SUPSI

B3209E
Algorithm Design

Chapter 02: Basics of Algorithm Analysis

24 September 2025

Omran Ayoub
omran.ayoub@supsi.ch

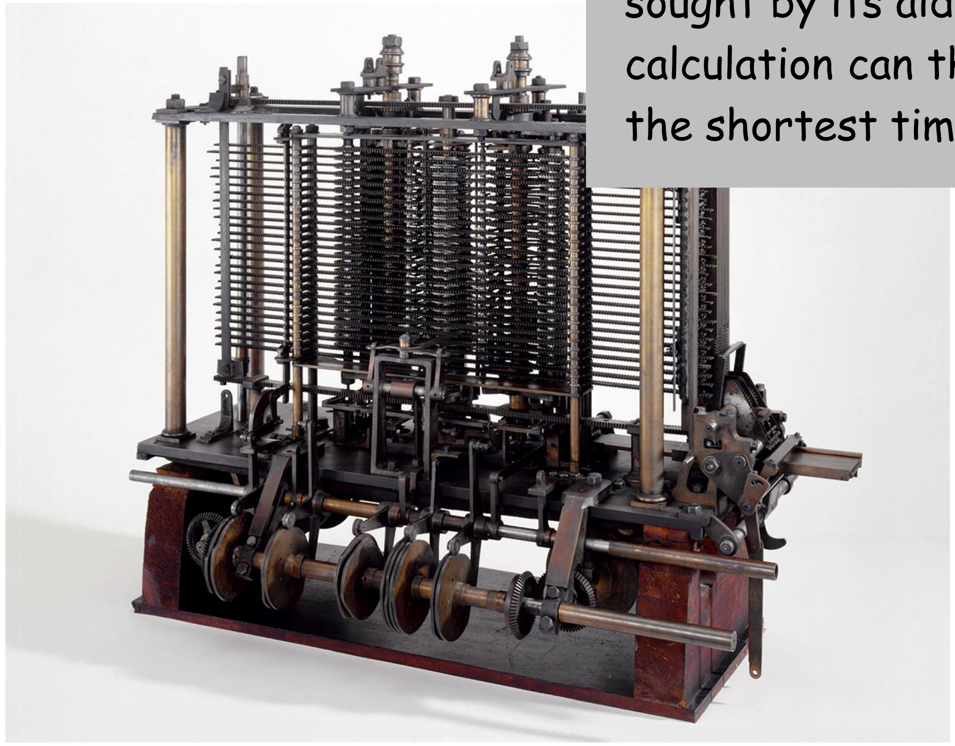
Outline

- Computational Tractability
- Asymptotic Order of Growth
- Survey of Common Running Times

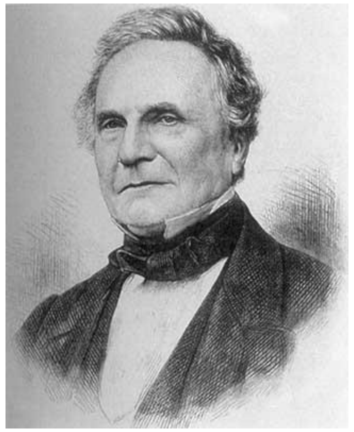
Outline

- **Computational Tractability**
- Asymptotic Order of Growth
- Survey of Common Running Times

Computational Tractability



As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)

Computational Tractability

- Why design algorithms? We have brute-force
- Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
  - Typically takes 2^N time or worse for inputs of size N.
  - Running time is too long: 2^N time or worse for inputs of size N
  - Unacceptable in practice, it is not efficient. But what is efficient?

n! for stable matching
with n men and n women

We will focus on efficiency
in terms of running time

- But generally, when is algorithm A better than algorithm B?
  - Algorithm A runs faster
  - Algorithm A requires less space to run

There exist a space/time trade off but
this is beyond our discussion

Computational Tractability

- Attempting to define efficiency.

- **Attempt #1**. An algorithm is efficient if, when implemented, it runs quickly on real input instances.
  - **But**, implemented where? And how? How well it was coded?
  - Also, what is a 'real' input instance?

- **Worst case running time**. Obtain bound on <u>largest possible running time</u> of algorithm on input of a given size N.

