2)$ where n is len(L)
MERGE SORT

- use a divide-and-conquer approach:
  1. if list is of length 0 or 1, already sorted
  2. if list has more than one element, split into two lists, and sort each
  3. merge sorted sublists
    1. look at first element of each, move smaller to end of the result
    2. when one list empty, just copy rest of other list
MERGE SORT

- divide and conquer

- split list in half until have sublists of only 1 element
MERGE SORT

- divide and conquer

- merge such that sublists will be sorted after merge
MERGE SORT

- divide and conquer

- merge sorted sublists
- sublists will be sorted after merge
MERGE SORT

- divide and conquer

```
unsorted

sorted

sorted
```

- merge sorted sublists
- sublists will be sorted after merge
MERGE SORT

- divide and conquer – done!

| sorted |
EXAMPLE OF MERGING

<table>
<thead>
<tr>
<th>Left in list 1</th>
<th>Left in list 2</th>
<th>Compare</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,5,12,18,19,20]</td>
<td>[2,3,4,17]</td>
<td>1,2</td>
<td>[]</td>
</tr>
<tr>
<td>[5,12,18,19,20]</td>
<td>[2,3,4,17]</td>
<td>5,2</td>
<td>1</td>
</tr>
<tr>
<td>[5,12,18,19,20]</td>
<td>[3,4,17]</td>
<td>5,3</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[5,12,18,19,20]</td>
<td>[4,17]</td>
<td>5,4</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td>[5,12,18,19,20]</td>
<td>[17]</td>
<td>5,17</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>[12,18,19,20]</td>
<td>[17]</td>
<td>12,17</td>
<td>[1,2,3,4,5]</td>
</tr>
<tr>
<td>[18,19,20]</td>
<td>[17]</td>
<td>18,17</td>
<td>[1,2,3,4,5,12]</td>
</tr>
<tr>
<td>[18,19,20]</td>
<td>[]</td>
<td>18,--</td>
<td>[1,2,3,4,5,12,17]</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
<td></td>
<td>[1,2,3,4,5,12,17,18,19,20]</td>
</tr>
</tbody>
</table>
def merge(left, right):
    result = []
i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result

- left and right sublists are ordered
- move indices for sublists depending on which sublist holds next smallest element

when right sublist is empty
when left sublist is empty
COMPLEXITY OF MERGING SUBLISTS STEP

- go through two lists, only one pass
- compare only *smallest elements in each sublist*
- $O(\text{len(left)} + \text{len(right)})$ copied elements
- $O(\text{len(longer list)})$ comparisons
- *linear in length of the lists*
MERGE SORT ALGORITHM
-- RECURSIVE

```python
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:]
        return merge(left, right)
```

- **divide list** successively into halves
- depth-first such that **conquer smallest pieces down one branch** first before moving to larger pieces
COMPLEXITY OF MERGE SORT

- at **first recursion level**
  - n/2 elements in each list
  - \(O(n) + O(n) = O(n)\) where \(n\) is \(\text{len}(L)\)

- at **second recursion level**
  - n/4 elements in each list
  - two merges \(\rightarrow O(n)\) where \(n\) is \(\text{len}(L)\)

- each recursion level is \(O(n)\) where \(n\) is \(\text{len}(L)\)

- **dividing list in half** with each recursive call
  - \(O(\log(n))\) where \(n\) is \(\text{len}(L)\)

- overall complexity is \(O(n \log(n))\) where \(n\) is \(\text{len}(L)\)
SORTING SUMMARY
-- n is len(L)

- bogo sort
  - randomness, unbounded O()

- bubble sort
  - O(n^2)

- selection sort
  - O(n^2)
  - guaranteed the first i elements were sorted

- merge sort
  - O(n \log(n))

- O(n \log(n)) is the fastest a sort can be
WHAT HAVE WE SEEN IN 6.0001?
KEY TOPICS

- represent knowledge with data structures
- iteration and recursion as computational metaphors
- abstraction of procedures and data types
- organize and modularize systems using object classes and methods
- different classes of algorithms, searching and sorting
- complexity of algorithms
OVERVIEW OF COURSE

- Learn computational modes of thinking
- Begin to master the art of computational problem solving
- Make computers do what you want them to do

Hope we have started you down the path to being able to think and act like a computer scientist
WHAT DO COMPUTER SCIENTISTS DO?

- they think computationally
  - abstractions, algorithms, automated execution

- just like the three r’s: reading, ‘riting, and ‘rithmetic – computational thinking is becoming a fundamental skill that every well-educated person will need
THE THREE A’S OF COMPUTATIONAL THINKING

- **abstraction**
  - choosing the right abstractions
  - operating in multiple layers of abstraction simultaneously
  - defining the relationships between the abstraction layers

- **automation**
  - think in terms of mechanizing our abstractions
  - mechanization is possible – because we have precise and exacting notations and models; and because there is some “machine” that can interpret our notations

- **algorithms**
  - language for describing automated processes
  - also allows abstraction of details
  - language for communicating ideas & processes

```python
def mergeSort(L, compare = operator.lt):
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L)/2)
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```
ASPECTS OF COMPUTATIONAL THINKING

- how difficult is this problem and how best can I solve it?
  - theoretical computer science gives precise meaning to these and related questions and their answers

- thinking recursively
  - reformulating a seemingly difficult problem into one which we know how to solve
  - reduction, embedding, transformation, simulation

\[ O(\log n) ; O(n) ; O(n \log n) ; O(n^2) ; O(c^n) \]