TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Searching Algorithms

Algorithms, Data and Security
A.Y. 2023/24

Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

Why searching and sorting algorithms?

- How do you find someone's phone number in your phone book?
- How do you find your keys when you've misplaced them?
- If a deck of cards has less than 52 cards, how do you determine which card is missing?
- Searching and sorting are common tasks in our daily life!

Why searching and sorting algorithms?

- Searching and sorting are also common tasks in computer programs
- We have well-known algorithms for doing searching and sorting
- We'll look at two searching algorithms and four sorting algorithms
- Considering also their performance (i.e., running times)

Searching algorithms

- Given a collection of values (e.g., a list), the goal of search is to find a target value in this collection or to recognize that the value does not exist in the collection

- We will study two algorithms:
  - Sequential search
  - Binary search
- To visualize how they work, see
  www.cs.usfca.edu/~galles/visualization/Search.html
  binary-search-visualization.netlify.app

Sequential search

- The simplest search algorithm
  - We look at each item in the list in turn, quitting once we find an item that matches the target value or once we reach the end of the list
  - Also known as linear search
- Find whether a value x is contained in list L

algorithm `SequentialSearch`$(item\ x,\ list\ L) \rightarrow Boolean$

```
n = |L|                                   ▷ list L has n items
for i = 1 to n do
    if L[i] = x then return found
    end if
end for
return not found
```

Sequential search: Running time

- Sequential search requires O(n) time (comparisons) in the worst case
  - Worst case means the most number of comparisons necessary to find the target value
  - In our case, it occurs when the target value is in the last slot in the list, or is not in the list. The number of comparisons is equal to the size of the list (say n)

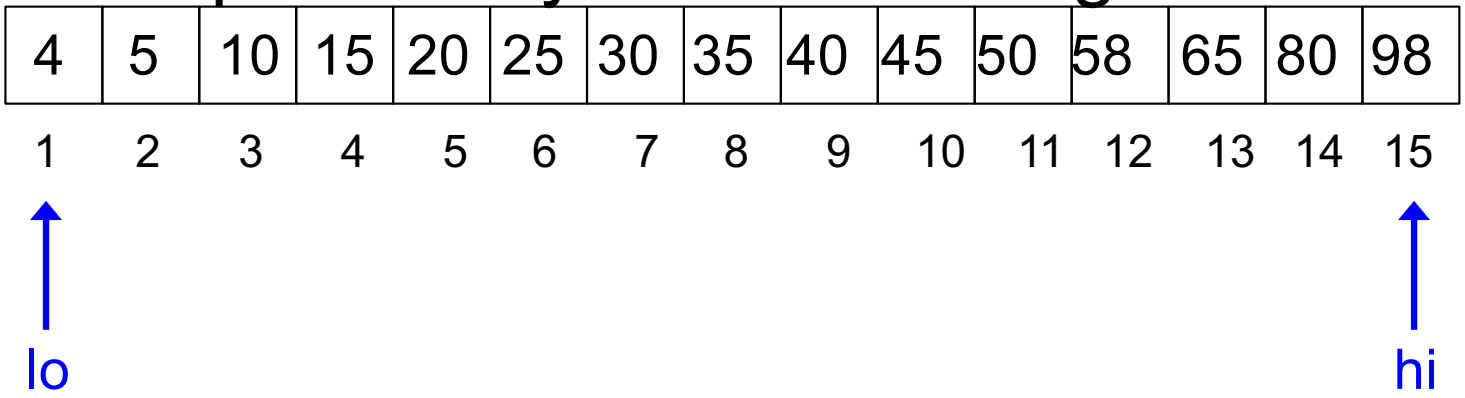
- Can we do better?
- Yes, if the list L is sorted

Binary search

- Input: sorted list and target value
- Idea: each time we compare the target value to the middle element of the list, we eliminate half of the list and continue the search on the remaining half, until we either find the target value or determine that it is not in the list

