ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Algorithm Complexity

Prof. Dr. Goran Glavaš

2.11.2023

Content

- Analyzing algorithms
- Complexity abstractions
  - Rate/order of growth
- Big-O notation

In the beginning, there were only problems

- Algorithms are designed to solve **problems**
- Problems are commonly specified with:
  - Inputs
  - Desired outputs
  - **Non-functional constraints**
    - E.g., time of space complexity

Find element

Input: A set of n numbers $\{a_1, a_2, \ldots, a_n\}$ and a query number b
Output: Answer to the question „is b in $\{a_1, a_2, \ldots, a_n\}$"

Analyzing algorithms

Find element

Input: A set of *n* numbers $\{a_1, a_2, ..., a_n\}$ and a query number b
Output: Answer to the question „is b in $\{a_1, a_2, ..., a_n\}$"

- You have written *an algorithm* for the above find element problem
  - Is it a **good** algorithm for the problem?
  - Is it the **only** algorithm that solves the given problem?
  - If you can think of more than one algorithm for the problem, which one is **better** and why?

Analyzing algorithms

Find element

Input: A set of *n* numbers $\{a_1, a_2, ..., a_n\}$ and a query number b
Output: Answer to the question „is b in $\{a_1, a_2, ..., a_n\}$”

- Criteria for evaluating algorithms
  - **Correctness**: does it give a correct output for every input?
    - In other words, does it actually solve the problem correctly

```
find_element(7, {2, 17, 35, 1, 14}) -> False
find_element(35, {2, 17, 35, 1, 14}) -> True
```

Analyzing algorithms

- Criteria for evaluating algorithms
  - **Efficiency**: how much computational resources and time does an algorithm's execution require?
  - If we have multiple **correct** algorithms for the problem, we would, intuitively, use the most efficient one
    - The **fastest** among correct algorithms – **time complexity**
    - The one using the **least computer resources** (typically **memory**) – **space complexity**
    - Time and space complexity are often in a trade-off relation
  - How to **measure** time and space complexity of algorithms?

Analyzing algorithms

Find element

Input: A set of *n* numbers $\{a_1, a_2, ..., a_n\}$ and a query number b
Output: Answer to the question „is b in in $\{a_1, a_2, ..., a_n\}$"

- How to measure time and space complexity of algorithms?
  - **Execution time** and **memory occupation** in most cases **directly depend on the actual input** (actual values provided for the input variables)

```
find_element(7, {2, 17, 35, 1, 14, 9, 43, 91}) -> False
```

vs.

```
find_element(35, {35, 1, 14}) -> True
```

Which execution is faster and requires less memory?

Analyzing algorithms

- **Complexity theory**
  - Formal examination of an algorithm with respect to its efficiency
  - Time efficiency usually much more important than space complexity.
    - **Q:** Why?

- The actual efficiency depends on concrete inputs, but we need to analyze algorithms "in general", that is, for "any (allowed) input"
  - **Best case running time** – time efficiency in/for the **most favorable** case/inputs
    - Lower bound: for no input can the running time be smaller than this
  - **Worst case running time** – time efficiency in/for the **least favorable** case/inputs
    - Upper bound: for no input can the running time be larger than this
  - **Average-case running time** – estimate of time efficiency across all input possibilities

Analyzing algorithms

Find element

Input: A set of n numbers $\{a_1, a_2, \ldots, a_n\}$ and a query number b
Output: Answer to the question „is b in $\{a_1, a_2, \ldots, a_n\}$?"

Algorithm

```
find_element(b, a_set)
  for a in a_set
    if a = b
      return True
  return False
```

read/write

comparison

assignment

- How do we measure time complexity?
- In terms of **number of elementary operations executed**
  - How does that number depend on the input?
- Given the length n of the input set „a_set", what is
  - The smallest possible number of comparisons?
  - The largest possible number of comparisons?
  - Number of comparisons „on average"?

Content

- Analyzing algorithms
- Complexity abstractions
  - Rate/order of growth
- Big-O notation

Complexity abstractions

- How do we measure time complexity?
  - In terms of **number of elementary operations executed**
  - How does that number depend on the input? What is the size of the problem?
  - What about the operations that do not depend on the size of the input?

- Let us go back to the sorting problem...