$\nwarrow$

Always important

- **Average case running time**. Obtain bound on running time of algorithm on <u>random</u> input as a function of input size N.

Not that important

  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.

Computational Tractability

- **Attempt #2**. An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.
  - Ok, this is nice, we improve with respect to brute-force.
  - **But** what is "qualitatively better performance"?

- **Alternative**: Polynomial time as a definition of efficiency.

There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c\,N^d$ steps.

- Search spaces for natural combinatorial problems grow exponentially; when input increases by 1, the number of possibilities increases multiplicatively
  - Example: 2^N for input N, 2^{N+1} for input N+1

- Desirable scaling property. When input increases by 1, the algorithm should only slow down by some constant factor
- **Attempt #3**. An algorithm is efficient if it has a polynomial running time

Computational Tractability

- **Polynomial time as a definition of efficiency.**
- Def. An algorithm is efficient if its running time is polynomial.

- Justification: It really works in practice!
  - Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice
  - In practice, poly-time algorithms that people develop almost always have low constants and low exponents
  - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem

- Exceptions.
  - Some poly-time algorithms do have high constants and/or exponents and are useless in practice
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare

Computational Tractability

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table taken from More Programming Pearls, p. 82, 400MhZ Pentium II scaled up
one step takes one nanosecond

Never underestimate Pentium II

Outline

- Computational Tractability
- **Asymptotic Order of Growth**
- Survey of Common Running Times

Asymptotic Order of Growth

- Goal. Examine how algorithm’s running time grows as the problem instance grows (as input grows)
- We need a framework to talk about the concept of growth of running time
- We will be counting the number of pseudocode steps that are executed by the algorithm

But what is the notion of a step?

- Also, we would like to express the running time of an algorithm by something like:
  - T(n) is in the order of f(n), T(n) is O(f(n))

Asymptotic Order of Growth

- Three kinds of bounds:
  - Upper bound
  - Lower bound
  - Tight bound

- Ex: $T(n) = pn^2 + qn + r$.
  - For $n \geq 1$, $qn \leq qn^2$ and $r \leq rn^2$
    - Then, $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 \leq (q + p + r)\, n^2 \leq c\, n^2$

- $T(n) \leq c\, n^2$ $\rightarrow$ T is asymptotically upper bounded by n^2
- **Upper bounds**. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $\mathbf{T(n) \leq c \cdot f(n)}$.

Asymptotic Order of Growth

- Ex: $T(n) = pn^2 + qn + r$.
  - For $n \geq 1$
    - $T(n) = pn^2 + qn + r \geq pn^2 \rightarrow$ T is asymptomatically lower bounded by n^2

- **Lower bounds**. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $\mathbf{T(n) \geq c \cdot f(n)}$**.**

- **Tight bounds**. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Asymptotic Order of Growth

- Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

- Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

- Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

- Ex: $T(n) = 32n^2 + 17n + 32$.
  - $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
  - $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Asymptotic Order of Growth

- Properties of asymptotic growth. **(2.1 to 2.6)**

- Transitivity.
  - If f = O(g) and g = O(h) then f = O(h).
  - If f = Ω(g) and g = Ω(h) then f = Ω(h).
  - If f = Θ(g) and g = Θ(h) then f = Θ(h).

- Additivity.
  - If f = O(h) and g = O(h) then f + g = O(h).
  - If f = Ω(h) and g = Ω(h) then f + g = Ω(h).
  - If f = Θ(h) and g = Θ(h) then f + g = Θ(h).

Asymptotic Order of Growth

- Asymptotic bounds of some common functions.
  - Polynomials
  - Logarithms
  - Exponentials

- Polynomials. $a_0 + a_1 n + \dots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
- Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n. **(2.7)**
- Algorithms with running time $O(n^2)$ and $O(n^3)$ are polynomial

Asymptotic Order of Growth

- $\log_b n$ is the number x such that $b^x = n$
- How $\log_b n$ grows?

- Ex. $1 + \log_2 n$ is the number of bits needed to represent n
  - If $n = 64$, $1 + \log_2 64 = 6$
  - If $n = 256$, $1 + \log_2 256 = 8$

- $\log_b n$ grows slowly..

- **Logarithms**. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$. **(2.8)**
- For every $x > 0$, $\log n = O(n^x)$.