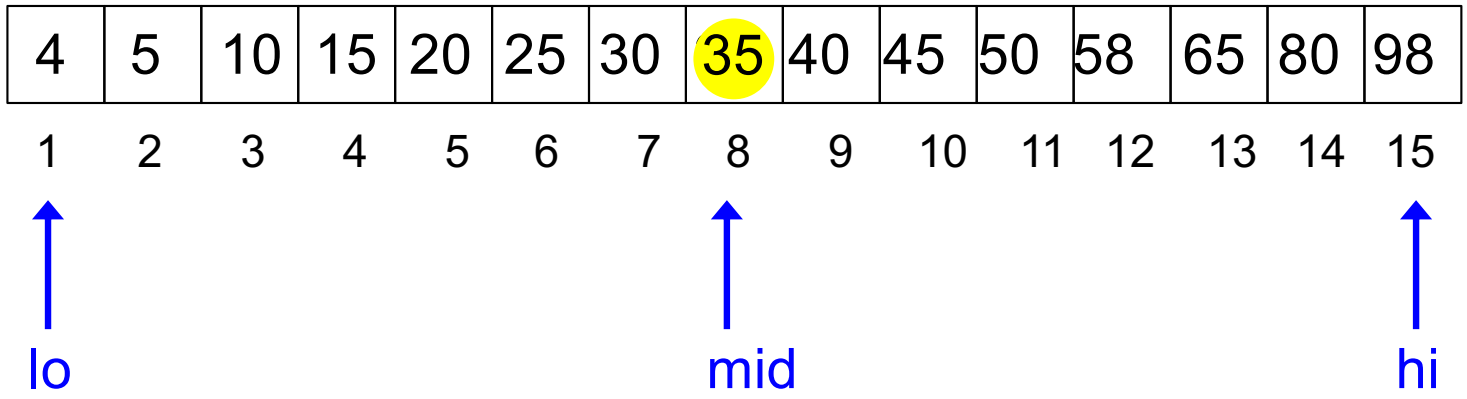
Binary search

- Given target value and sorted list (or array) L as input, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo] ≤ value ≤ L[hi]
  - lo and hi are the lower and upper end of the portion of the list that we are considering
  - mid is the midpoint of the list
- Example: binary search for target value 20



