Module 08: Searching and Sorting Algorithms

Topics:

• Searching algorithms
• Sorting algorithms
Application: Searching a list

Suppose you have a list $L$. How could you determine if a particular value is in that list, if $L$ is in no particular order?

Algorithm (called Linear Search)

• Check the first element in $L$: is it the one?
  – If Yes, return True
  – Else, check the next value

• The value is not in the list if you don’t find it (return False)
Implementing Linear Search

```python
## linear_search(L, target)
##   returns True if target is in L,
##   False otherwise
## linear_search: (listof X) X -> Bool
## Note: equivalent to: target in L

def linear_search(L, target):
    for val in L:
        if val == target:
            return True
    return False
```
Running Time of `linear_search`

- Let \( n = \text{len}(L) \)
- Best Case:
  - If target is in first position, we find it right away \( \implies O(1) \)
- Worst Case:
  - If target is not in \( L \), we have to check all \( n \) elements \( \implies O(n) \)
  - What is the other worst case?
Alternatives to Linear Search

• If $L$ is unsorted, we can’t do any better than Linear Search.
• How could we improve Linear Search if $L$ was sorted into increasing order?
  – Are there situations in which we could stop earlier?
  – Is this any faster in the worst case?
A better approach: Binary Search

• Suppose \( L \) is a listing of the taxpayers in Canada, sorted into increasing order by Social Insurance numbers.
• Approximately 22,000,000 entries
• Look at \( L[11000000] \)
• Is it the target taxpayer?
  – If yes, stop.
  – If not, is \( \text{target} < L[11000000] \) ?
    • If yes, then \( \text{target} \) is in the first half of \( L \)
    • If not, then \( \text{target} \) is in the second half of \( L \)
  – Repeat this process for the half containing \( \text{target} \)
Developing \texttt{binary\_search}

• We need to determine how to keep track of the section of the list still being searched
  \rightarrow Variables \ \texttt{beginning, end}
  \rightarrow Determine their initial values
• Determine the \texttt{middle} position
• If \texttt{L[middle]} is target, return \texttt{stop}
• Otherwise, update \texttt{beginning} and \texttt{end}
• Determine when we to continue (or stop) searching
def binary_search(L, target):
    beginning = …
    end = …
    while …:
        middle = …
        if L[middle] == target:
            return True
        elif L[middle] > target:
            …
        else:
            …
    return False
binary_search tests should include

- empty list
- list of length 1: target in list and not in list
- small list, both even and odd lengths
- larger list
  - target “outside” list, i.e. \texttt{target < L[0]} or \texttt{target > L[len(L)-1]}
  - target in the list, various positions (first, last, middle)
  - target not in the list, value between two list consecutive values
Worst Case running time of `binary_search`

- What is the runtime of each iteration?
- How many iterations are required at most?

Suppose $n = 2^k$:
- First comparison reduces search region size to $2^{k-1}$
- Second comparison reduces region size to $2^{k-2}$
- Third comparison reduces region size to $2^{k-3}$
- ...
- $m^{th}$ comparison reduces region size to $2^{k-m}$
- When is the region reduced to size 1?
Comments on running time for binary_search

• Worst case running time is $O(\log n)$
  – For $n \sim 1000$, will consider at most 11 elements ($2^{10} = 1024$)
  – For $n \sim 100,000$, will consider at most 18 elements ($2^{17} = 131072$)
  – For $n \sim 22,000,000$, will consider at most 26 elements ($2^{25}=33,554,432$)
  – Doubling the size of list requires 1 more comparison worst-case!!!!
Comments and Questions on running time for `binary_search`

- Could be modified to return something other than a Boolean:
  - *What would be a good value?*

- Could be written recursively instead in Python and still have worst case run-time of $O(\log n)$
  - *Would the worst case for a recursive implementation in Racket still be $O(\log n)$?*
Application: Sorting a list

# sort_list(L) sorts L into increasing order
# Effects: L is mutated
# sort_list: (listof Int) -> None
# requires: No duplicate values in L
#
# Example: Suppose lst = [1,4,3,2],
# calling sort_list(lst) => None, but
# reorders lst as [1,2,3,4]
def sort_list(L):

Sorting Algorithms

There are many different approaches to sorting. We will study the following algorithms and their runtimes:

• Selection sort
• Insertion sort
• Mergesort
Selection sort: basic idea

• Place the smallest entry into L[0]
• Place the second smallest entry into L[1]
• Place the third smallest entry into L[2]
• ...
• After step n-1, the list is sorted.
Selection sort implementation

def selection_sort(L):
    n = len(L)
    positions = list(range(n-1))
    for i in positions:
        min_pos = i
        for j in range(i,n):
            if L[j] < L[min_pos]:
                min_pos = j
        temp = L[i]
        L[i] = L[min_pos]
        L[min_pos] = temp
Selection sort: Runtime

• Before the loop: $O(n)$
• Inner loop: $O(n)$ each iteration
• Outer loop: $O(n)$ iterations

$\Rightarrow O(n) + O(n) \times O(n)$
$\Rightarrow O(n^2)$

Selection sort is, perhaps, the easiest sorting algorithm, but there are faster algorithms.