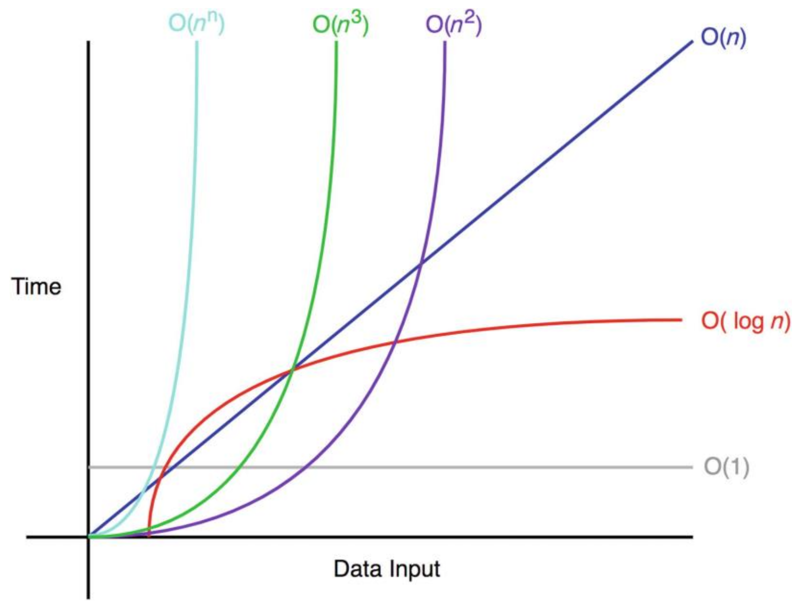
log grows slower than every polynomial

can avoid specifying the base

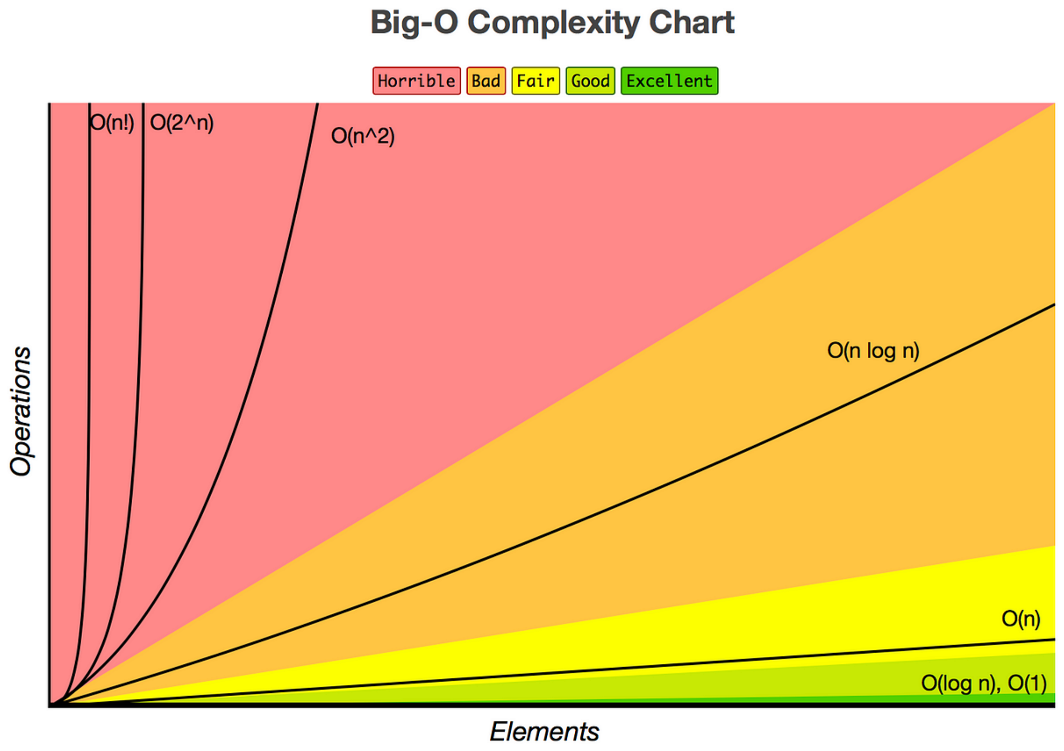
Asymptotic Order of Growth

- **Exponentials**. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$. **(2.9)**

every exponential grows faster than every polynomial



From the book: JavaScript Data Structures and Algorithms
Ref.: Bae S. (2019) Big-O Notation. In: JavaScript Data Structures and Algorithms. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3988-9_1