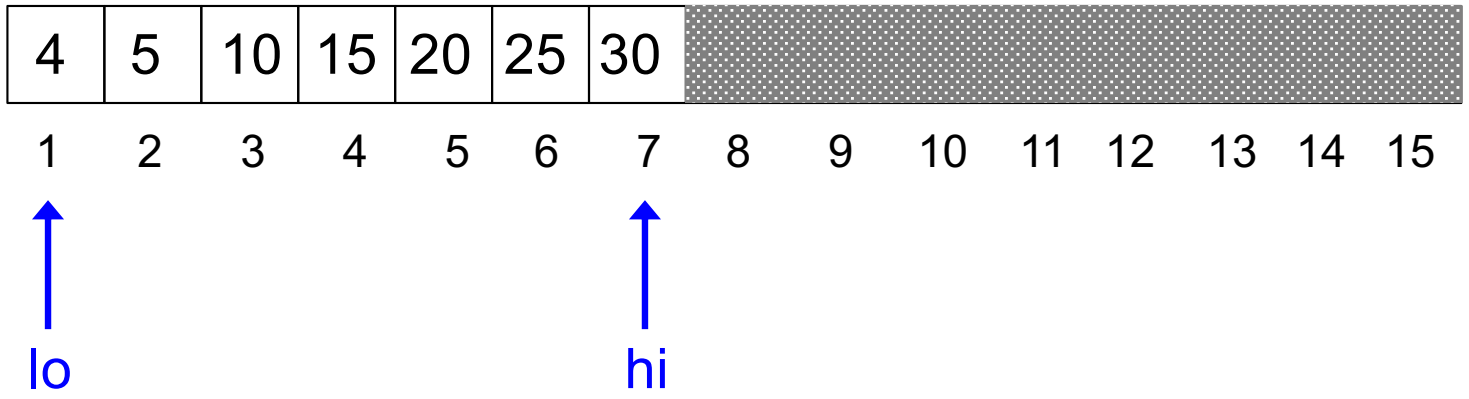
Binary search: example

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



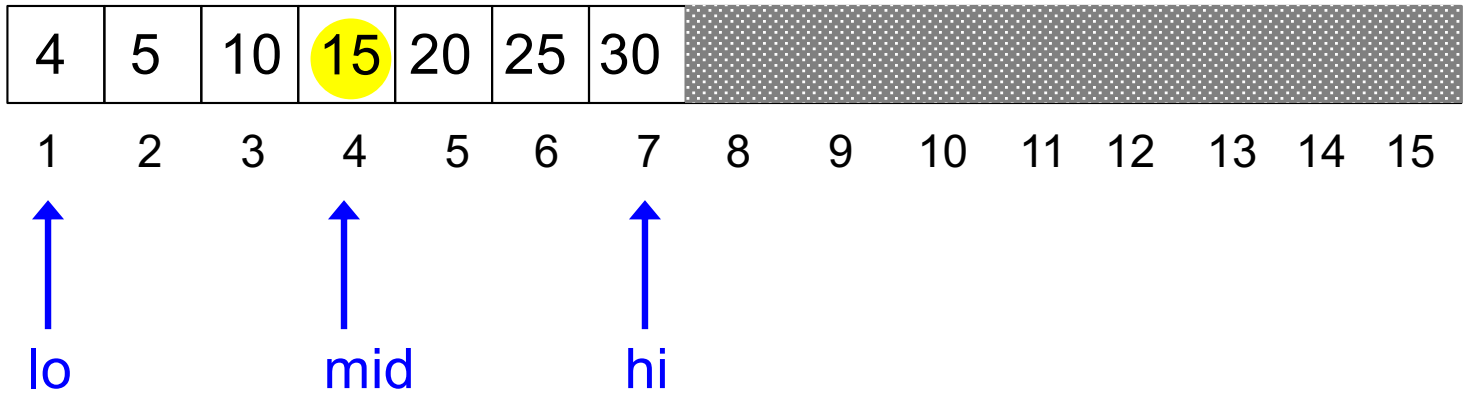
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



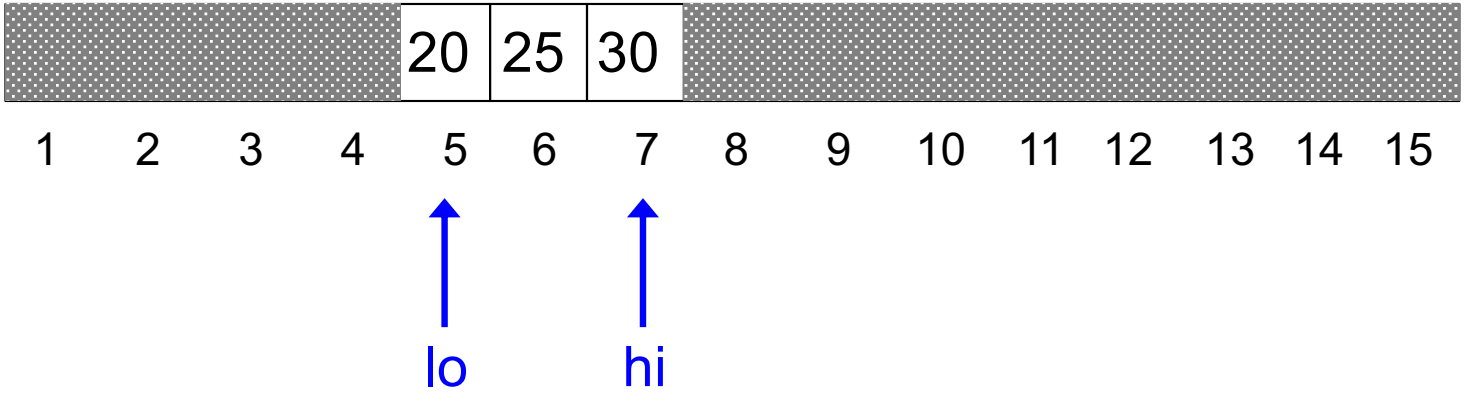
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



