SEARCHING AND SORTING ALGORITHMS

6.0001 LECTURE 12

Kevin Alejandro Hernandez Campillo

Polytechnic University of Victoria

September-December 2022

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 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

Universidad Politécnica de Victoria
"Construyendo Conciencia"

LINEAR SERCH ON UNSORTED LIST:RECAP

```
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

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

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
```

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)

Example code:

```
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)
```

UNIVERSIDAD POLITÉCNICA de VICTORIA "Construyendo Conciencia"

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)
  → for large K, **SORT time becomes irrelevant,** if cost of sorting is small enough

- 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

```python
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

UNIVERSIDAD POLITÉCNICA de VICTORIA "Construyendo Conciencia"

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]
```

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 a_er pass, so at most n passes

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

COMPLEXITY OF BUBBLE SORT

Example code:

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

Output:

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

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

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

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)**

COMPLEXITY OF SELECTION SORT

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:

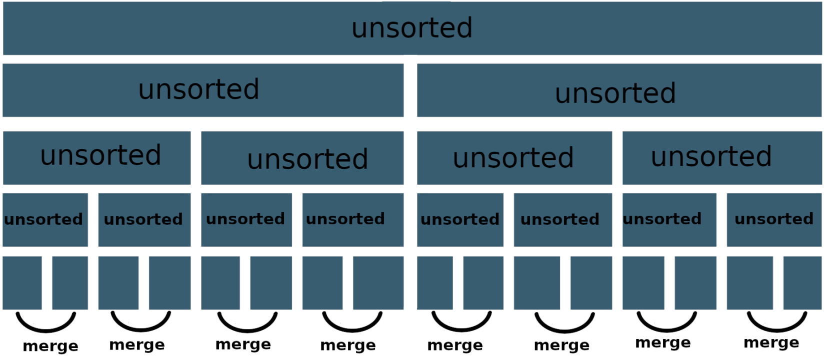
```
[2, 3, 4, 5, 9]
```

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

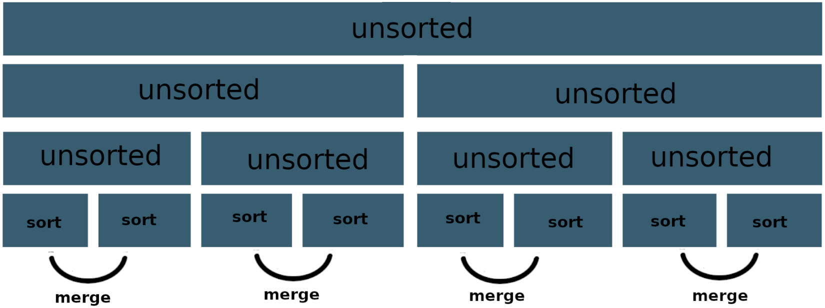
- divide and conquer



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

MERGE SORT

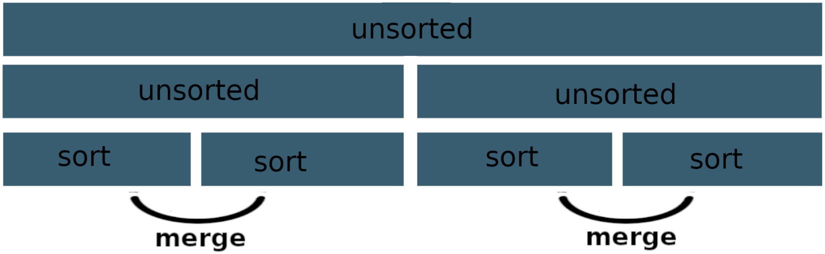
- divide and conquer



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

MERGE SORT

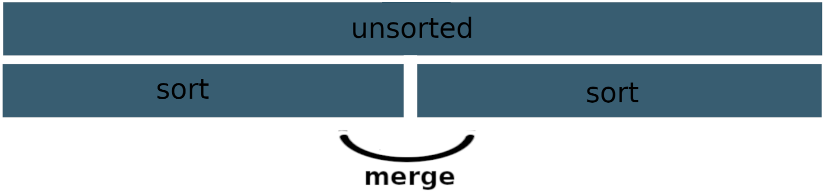
- divide and conquer



- 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]

MERGING SUBLISTS STEP

```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
```

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

Placing code in multiple columns

When required, you should adjust the code to multiple columns as shown in this slide

A variable is declared and represents a value that is expected to change throughout a program.

```
age = 25
string = 'Cadena'
```

An immutable variable is declared with the val keyword and represents a value that must remain constant throughout a program.

```
goldenRatio = 1.618
```

When a data type is not specified in a variable declaration, the variable's data type can be inferred through type inference.

```
color = "Purple"
```

String concatenation is the process of combining Strings using the + operator.

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

Using Images in Beamer

Images can be placed in one single column or in two (as shown in this slide)

- Mobile Programming
- Intelligent Systems
- Automaton and Languages