Outline

- Computational Tractability
- Asymptotic Order of Growth
- **Survey of Common Running Times**

Survey of Common Running Times

- There are styles of analysis of algorithms that occur frequently: O(n), O(n log n), O(n^2)

- Our goal is to learn how to recognize these common styles of analysis and some of the typical approaches that lead to them

- When approaching a problem, it often helps to think of two bounds:

1) Desirable running time: running time we hope to achieve
2) Size of the problem's natural search space: running time of brute-force algorithm of the problem

Survey of Common Running Times

- Linear time. O(n). Running time is at most a constant factor times the size of the input.

- Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.
- Performed by processing the input in a single pass (constant amount of time per input)

```
max ← $a_1$
for i = 2 to n {
   if ($a_i$ > max)
      max ← $a_i$
}
```

- Linear time. O(n). Running time is at most a constant factor times the size of the input.
- Merging two sorted lists. Sort two lists A and B of n numbers each, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ each sorted ascending order, in one list.

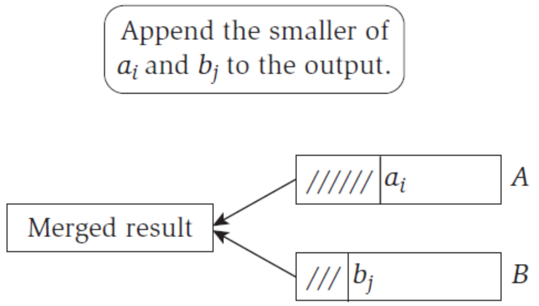
- We can put the lists together and run a sorting algorithm. Wasteful!

Survey of Common Running Times

- Linear time. O(n). Running time is at most a constant factor times the size of the input.
- Merging two sorted lists. Sort two lists A and B of n numbers each, $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ each sorted in ascending order, in one list.

- We can Create a pointer to front element of each list, compare values of front elements and append smaller one to output

```
i = 1, j = 1
while (both lists are nonempty) {
   if ($a_i \le b_j$) append $a_i$ to output list and increment i
   else($a_i \le b_j$)append $b_j$ to output list and increment j
}
append remainder of nonempty list to output list
```



- Claim. Merging two lists of size n takes O(n) time.
- Pf. After each comparison, the length of output list increases by 1.

Survey of Common Running Times

- **O(n log n) time**. Arises in divide-and-conquer algorithms.
- Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.
- Largest empty interval. Given n time-stamps x_1, …, x_n on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?
- O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

- **Quadratic time**. $O(n^2)$. Enumerate all pairs of elements. Two nested loops.

- **Cubic time**. $O(n^3)$. Enumerate all triples of elements. Three nested loops.

- **Polynomial Time**. $O(n^k)$.

- **Exponential Time**. $O(k^n)$.

Survey of Common Running Times

- 2^n and n!

- 2^n is the total number of n-element sets. Example: number of sets we can find with 1, 2 and 3 is 8: {empty, 1, 1 and 2, 1 and 3, 1 and 2 and 3, 2, 2 and 3, 3}

- n! arises in two ways:
  - 1) Number of ways to match n items with another n items. Example: number of possible matchings we can have with the stable matching problem with n men and n women.
  - 2) When we arrange n items. Number of ways to arrange n items. Example: there are 6 ways to arrange three numbers 1, 2 and 3.

