



Recursive Searching Algorithms

Module Code: CELEN086

Lecture 5

(28/10/24)

Lecturer & Convenor: Manish Dhyani

Email: Manish.Dhyani@nottingham.edu.cn

Searching and Sorting

❑ Searching

to find and retrieve a specific element within a dataset

❑ Sorting

to arrange elements in a specific order within a dataset

We will learn classical searching/sorting algorithms on data structures:
e.g., lists (Lecture 5-6), trees (Lecture 7-8) and graphs (Lecture 9-10).

Recursive Linear search algorithm

Begin by checking (one-by-one comparison) every element of the dataset (list) from the start to end until we find a match.

| | | | | | | | | | | |
|----|---|----|----|----|---|---|---|----|----|----|
| 10 | 3 | 17 | 11 | 14 | 6 | 2 | 0 | 15 | 22 | 18 |
|----|---|----|----|----|---|---|---|----|----|----|

Search for 14 (14 is the **searching key**)

| | | | | | | | | | | |
|---------------|--------------|---------------|---------------|----|---|---|---|----|----|----|
| 10 | 3 | 17 | 11 | 14 | 6 | 2 | 0 | 15 | 22 | 18 |
|---------------|--------------|---------------|---------------|----|---|---|---|----|----|----|

Considering the number of comparison steps, what is the worst case that may happen?

Search for 18 Lastly found after comparisons to all elements.

Search for 5 No result.

Algorithm: linSearch

Design a recursive algorithm that check if a given number exists in a list (assume there are no duplicated elements in the list).

Algorithm: linSearch(x, list)

Requires: a number and a list having distinct numbers

Returns: True if x exists in list; False otherwise

```
1. if isEmpty(list)
2.   return False // base case
3. elseif x == value(list)
4.   return True // base case
5. else
6.   return linSearch(x, tail(list)) // recursive call
7. endif
```

HW Exercise 1:

Trace linSearch(5,[3,1,8,5,7])

Measuring algorithm performance

When designing algorithms, we need to consider its performance mainly from two aspects:

- Time complexity

is estimated by counting the **number of elementary operations** (e.g., one-by-one comparison) performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

- Space complexity

is usually referred to as the amount of computer memory consumed by the algorithm.

Time complexity

Time complexity of an algorithm/program is **not equal to** the actual running time of a particular code.(e.g. depends on hardware, complier and language we use for coding.)

The performance in time of an algorithm can be measured as a **function of the dataset size n** (e.g., length of an input list).

For algorithms of searching/sorting a list, we may discuss the best/worst case scenarios as the references using **Big O notation**.

Big O notation

Consider finding one element in the list of length n using **linear search**:

| | | | | | | | | | | |
|----|---|----|----|-----|-----|---|---|----|----|----|
| 10 | 3 | 17 | 11 | ... | ... | 2 | 0 | 15 | 22 | 18 |
|----|---|----|----|-----|-----|---|---|----|----|----|

| | | |
|----------------------|----------|--|
| Best case scenario: | $x = 10$ | time complexity of $O(1)$ (constant in time, not depending on n) |
| Worst case scenario: | $x = 18$ | time complexity of $O(n)$ (linear in time, depending on n) |

Examples of $O(n)$: $3n + 2$, $100n$, $n + 1000$

Examples of $O(n^2)$: $4n^2$, $300n^2 - 5000n + 10000$

Examples of $O(1)$: 1 , 32 , 20000

Binary search

If the list is **sorted** (in ascending/descending order), we can perform the searching more efficiently using **binary search**.

| | | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|----|
| 0 | 2 | 3 | 6 | 10 | 11 | 14 | 15 | 17 | 18 | 22 |
|---|---|---|---|----|----|----|----|----|----|----|

Search for 14

Compare 14 to the the middle element:

$14 > 11$

Search on the right half list

| | | | | |
|----|----|----|----|----|
| 14 | 15 | 17 | 18 | 22 |
|----|----|----|----|----|

Compare 14 to the the middle element:

$14 < 17$

Search on the left half list

| | |
|----|----|
| 14 | 15 |
|----|----|

Compare 14 to the “middle” element:

$14 < 15$

Search on the left half list

| |
|----|
| 14 |
|----|

Compare 14 to the “middle” element:

$14 == 14$

Done.