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$
(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, ..., a'_n>$ such that
$a'_1 \leq a'_2 \leq ... \leq a'_n$

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of *n* numbers $\langle a_1, a_2, ..., a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, ..., a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq ... \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)  # L is a list of numbers
  for i = 1 to L.length - 1 # 0-indexing, first element is at index 0, last at len-1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

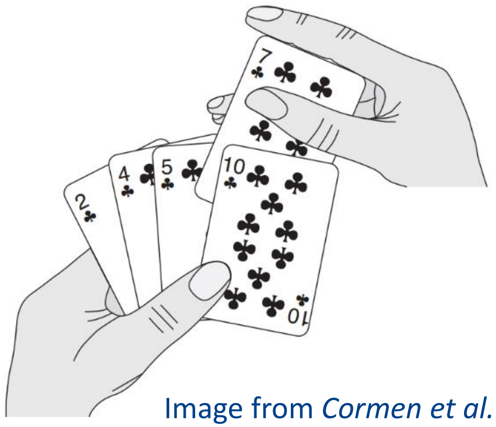


Image from *Cormen et al.*

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

`L.Length = 8`

0	1	2	3	4	5	6	7
9	2	6	4	5	1	8	7

1st iteration of outer loop (**for**)

9	2	6	4	5	1	8	7

```
i = 1
key = L[i] = 2
j = i-1 = 0
j > -1 and L[j] > key -> True
L[1] = L[0] = 9
```

9	9	6	4	5	1	8	7

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$

(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, ..., a'_n>$ such that

$$a'_1 \leq a'_2 \leq ... \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length – 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j – 1
    L[j+1] = key
```

```
L.Length = 8
```

0	1	2	3	4	5	6	7
9	2	6	4	5	1	8	7

1st iteration of outer loop (**for**)

9	9	6	4	5	1	8	7

```
key = 2
...
L[1] = L[0] = 9
j = j-1 = -1
j > -1 and L[j] > key -> False
-> exited while loop
L[0] = key = 2
```

2	9	6	4	5	1	8	7

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$

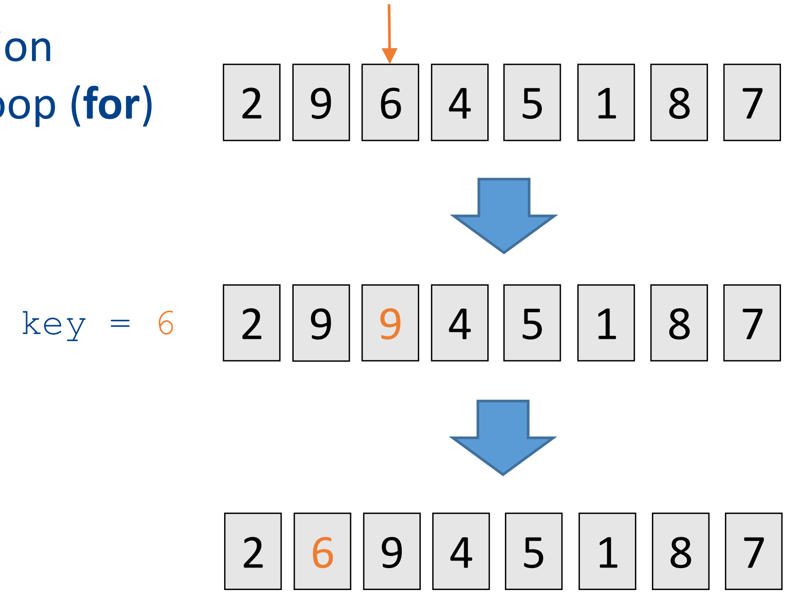
(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, ..., a'_n>$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

2nd iteration
of outer loop (**for**)



Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

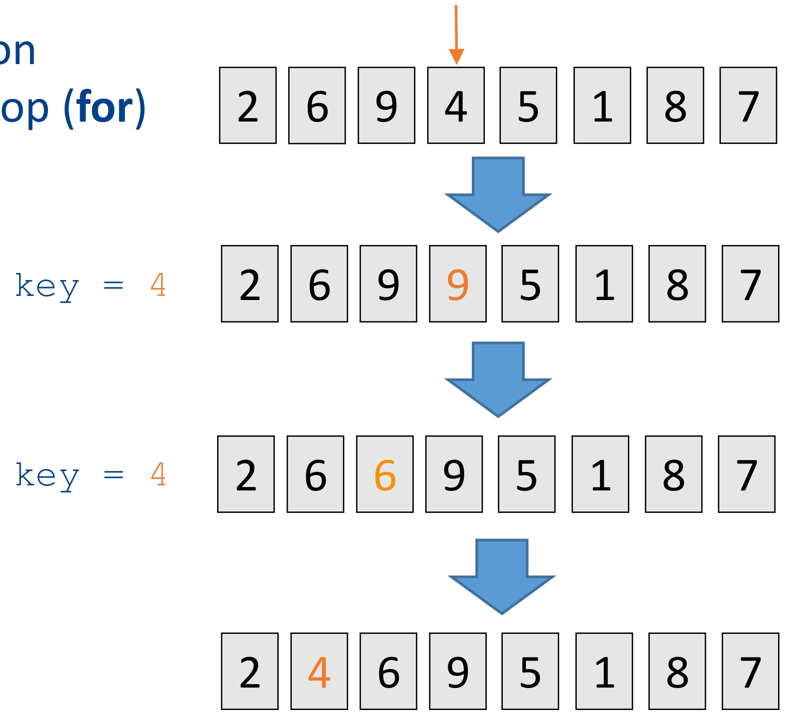
(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

3rd iteration
of outer loop (**for**)



Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that
$a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Algorithm: insert(ion) sort