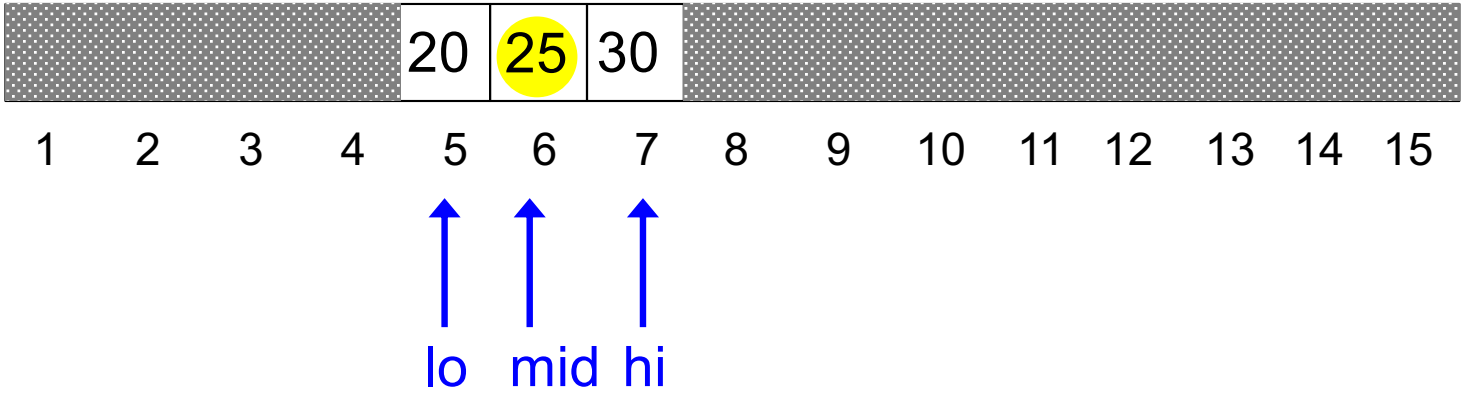
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



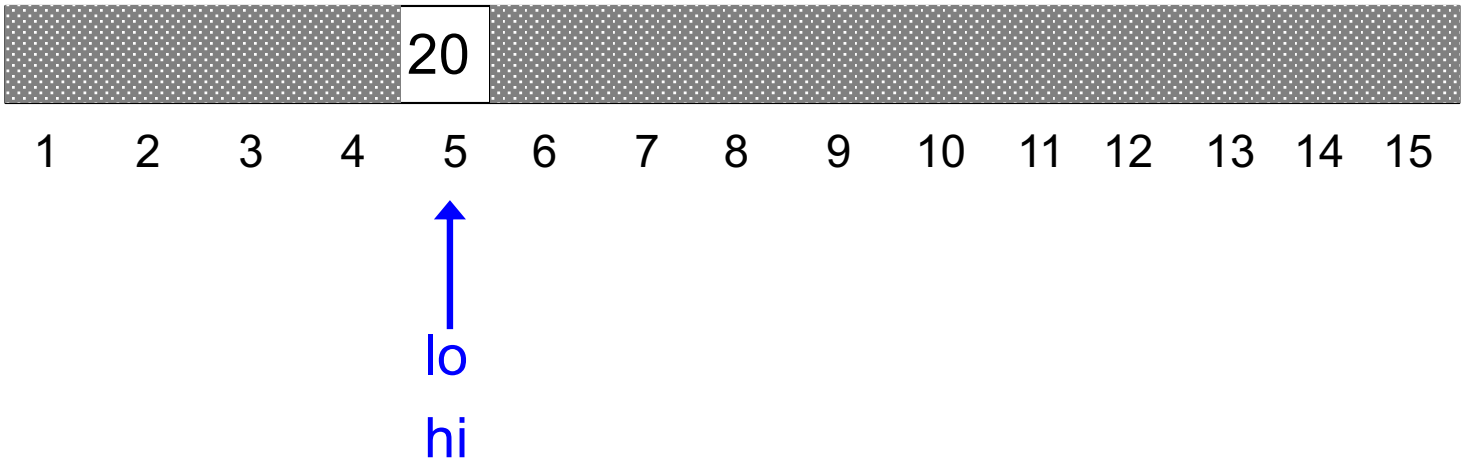
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