Time complexity of binary search

| | Length of list | # of comparisons |
|---|-------------------------------|---------------------|
| <div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div> | n | 1 |
| <div><div></div><div></div><div></div><div></div><div></div><div></div></div> | $\frac{n}{2}$ | 2 |
| Worst case of binary search: <div><div></div><div></div><div></div></div> | $\frac{n}{4} = \frac{n}{2^2}$ | 3 |
| We need to perform about | \vdots | \vdots |
| $\log_2 n$ operations (comparisons). | <div><div></div></div> | $1 = \frac{n}{2^x}$ |
| i.e., binary search has a time complexity of | | $x + 1$ |
| $O(\log_2 n)$ (in worst case). | | $= \log_2 n + 1$ |
| | | $= O(\log_2 n)$ |

Linear search vs. Binary search

Linear search

(Sequential search)

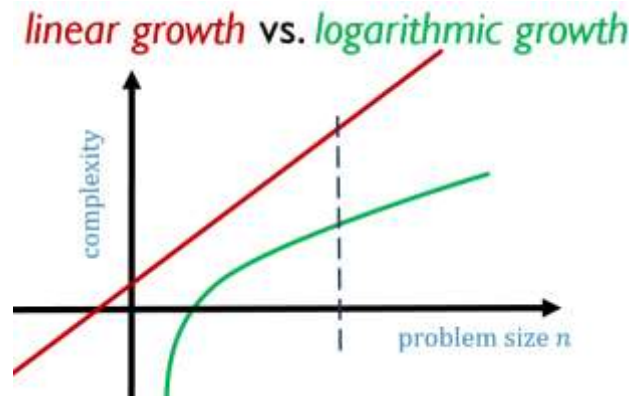
works on any list

best case is to find the first element

Time complexity is $O(n)$
(worst case and average case)

less efficient and less complex

brute force algorithm



Binary search

(Logarithmic search)

works only on **sorted** list

best case is to find the middle element

Time complexity is $O(\log_2 n)$
(worst case and average case)

more efficient and more complex

divide-and-conquer algorithm

Why we need to study Sorting Algorithms

It is estimated that 25~50% of all computing power is used for sorting activities.

Our Sorting Activities

- Common problem: sort a list of values, starting from lowest to highest on basis of size or type.
 - List of exam scores.
 - Search all music files from personal computer based on date of modification.
 - Students names listed alphabetically.
 - Student records sorted by ID.
 - Booking a hotel in holiday destination based on date ,facilities and budget.



Picture: sorting algorithms



The most common types of sorting in data structure are insertion sort, selection sort, bubble sort, quick sort, heap sort, and merge sort.

Insertion sort

We will introduce some sorting methods that take **unsorted list** and return **sorted list** (in ascending order).

Idea of insertion sort:

Step 1: Hold all cards in left hand (unsorted).

Step 2: Take the first card from left hand, and put it in right hand.

Step 3: Take the next card from left hand, **insert** it into the cards in right hand and make sure **all cards in right hand maintain sorted**.

Step 4: Repeat Step 3 until there is no card in left hand. All cards are sorted in right hand.



Insertion sort

Left hand (unsorted)

| | | | | | |
|---|---|---|---|---|---|
| 7 | 9 | 4 | 1 | 3 | 5 |
| | 9 | 4 | 1 | 3 | 5 |
| | | 4 | 1 | 3 | 5 |
| | | | 1 | 3 | 5 |
| | | | | 3 | 5 |
| | | | | | 5 |
| | | | | | |

(empty!)

Right hand

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| 7 | | | | | |
| 7 | 9 | | | | |
| 4 | 7 | 9 | | | |
| 1 | 4 | 7 | 9 | | |
| 1 | 3 | 4 | 7 | 9 | |
| 1 | 3 | 4 | 5 | 7 | 9 |

(sorted)

Algorithm ideas

We need to start by **defining an algorithm**
to insert a new element into a sorted list.