```
insert_sort(L)  # L is a list of numbers
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

- Let's analyze running time
  - n = `L.length`: num. elements in the list
- Elementary operation: assignment
  - Assignment of value to iterator variable *i*
  - Assigned fixed cost c_1
- Executed how many times?
  - Cost: $(n-1) * c_1$

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$
(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, ..., a'_n>$ such that
$a'_1 \leq a'_2 \leq ... \leq a'_n$

Algorithm: insert(ion) sort

```
insert_sort(L) # L is a list of numbers
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

Cost: (n-1) * c_3

- Let's analyze running time
  - n = `L.length`: num. elements in the list
- Elementary operations:
  - Reading value L[i]
  - Assignment of that value to variable *key*
  - Assigned fixed cost c_2
- Executed how many times?
  - Cost: (n-1) * c_2

read/write

assignment

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$
(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, ..., a'_n>$ such that
$a'_1 \leq a'_2 \leq ... \leq a'_n$

Algorithm: insert(ion) sort

```
insert_sort(L)  # L is a list of numbers
  for i = 1 to L.length - 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

- L[j+1] = L[j]: Cost $\sum_{i=1}^{n-1} c_5 * (t_i\text{-}1)$
- j = j - 1: Cost $\sum_{i=1}^{n-1} c_6 * (t_i-1)$
- L[j+1] = key: Cost $(n\text{-}1) * c_7$

- Let's analyze running time
  - n = `L.length`: num. elements in the list
- Elementary operations (while condition):
  - 2 comparisons in the complex condition
  - Assigned fixed cost c_4
- Executed how many times?
  - That **depends on the condition**
  - For each *i* we have $\mathbf{t_i}$ executions of while conditions and all commands inside of the **while** loop
  - Cost $\sum_{i=1}^{n-1} c_4 * t_i$

Complexity abstractions (on insert sort)

Sorting Problem

Input: A sequence of n numbers <a_1, a_2, ..., a_n>
(Desired) Output: A permutation (reordering) of the input <a'_1, a'_2, ..., a'_n> such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Algorithm: insert(ion) sort

```
insert_sort(L)
  for i = 1 to L.length - 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key # Σ_{i=1}^{n-1} c4 * ti
      L[j+1] = L[j] # Σ_{i=1}^{n-1} c5 * (ti-1)
      j = j - 1  # Σ_{i=1}^{n-1} c6 * (ti - 1)
    L[j+1] = key  # (n-1) * c7
```

- Let's analyze running time

  n = `L.length`: num. elements in the list

- Total running time **T(n)**

T(n) = $(n-1) * (c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * t_i + (c_5 + c_6) * (t_i - 1)$$

Complexity abstractions (on insert sort)

Algorithm: insert(ion) sort

```
insert_sort(L)
  for i = 1 to L.length - 1 # (n-1) * c_1
    key = L[i]  # (n-1) * c_2
    j = i-1   # (n-1) * c_3
    while j > -1 and L[j] > key # Σ_{i=1}^{n-1} c_4 * ti
      L[j+1] = L[j] # Σ_{i=1}^{n-1} c_5 * (t_i - 1)
      j = j - 1  # Σ_{i=1}^{n-1} c_6 * (t_i - 1)
    L[j+1] = key # (n-1) * c_7
```

- Total running time **T(n)**

T(n) = $(n\text{-}1) *(c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * t_i + (c_5 + c_6) * (t_i - 1)$$