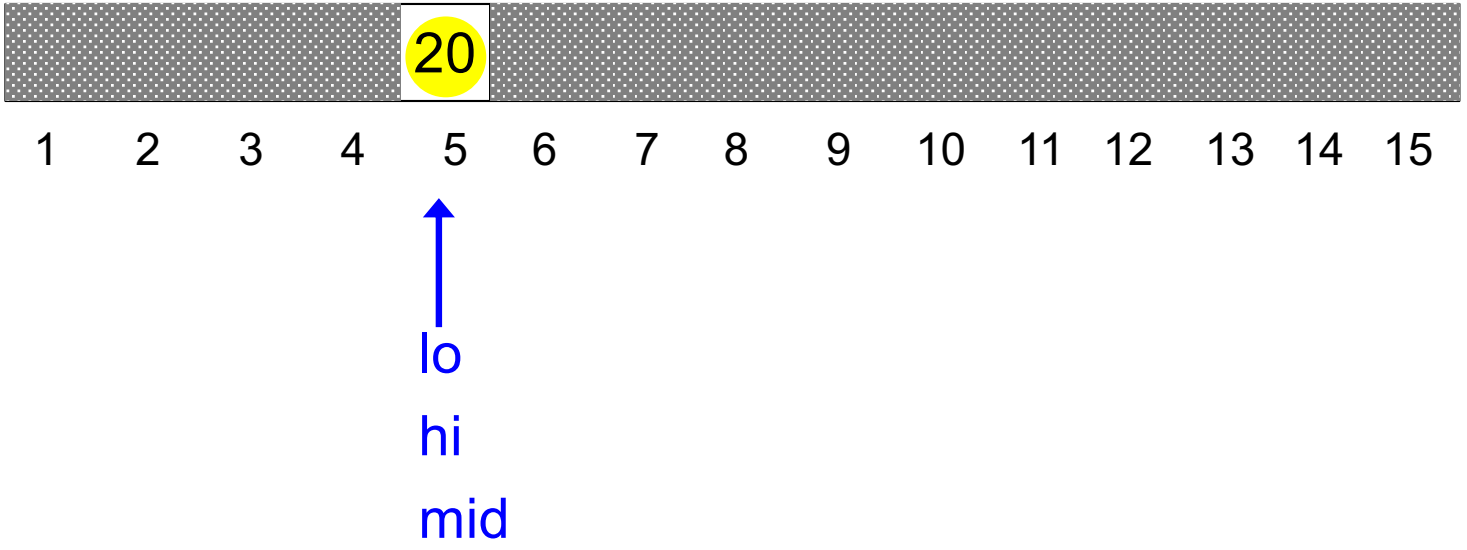
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



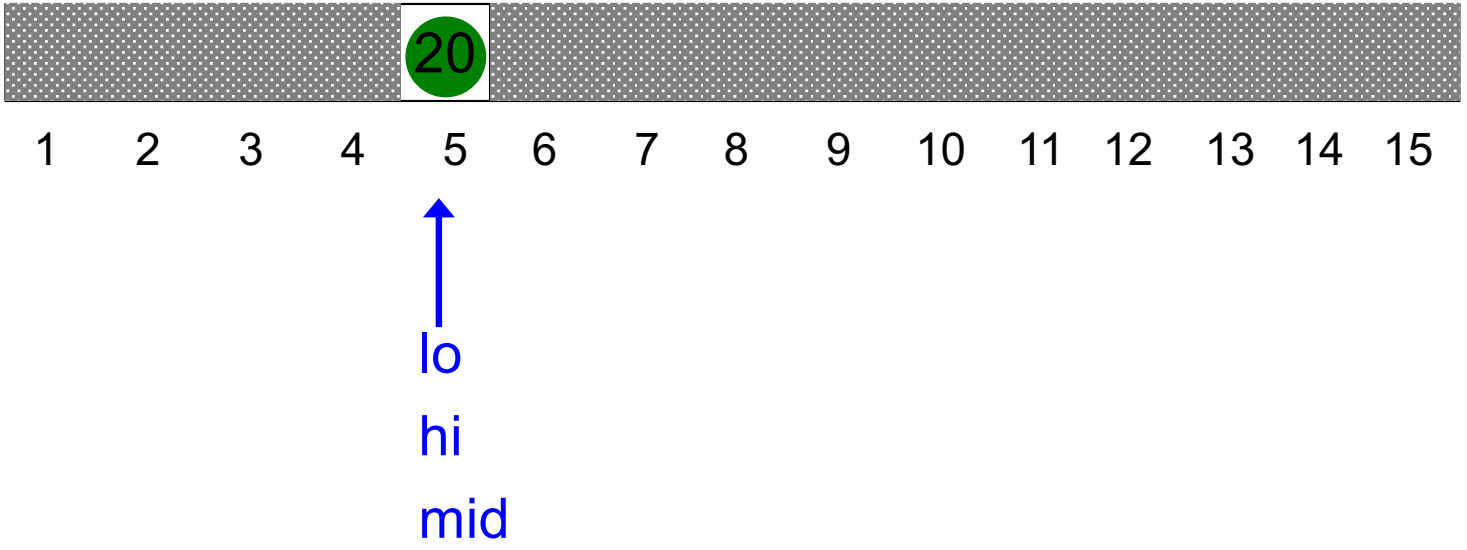
Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



