SEARCHING AND SORTING ALGORITHMS

6.0001 LECTURE 12

Kevin Alejandro Hernandez Campillo

Polytechnic University of Victoria

September-December 2022

- 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 serch
        - Newton-Raphson
- collection could be explicit
    - example - is a stident record in a stored collection of data?

SERCHING 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 SERCH ON UNSORTED LIST:RECAP

```python
def linear_serch(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). 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

```python
if e==L[i]:
  found = True
```

Universidad Politécnica de Victoria "Construyendo Conciencia"

LINEAR SERCH ON UNSORTED LIST:RECAP

Example code:

```python
def linear_serch(L,e):
    found = False
    for i in range(len(L)):
      if e==L[i]:
        found = True
    return found
lista = [4,2,6,7]
print(linear_serch(lista,8))
print(linear_serch(lista,6))
```

Output:

```
False
True
```

UNIVERSIDAD POLITÉCNICA de VICTORIA
"Construyendo Conciencia"

LINEAR SERCH ON SORTED LIST:RECAP

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

- musto only look until reach a number geater than e
- O(len(L)) for the loop*O(1) to test if e==L[i]
- overall complexity is O(n)-where n is len(L)

LINEAR SERCH ON SORTED LIST:RECAP

Example code:

```python
def search(L,e):
  for i in range(len(L)):
    return True
  if L[i] > e:
    return False
  return False
lista = [3,4,6,8,9]
print(linear_serch(lista,6))
print(linear_serch(lista,5))
```

Output:

```
True
False
```

USE BISECTION SERCH: 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 answare, sherch left or right half of L for e

A new version of a divide-and conqer algorithm

- Breake into smaller version of problema(smaller list), puls some simple operations
- Answare to smaller version is answer to original problem

BISECTION SERCH IMPLEMENTATION: RECAP

```python
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 serach
                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_serch2** and its helper
  - O(log n) bisection serch 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 funcion
  - $\rightarrow$ **O(log n)**

Example code:

```
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 serach
                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)
lista = [4,2,6,7]
print(bisect_search2(lista, 6))
print(bisect_search2(lista, 5))
```

Output:

```
True
False
```

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**?
  - SORT + O(log n) < O(n) $\rightarrow$ SORT < O(n) - O(log n)
  - when sorAng is less than O(n)
- NEVER TRUE!
  - to sort a collecEon 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 + kO(log n) < kO(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, permutaAon 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

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

- best case: **O(n) where n is len(L)** to check if sorted
- worst case: O(?) it is **unbounded** if really unlucky

COMPLEXITY OF BOGO SORT

Example code:

```
import random
def is_sorted(L):
    sort = True
    for i in range(0,len(L)-1):
        if L[i] > L[i+1]:
            sort = False
    return sort
def bogo_sort(L):
    while not is_sorted(L):
        random.shuffle(L)
    return L

lista=[2,1,7,9,10]
print(bogo_sort(lista))
```

Output:

```
[1, 2, 7, 9, 10]
```

- **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 a_er pass, so at most n passes

COMPLEXITY OF BUBBLE SORT

```
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** unti no more swaps
- **$O(n^2)$ where n is len(L)** to do len(L)1 comparsions and len(L)1 passes

Example code:

```
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
    return L
lista=[7,1,4,5,3]
print(bubble_sort(lista))
```

Output:

```
[1, 3, 4, 5, 7]
```

Universidad Politécnica de Victoria "Construyendo Conciencia"

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'step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

- 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

COMPLEXITY OF SELECTION SORT

```python
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 tomes
- complexity of selection sort is **$O(n2)$ where n is len(L)**

Example code:

```python
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
    return L
lista=[4,2,5,9,3]
print(selection_sort(lista))
```

Output:

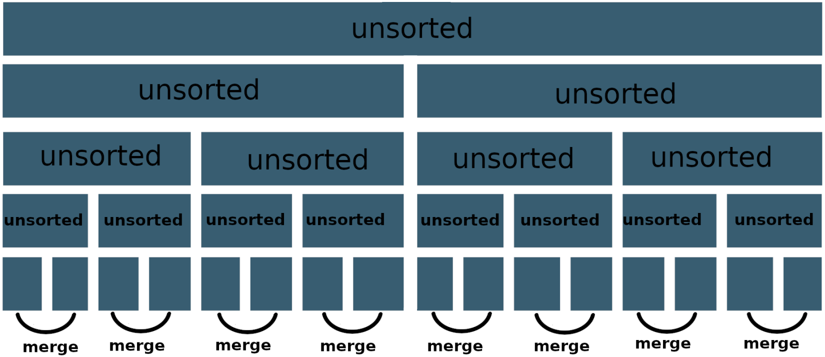
```
[2, 3, 4, 5, 9]
```

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

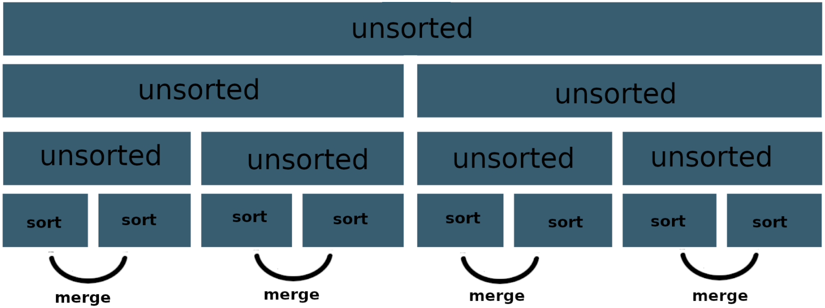
- divide and conquer



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

MERGE SORT

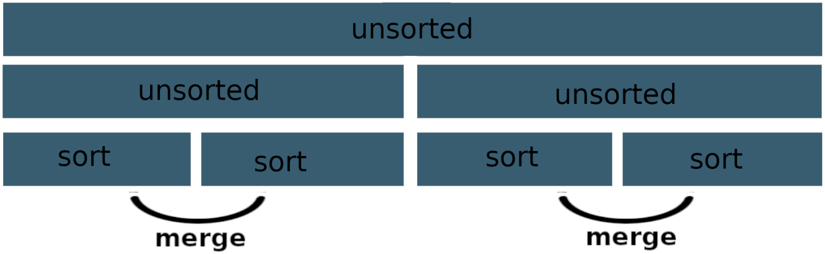
- divide and conquer



- merge such that **sublists will be sorted after merge**

MERGE SORT

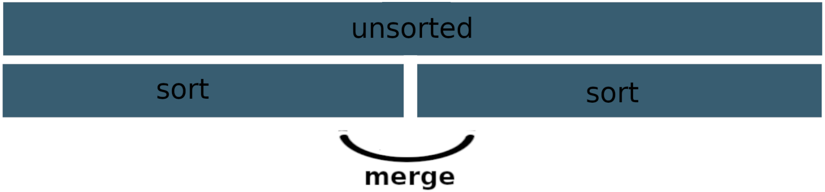
- divide and conquer



- merge sorted sublists
- sublists will bee sorted after merge

MERGE SORT

- divide and conquer



- merge sorted sublists
- soblists will be sorted after merge

MERGE SORT

- divide and conquer - done!

sort

EXAMPLE OF MERGING

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1,2	[]
[5,12,18,19,20]	[2,3,4,17]	5,2	[1]
[5,12,18,19,20]	[3,4,17]	5,3	[1,2]
[5,12,18,19,20]	[4,17]	5,4	[1,2,3]
[5,12,18,19,20]	[17]	5,17	[1,2,3,4]
[12,18,19,20]	[17]	12,17	[1,2,3,4,5]
[18,19,20]	[17]	18,17	[1,2,3,4,5,12]
[18,19,20]	[]	18,–	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

UNIVERSIDAD POLITÉCNICA de VICTORIA
"Construyendo Conciencia"

MERGING SUBLISTS STEP

```
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
```

MERGING SUBLISTS STEP - CODE EXPLAINED

```
if left[i] < right[j]:
  result.append(left[i])
  i += 1
else:
  result.append(right[j])
  j += 1
```