Survey of Common Running Times

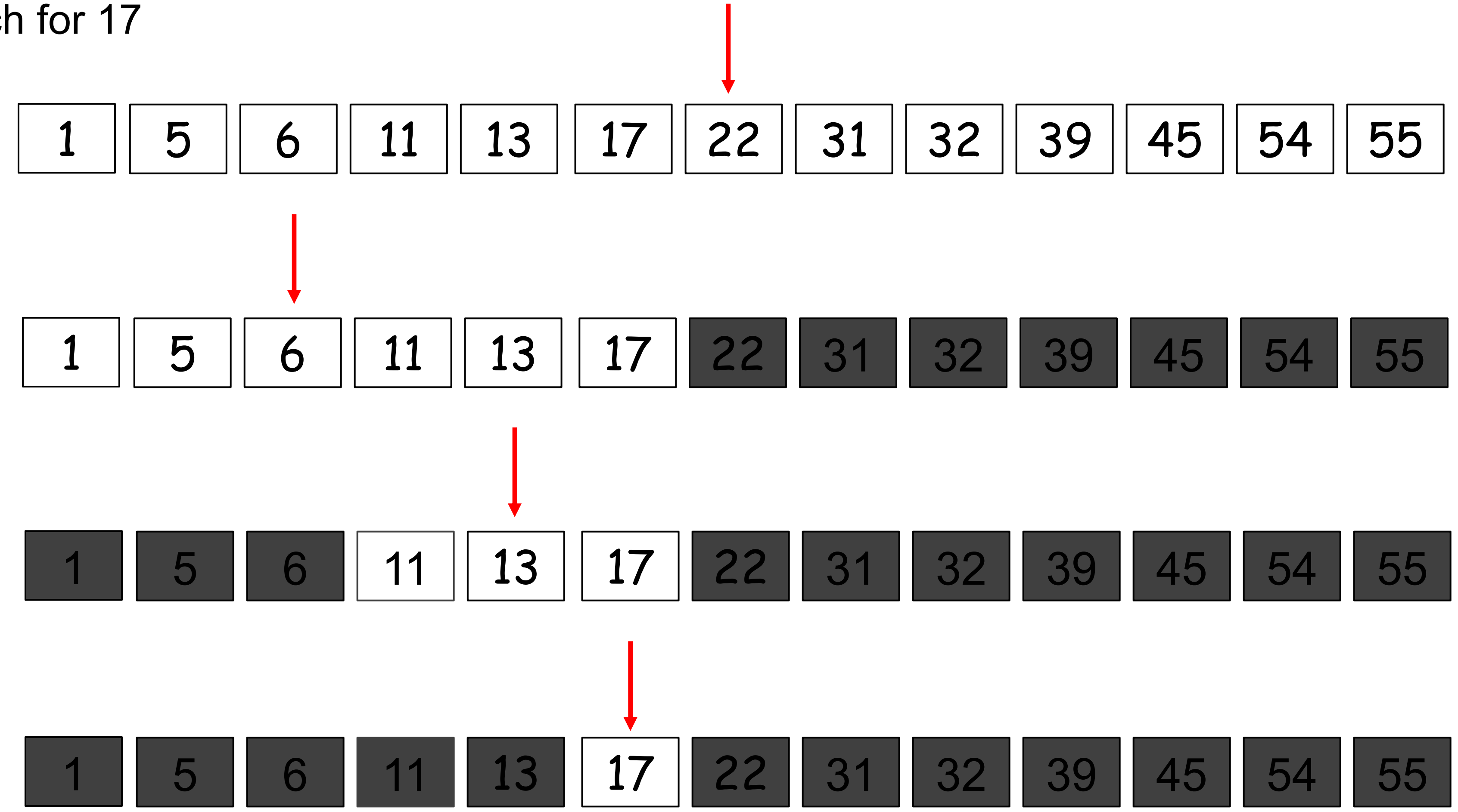
- Sublinear Time. Running time asymptotically smaller than linear!

- But how? Reading the input requires linear time…
  - These situations arise in a model where the input is queried rather than read

- Ex. Binary Search algorithm. O(log n).
  - Given: sorted array A of n numbers
  - Goal: determine if number p belongs to A

Survey of Common Running Times

- Sublinear Time. Running time asymptotically smaller than linear!
- Let us search for 17



Summary

- **Polynomial time as a definition of efficiency.**
  - Desirable scaling property. When input increases by 1, the algorithm should only slow down by some constant factor
  - An algorithm is efficient if it has a polynomial running time

- **Asymptotic Order of Growth**
  - Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.
  - Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
  - Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

- Survey of **common running times**
  - There are styles of analysis of algorithms that occur frequently: $O(n)$, $O(n \log n)$, $O(n^2)$