Binary search

- Given target value and sorted list L, find index i such that L[i]=value, or report that no such index exists
- Algorithm maintains L[lo]≤value≤L[hi]
- Example: binary search for target value 20



Binary search: Algorithm

algorithm `BinarySearch`$(item\ x,\ list\ L) \rightarrow Boolean$

$n = |L|$
if $n = 0$ **then return** not found
end if
$i = \lceil n/2 \rceil$
if $L[i] = x$ **then**
    return found
else if $L[i] > x$ **then**
    return BinarySearch$(x, L[1; i-1])$
else
    return BinarySearch$(x, L[i+1; n])$
end if

Binary search: Algorithm

The list in input is sorted

algorithm `BinarySearch`$(item\ x,\ list\ L) \rightarrow Boolean$

$n = |L|$ ← length of list (number of elements in the list)

if $n = 0$ **then return** not found

end if

$i = \lceil n/2 \rceil$ ← midpoint of list

if $L[i] = x$ **then**

return found

else if $L[i] > x$ **then**

return BinarySearch$(x, L[1; i-1])$ ← recursive calls, left part of list

else

return BinarySearch$(x, L[i+1; n])$ ← right part of list

end if

Running time?

Binary search: Running time

- Let T(n) be number of comparisons done by BinarySearch to find an item x in a sorted list of size n

  $T(n) = 1 + T(n/2)$

  $T(1) = 1$

- T(1) = 1 (if the list has just one element, only 1 comparison is needed)
- It is a recurrence. How do we solve it?

Binary search: Running time

- We have to solve

  $T(n) = T(n/2) + 1$

- If this is true, then we can also say

  $T(n/2) = T(n/4) + 1$

- Putting this back into the recurrence, we obtain:

  $T(n) = T(n/2) + 1 = T(n/4) + 1 + 1$

- i.e.,

  $T(n) = T(n/4) + 2$

Binary search: Running time

- We have

  $T(n) = T(n/2) + 1 = T(n/4) + 2$

- We can also say

  $T(n/4) = T(n/8) + 1$

- Putting this back into the recurrence, we obtain:

  $T(n) = T(n/4) + 2 = T(n/8) + 1 + 2$

- i.e.,

  $T(n) = T(n/8) + 3$

Binary search: Running time

- We can go on like this:

  $T(n) = T(n/8) + 3 = T(n/16) + 4 = ...$
- In general, for k = 1, 2, 3,...

  $T(n) = T(n/2^k) + k$
- When do we stop? Can't go on forever…

Binary search: Running time

- We can stop when k is such that $n/2^k = 1$
  - i.e., $n = 2^k$, i.e., $k = \log_2 n$
- For this value of k:

  $T(n) = T(n/2^k) + k = T(1) + \log_2 n = 1 + \log_2 n$
- Namely:

  $T(n) = 1 + \log_2 n = O(\log n)$

Binary search: Running time

In summary:

- The number of comparisons T(n) done by binary search to find an item in a sorted list of size n is described by the recurrence:

  T(n) = T(n/2) + 1

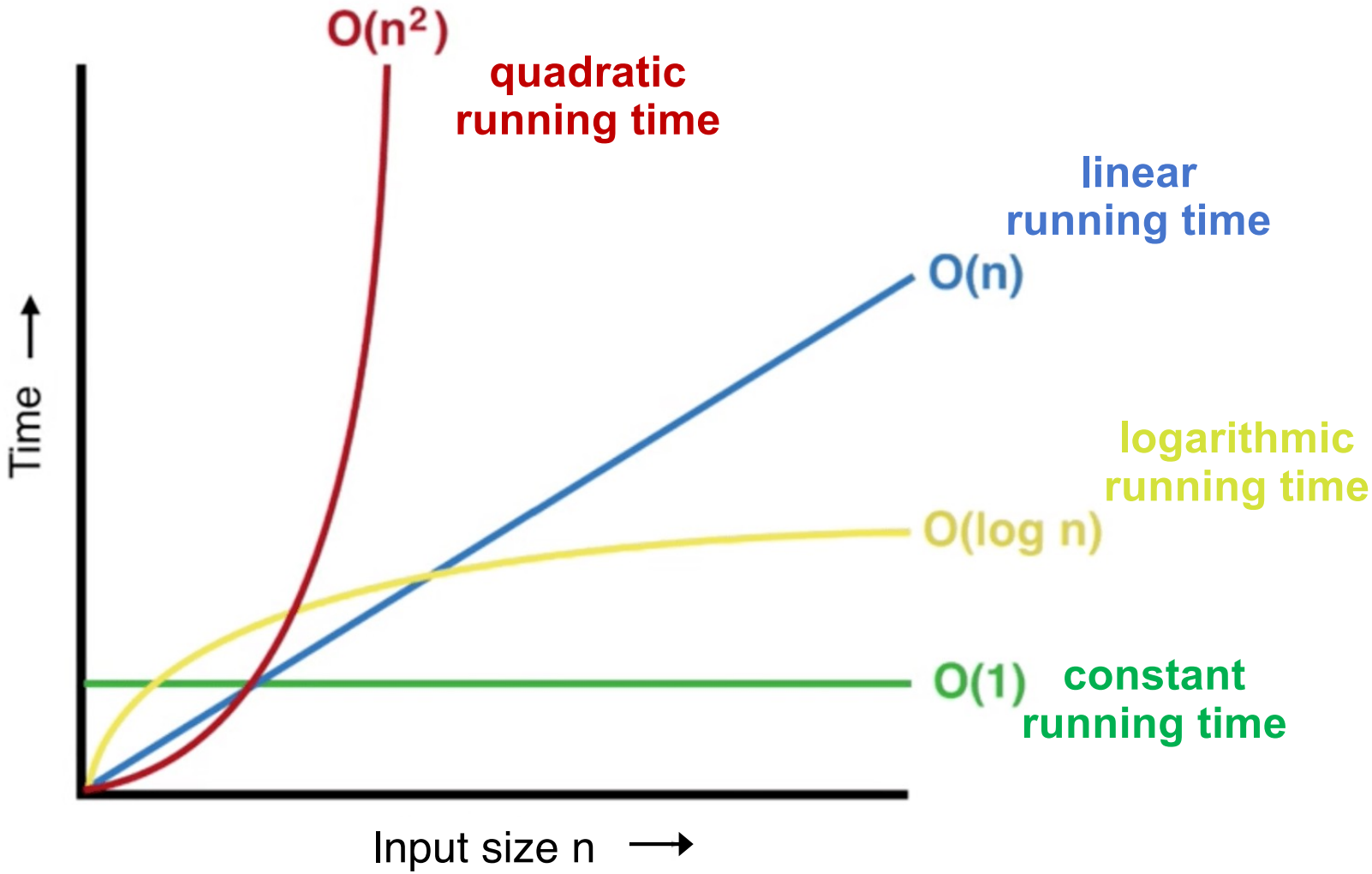
- The solution of this recurrence is

  T(n) = O(log n)

- Binary search requires running time **O(log n)**
  - Better than sequential search, which is O(n)!

