TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Searching Algorithms

Algorithms, Data and Security
A.Y. 2025/26

Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

Why searching and sorting algorithms?

- How do you find a contact in your phone or a particular email?
- If a deck of cards is missing a card, how do you figure out which one it is?
- Searching and sorting are common tasks in everyday life!

Why searching and sorting algorithms?

- Searching and sorting are also common tasks in computer programs
- We have well-known algorithms to perform these tasks efficiently
- We will look at two searching algorithms and four sorting algorithms and also consider their performance (i.e., running times)

Searching algorithms

- Given a collection of values, the goal of search is to find a target value in the collection or to determine that the value does not exist
- We will study two searching algorithms:
  - Sequential search
  - Binary search
- We assume the values are stored in a list (or array) and are numerical
  - These algorithms also work for non-numerical data
- To visualize how these algorithms work, see https://www.cs.usfca.edu/~galles/visualization/Search.html

Sequential search

- The simplest searching algorithm
  - Like scanning a book index for a specific topic
  - We look at each item in the list in turn, stopping when we find the target or 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*

$n = |L|$ $\qquad\triangleright$ 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 largest number of comparisons necessary to find the target
  - The worst case occurs when the target is in the last slot of the list, or is not in the list
  - In this case, the number of comparisons is equal to the size of the list (n)
- Can we do better?
- Yes, if the list is sorted

Binary search

- Similar to finding a word in a dictionary
- Input: a sorted list and a target value
- Idea: each time we compare the target value to the middle element of the list
- If it matches, we are done
- Otherwise, we eliminate half of the list and continue the search on the remaining half
- We repeat until we find the target value or determine it is not in the list

Binary search

- The algorithm uses the middle element of a sorted list
- It works by repeatedly dividing the list in half until the target value is found or the list is empty
- At each step
  - Compare the target value to the middle element
  - If they are equal, the search terminates successfully
  - Otherwise, eliminate the half in which the target cannot lie
  - Repeat the process on the remaining half
- Continue until the target value is found or determined not to be in the list

Binary search: example

- Binary search for target value 20

4	5	10	15	20	25	30	35	40	45	50	58	65	80	98
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

4	5	10	15	20	25	30	35	40	45	50	58	65	80	98
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

4	5	10	15	20	25	30								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

4	5	10	15	20	25	30								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

				20	25	30								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Binary search: example

- Binary search for target value 20

				20	25	30								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

				20										
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

				20										
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

				20										
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Binary search: Algorithm

- Given target value x and sorted list L as input, find if x is in the list or not

$$
\begin{array}{l}
\textbf{algorithm } \texttt{BinarySearch}(\mathit{item}\ x,\ \mathit{list}\ L) \to \mathit{Boolean} \\
\quad n = |L| \\
\quad \textbf{if } n = 0 \textbf{ then return } \text{not found} \\
\quad \textbf{end if} \\
\quad \mathit{mid} = \lceil n/2 \rceil \\
\quad \textbf{if } L[\mathit{mid}] = x \textbf{ then} \\
\qquad \textbf{return } \text{found} \\
\quad \textbf{else if } L[\mathit{mid}] > x \textbf{ then} \\
\qquad \textbf{return } \text{BinarySearch}(x, L[1; \mathit{mid}-1]) \\
\quad \textbf{else} \\
\qquad \textbf{return } \text{BinarySearch}(x, L[\mathit{mid}+1; n]) \\
\quad \textbf{end if}
\end{array}
$$

Binary search: Algorithm

List is sorted

$$
\begin{array}{l}
\textbf{algorithm } \texttt{BinarySearch}(\mathit{item}\ x,\ \mathit{list}\ L) \to \mathit{Boolean} \\
\quad n = |L| \quad \leftarrow \text{length of list (number of elements in the list)} \\
\quad \textbf{if } n = 0 \textbf{ then return } \text{not found} \\
\quad \textbf{end if} \quad \leftarrow \text{midpoint of list} \\
\quad \mathit{mid} = \lceil n/2 \rceil \\
\quad \textbf{if } L[\mathit{mid}] = x \textbf{ then} \\
\qquad \textbf{return } \text{found} \\
\quad \textbf{else if } L[\mathit{mid}] > x \textbf{ then} \qquad \text{recursive calls} \\
\qquad \textbf{return } \text{BinarySearch}(x, L[1; \mathit{mid}-1]) \quad \leftarrow \text{left half of list} \\
\quad \textbf{else} \\
\qquad \textbf{return } \text{BinarySearch}(x, L[\mathit{mid}+1; n]) \quad \leftarrow \text{right half of list} \\
\quad \textbf{end if}
\end{array}
$$

Running time?