Using this algorithm, we can iterate (**recurse**) over
our original list,
building a longer and longer sorted list.

Insert: pictures

We need to distinguish three cases

• Base case:



• Recursive step option 1:



• Recursive step option 2:



Insert algorithm

Algorithm: **Insert**(*list*, *x*)

Requires: a sorted list of numbers *list*; a number *x*

Returns: a new, sorted list of numbers containing both *list* and *x*.

- 1: if isEmpty(list) then
- 2: return Cons(x, Nil)
- 3: elseif value(list) > x then
- 4: return Cons (x, list)
- 5: else
- 6: return Cons(value(list), **Insert**(tail(list), x))
- 7: endif

➤ **Base case:**



➤ **Recursive step option 1:**



➤ **Recursive step option 2:**



Starting Insertion Sort

Insertion sort requires two lists.

just like you used both hands to sort the cards.

To start off insertion sort, we call a **helper function** that does all the work:

The **helper function** takes two lists:

one list of unsorted or “unprocessed” elements
one sorted list, that we have constructed so far.

Insertion Sort

Algorithm :insertionSort(list)

Requires: a list of numbers list;

Returns: the list of numbers obtained from sorting list.



1: sortHelper(list, Nil)

Why we need a helper here?

SortHelper pictures

(Think of the first argument as the **unsorted** cards in your left hand, and the second argument as the **sorted** cards in you right hand)

Base case:

SortHelper ( and ) = 

Recursive step:

SortHelper ( and ) =

SortHelper ( and insert(x, )



Sort Helper(Insertion Sort)

Algorithm: sortHelper(*list1*, *list2*)

Requires: a list of numbers *list1*; a sorted list *list2*.

Returns: the list of numbers obtained from sorting *list1* & *list2*.

1. **if** isEmpty(*list1*) **then**

Base Case

2. **return** *list2*

3. **else**

Recursive Step

4. **return** sortHelper(tail(*list1*),insert(*list2*,value(*list1*)))

5. **endif**

Algorithm: insertion Sort

Algorithm :insertionSort(list) [Main]

Requires: a list of numbers *list*;

Returns: the list of numbers obtained from sorting *list*.

1: sortHelper(list, Nil)

Algorithm: sortHelper(list1, list2) [Helper]

Requires: a list of numbers *list1*; a sorted list *list2*.

Returns: the list of numbers obtained from sorting *list1* & *list2*.

1. **if** isEmpty(*list1*) **then**
2. **return** *list2*
3. **else**
4. **return** sortHelper(tail(*list1*),insert(*list2*,value(*list1*)))
5. **endif**

Algorithm: Insert(*list*, *x*) [Sub Algorithm]

Requires: a sorted list of numbers *list*; a number *x*

Returns: a new, sorted list of numbers containing both *list* and *x*.

- 1: if isEmpty(*list*) then
- 2: return Cons(*x*, Nil)
- 3: else
- 4: if value(*list*) > *x* then
- 5: return Cons (*x*, *list*)
- 6: else
- 7: return Cons(value(*list*), Insert(tail(*list*), *x*))
- 8: endif
- 9: endif

Calling insert algorithm as
a sub-function.

Related Questions:

HW Exercise :

- Is insertion sort **fast**?
- Are there **other** sorting algorithms?
- Is there a **mathematical proof** that shows what the **fastest** possible sorting algorithm is?

Time complexity (best/worst case)

Suppose we want to sort a list of length n in ascending order, think about:

- What is the best/worst case that may happen?
Explain using a list with $n=5$ (e.g., list with elements 1, 2, 3, 4, 5).
- What is the time complexity for best/worst case (using big O notation)?

Make a note of your answer and bring it to next seminar.



Review Quiz

<https://forms.office.com/r/m4KydkZToY>

Sorting and Searching(Review)



Further Reading

<https://www.sortvisualizer.com/insertionsort/>

<https://visualgo.net/en/sorting?slide=1>