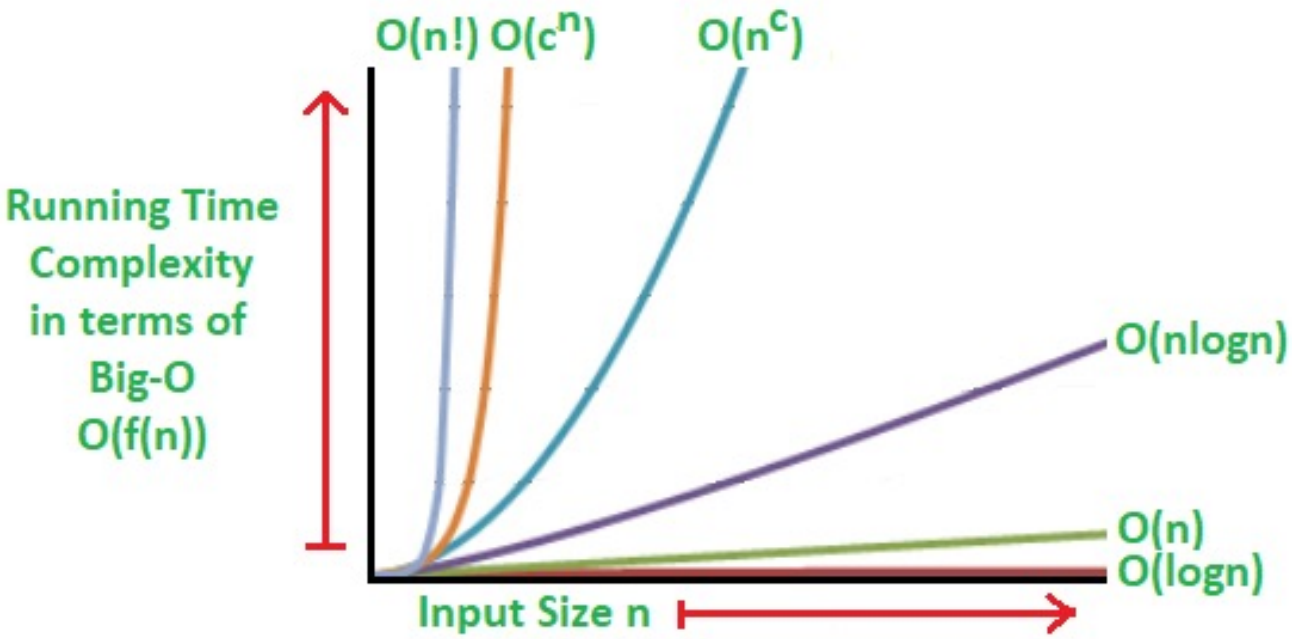
Big O notation: common cases

- Some common Big O notations:



Big O notation: common cases

- More Big O notations:



O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

Big O notation: common cases

- Constant algorithm – O(1)
  - The fastest possible running time: the algorithm always takes the same amount of time to execute, regardless of the input size; ideal but rarely achievable
- Logarithmic algorithm – O(log n)
  - Running time grows logarithmically in proportion to n
  - E.g., binary search
- Linear algorithm – O(n)
  - Running time grows directly in proportion to n
  - E.g., sequential search, `fibonacci3`
- Superlinear algorithm – O(n log n)
  - Running time grows faster than n, still practical

Big O notation: common cases

- Polynomial algorithm – $O(n^c)$
  - Running time grows quicker than previous all
- Exponential algorithm – $O(c^n)$
  - Running time grows even faster than polynomial algorithm
  - E.g., `fibonacci2`
- Factorial algorithm – O(n!)
  - Running time grows the fastest and the algorithm becomes quickly unusable for even small values of n

Take-away

- We described two searching algorithms:
  - Sequential search: O(n)
  - Binary search (sorted input): O(log n)

n	10	100	1000	10^6	10^9
$\log_2 n$	~ 3.3	~ 6.65	~ 10	~ 20	~ 30

- If we have to search through trillion ($n=10^{12}$) of items, and each step takes 10 nanosec. (10^{-8} sec.):
  - n steps take: $10^{12} \times 10^{-8}$ sec $= 10^4$ sec ~ 3 hours!
  - $\log_2 n$ steps take: $\log_2(10^{12}) \times 10^{-8}$ sec =
    $= \log_2((10^3)^4) \times 10^{-8}$ sec ~ $\log_2((2^{10})^4) \times 10^{-8}$ sec =
    $= 40 \times 10^{-8}$ sec = 400 nanosec!

References

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms, 4th ed., MIT Press, 2022
- C. Demetrescu, I. Finocchi, G. F. Italiano. Algoritmi e Strutture Dati, Mc-Graw Hill, 2008 (in Italian)