Binary search: Running time

- Let T(n) be the 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:

$$\mathbf{T(n) = T(n/2) + 1}$$

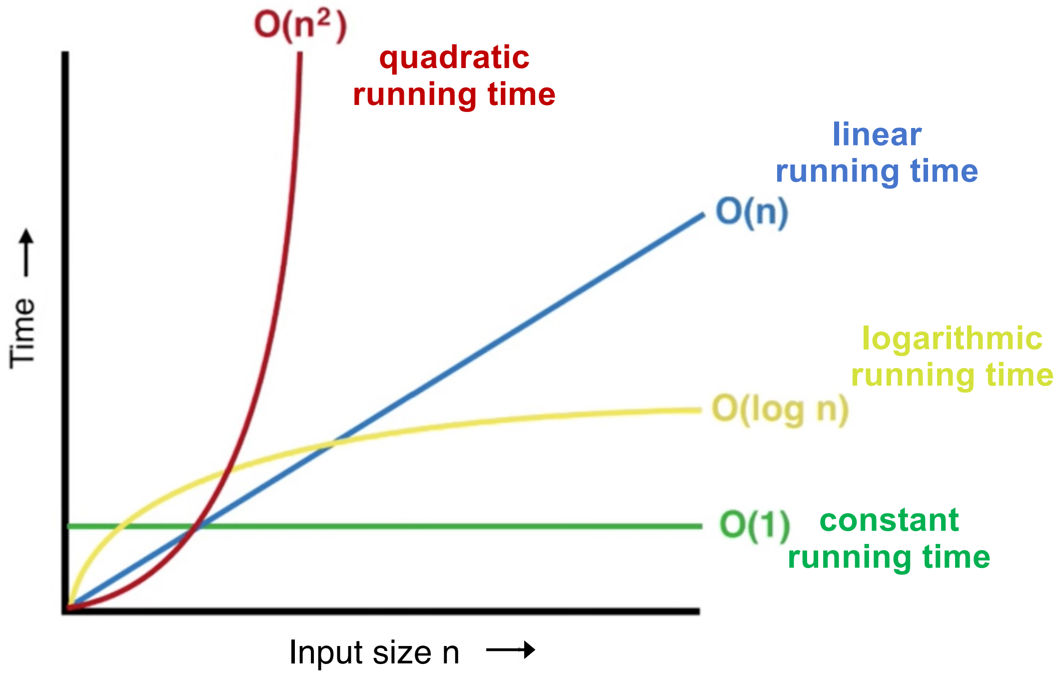
- The solution to this recurrence is

$$\mathbf{T(n) = O(\log n)}$$

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

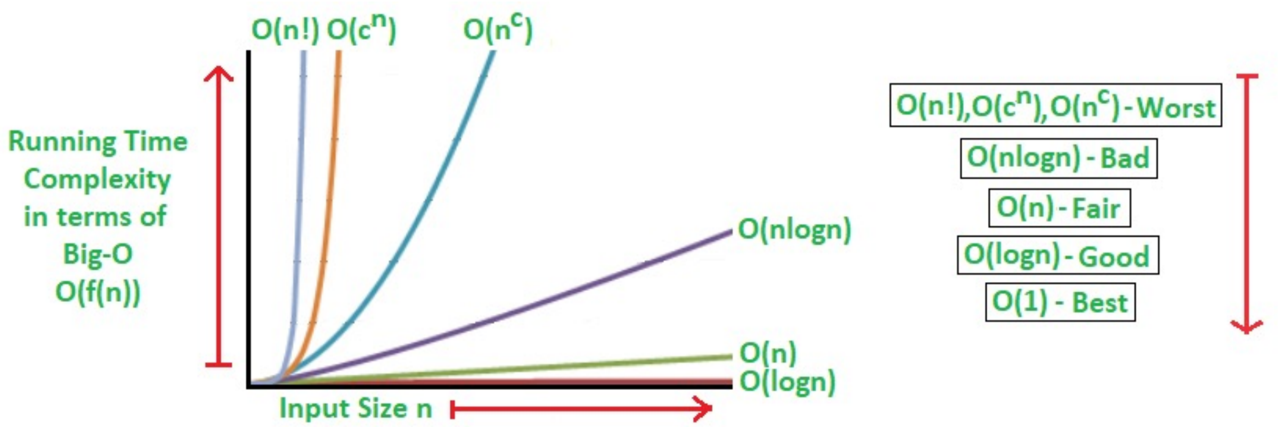
Big O notation: common cases

- Some common Big O notations:



Big O notation: common cases

- Additional Big O notations:



Big O notation: common cases

- Constant time – O(1)
  - The fastest possible running time: the algorithm always takes the same amount of time, regardless of input size; ideal but rarely achievable in practice
- Logarithmic time – O(log n)
  - Running time grows logarithmically with 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 time – $O(n^c)$
  - Running time grows faster than previous cases
  - Still practical for small or moderate input sizes, depending on c
- Exponential time – $O(c^n)$
  - Running time grows much faster than polynomial algorithm
  - E.g., `fibonacci2`
  - Impractical even for moderate n
- Factorial time – O(n!)
  - Running time grows extremely fast and the algorithm becomes 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: $10^{12}\times10^{-8}$ sec $=10^4$ sec ~ 3 hours!
  - $\log_2 n$ steps take: $\log_2(10^{12})\times10^{-8}$ sec =
    $= \log_2((10^3)^4) \times10^{-8}$ sec ~ $\log_2((2^{10})^4)\times10^{-8}$ sec =
    $= 40\times10^{-8}$ sec = 400 nanosec!

Exercise

- Consider the list [11, 6, 4, 21, 8] and perform sequential search to find the target value 8
- Consider the sorted list [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] and perform binary search to find the target value 14
  - Write also the sequence of recursive calls BinarySearch(14, …)

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 3ª edizione, Mc-Graw Hill, 2025 (in Italian)
- Algorithms: Binary Search https://www.youtube.com/watch?v=P3YID7liBug