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

```
while (i < len(left)):
    result.append(left[i])
    i += 1
```

- when right sublist is empty

```
while (j < len(right)):
    result.append(right[j])
    j += 1
```

- when left sublist is empty

Example code:

```python
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
lista1=[2,4,7,9]
lista2=[3,8,13]
merge(lista1,lista2)
```

Output:

```
[2, 3, 4, 7, 8, 9, 13]
```

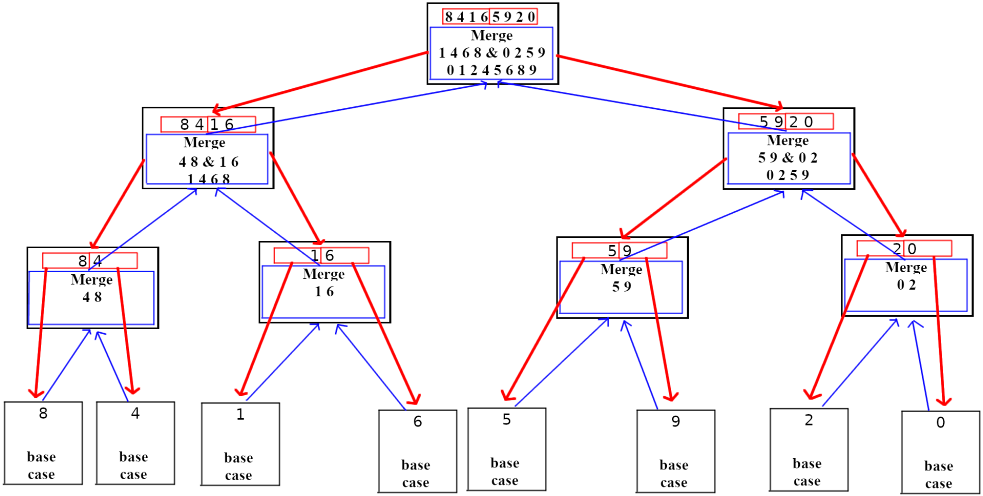
COMPLEXITY OFMERGING SUBLISTS STEP

- go through two lists, only one pass
- compare only **smallest elements in each sublist**
- O(len(left) + len(right)) copied elements
- O(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


8 4 1 6 5 9 2 0
Merge
1 4 6 8 & 0 2 5 9
0 1 2 4 5 6 8 9
8 4 1 6
Merge
4 8 & 1 6
1 4 6 8
5 9 2 0
Merge
5 9 & 0 2
0 2 5 9
8 4
Merge
4 8
1 6
Merge
1 6
5 9
Merge
5 9
2 0
Merge
0 2
8
base case
4
base case
1
base case
6
base case
5
base case
9
base case
2
base case
0
base case

COMPLEXITY OF MERGE SORT

- **at first recursion level**
  - n/2 elements in each list
  - O(n) + O(n) = O(n) where n is len(L)
- at **second recorsion level**
  - n/4 elements in each list
  - two merges $\rightarrow$ O(n) where n is len(L)
- each recursion level is O(n) where n is len(L)
- **dividing list in half** with each recursive call
  - O(log(n)) where n is len(L)
- overall complexity is **O(n log(n)) where n is len(L)**

SORTING SUMMARY –n is len(L)

- bogo sort
  - randomness, unbonded 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 sorAng
- **complexity** of algorithms

- 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

- 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

Alan Turing Image in the Public Domain, courtesy of Wikipedia Commons.

Ada Lovelace Image in the Public Domain, courtesy of Wikipedia Commons.

THE THREE A'S OF COMPUTATIONAL THINKING

- abstraction
  - choosing the right abstractions
  - operationg 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
  - anguage for describing automated processes
  - also allows abstaction of details
  - language for communicating ideas & processes

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 onewhich we know how to solve
  - reduction, embedding, transformation, simulaAon

MIT OpenCourseWare
https://ocw.mit.edu

6.0001 Introduction to Computer Science and Programming in Python
Fall 2016