Next up: Insertion sort
Insertion Sort: an introduction

• Idea: consider the list to be in two pieces

  SORTED     UNSORTED

• Inserting the first item in "Unsorted" into its proper place in "Sorted", shrinks "Unsorted" and enlarges "Sorted"

• Repeat this process until "Unsorted" is empty
Insertion sort: an example

Sorting L=\[5,8,2,4,3,1,9,6\]

• [5] is sorted, insert 8
• [5,8] is sorted, insert 2
• [2,5,8] is sorted, insert 4
• [2,4,5,8] is sorted, insert 3
• [2,3,4,5,8] is sorted, insert 1
• [1,2,3,4,5,8] is sorted, insert 9
• [1,2,3,4,5,8,9] is sorted, insert 6
• [1,2,3,4,5,6,8,9] is sorted. No more to insert.
# insert(L,pos) sorts L[0:pos] when L[0:pos-1]
#   is already sorted.
def insert(L, pos):
    while pos > 0 and L[pos] < L[pos-1]:
        temp = L[pos]
        L[pos] = L[pos-1]
        L[pos-1] = temp
        pos = pos-1

def insert_sort(L):
    for i in range(1,len(L)):
        insert(L,i)
Running time of `insert_sort`

`insert_sort` requires $O(n)$ calls to `insert`

`insert` requires at most $O(n)$ while loop iterations

Each while loop body requires $O(1)$ steps

$\rightarrow O(n) \times O(n) \times O(1)$

$\rightarrow O(n^2)$

*What is the best-case?*
Mergesort— another sorting algorithm

Consider the following approach

- Divide the list into two halves
- Sort the first half
- Sort the second half
- Combine the sorted lists together

⇒ Done!

*Mergesort is a "Divide and Conquer" algorithm.*
Mergesort questions

• How to split the list?
  – Find the middle – before and after

• How to sort smaller lists?
  – Use same idea again (mergesort recursively)

• When to stop recursion?
  – When the list is empty

• How to combine the parts?
  – merge
Merge helper function

Suppose \( L_1, L_2 \) are in increasing order. To merge:

- If \( L_1 \) is empty, the merged list is \( L_2 \)
- If \( L_2 \) is empty, the merged list is \( L_1 \)
- If \( L_1[0] < L_2[0] \), then the merged list is:
  \[ [L_1[0]] + \text{merge}(L_1[1:], L_2) \]
- Otherwise, the merged list is:
  \[ [L_2[0]] + \text{merge}(L_1, L_2[1:]) \]

Note: we will use a modified version of this algorithm to get a better run-time
def merge(L1, L2, L):
    pos1, pos2, posL = 0, 0, 0
    while (pos1 < len(L1)) and (pos2 < len(L2)):
        if L1[pos1] < L2[pos2]:
            L[posL] = L1[pos1]
            pos1 += 1
        else:
            L[posL] = L2[pos2]
            pos2 += 1
        posL += 1
    while (pos1 < len(L1)):
        L[posL] = L1[pos1]
        pos1, posL = pos1+1, posL+1
    while (pos2 < len(L2)):
        L[posL] = L2[pos2]
        pos2, posL = pos2+1, posL+1

Note: L1 and L2 must be sorted before `merge` is called, and L is combined length of L1 and L2

pos1, pos2, posL are list positions
Running Time of *merge*

- Suppose \( \text{len}(L1) = m \) and \( \text{len}(L2) = p \)

- Maximum number of while loop iterations is \( O(m + p) \)
- Each loop is \( O(1) \)
- Total \( \rightarrow O(m + p) \)

- Note: if \( m = n/2 \) and \( p = n/2 \), then \( O(n) \)
```python
def mergesort(L):
    if len(L) < 2: return
    mid = len(L)//2
    L1 = L[:mid]
    L2 = L[mid:]
    mergesort(L1)
    mergesort(L2)
    merge(L1,L2,L)
```

Running time:

\[ T(n) = O(n) + 2T\left(\frac{n}{2}\right) \rightarrow O(n \log n) \]
Calls to **mergesort**

[Diagram showing recursive calls to mergesort with branching factors of n/4 and n/2, leading to a tree structure with 1 at each leaf node.]
Running time of *mergesort*

- Consider the time across each level of the tree.
  - How long does it take to divide the lists in half?
  - How long does it take to merge the lists together?
- How many levels of the tree are there?

- Total running time is $O(n \log n)$
Built-in sorted and sort

- Python: `sorted`
  - Built-in function
  - Consumes a list and returns a sorted copy
- Python: `sort`
  - A list method
  - Consumes a list and modifies into sorted order
- Additional arguments can be provided to change the sort (e.g. into decreasing order)
- $O(n \log n)$ runtime for Python's sorting functions
Goals of Module 08

• Understand how linear and binary search work
• Be able to compare running times of searching algorithms
• Understand how insertion sort, selection sort and mergesort work
• Be able to compare running times of sorting algorithms