- **T(n)** depends not only on n but also on concrete numbers in `L` (their order)
- What is the **best possible** scenario (smallest possibe running time)?
  - If the input L is already sorted
  - $t_i = 1$ for each i
  - $T(n) = (n\text{-}1) *(c_1 + c_2 + c_3 + c_7 + c_4)$
  - Let's sum up the constant elementary operation costs: $a = c_1 + c_2 + c_3 + c_7 + c_4$
  - $T(n) = a*n - a$: this is a linear function of n

Complexity abstractions (on insert sort)

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length – 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key # Σ_{i=1}^{n-1} c4 * ti
      L[j+1] = L[j] # Σ_{i=1}^{n-1} c5 * (ti-1)
      j = j – 1  # Σ_{i=1}^{n-1} c6 * (ti – 1)
    L[j+1] = key # (n-1) * c7
```

- Total running time **T(n)**

T(n) = $(n-1) * (c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * ti + (c_5 + c_6) * (t_i - 1)$$

- What is the **worst possible** scenario (largest possible running time)?
  - If the input L is inversely sorted (from largest to smallest value)
  - $t_i = i$ for each i
  - $\sum_{i=1}^{n-1} c_4 * t_i = (1 + 2 + ... + (n-1)) * c_4 = \frac{(n-1)*n}{2} * c_4$
  - $\sum_{i=1}^{n-1} c_5 * (t_i - 1) = (0 + 1 + ... + (n-2)) * c_5 = \frac{(n-2)*(n-1)}{2} * c_5$
  - $\sum_{i=1}^{n-1} c_6 * (t_i - 1) = (0 + 1 + ... + (n-2)) * c_6 = \frac{(n-2)*(n-1)}{2} * c_6$

Complexity abstractions (on insert sort)

Algorithm: insert(ion) **sort**

```
insert_sort(L)
  for i = 1 to L.length - 1 # (n-1) * c1
    key = L[i]  # (n-1) * c2
    j = i-1   # (n-1) * c3
    while j > -1 and L[j] > key # Σ_{i=1}^{n-1} c4 * ti
      L[j+1] = L[j] # Σ_{i=1}^{n-1} c5 * (ti−1)
      j = j - 1  # Σ_{i=1}^{n-1} c6 * (ti − 1)
    L[j+1] = key # (n-1) * c7
```

- Total running time **T(n)**

T(n) = $(n\text{-}1) * (c_1 + c_2 + c_3 + c_7) +$

$$\sum_{i=1}^{n-1} c_4 * ti + (c_5 + c_6) * (t_i - 1)$$

- What is the **worst possible** scenario (largest possible running time)?
  - If the input L is inversely sorted (from largest to smallest value)
  - $t_i = i$ for each i
  - $T(n) = (n\text{-}1) * (c_1 + c_2 + c_3 + c_7) + \frac{(n-1)*n}{2} * c_4 + \frac{(n-2)*(n-1)}{2} * (c_5 + c_6)$
  - $T(n) = a*n^2 + b*n + c$
  - This is a quadratic function of n

Focus on worst case running time

- In much of algorithm complexity analysis, we focus on the **worst case running time** because of the following

1. Worst case running gives an upper bound on the running time
   - whatever the input, the running time **cannot be worse than this**
2. For many algorithms the worst case running time occurs often
   - Example: search database for values not in database
3. The average running time is often **not much better** than worst case
   - Insert-sort average: $t_i = i/2$
   - This still makes the **T(n)** a quadratic function of n
     - Just the coefficients a, b, and c will be smaller
     - But this has little effect if *n* **is large** → **growth of functions**

Complexity abstractions: **rate of growth**

- To compute **T(n)** we already used simplifying abstractions
  1. Ignored actual costs of elementary operations, replaced them with constants c_i
  2. Replaced any combination of constants c_i with a constant (a, b, c)
- This gave the worst case running time function for insert sort
  - **$T(n) = a*n^2 + b*n + c$**
- But we are actually interested in the rate of growth of the running time, with the increase of n
  - For small n, any algorithm will run „fast enough"
  - We need to see how T(n) grows with n

Complexity abstractions: **rate of growth**

- **T(n) = a*n^2 + b*n + c**
  - For growing n

- We introduce further simplifications for simpler description of time efficiency
  1. We keep only the leading term of the polynomial above, **a***n^2
     - For large n, n^k is an order of magnitude larger than n^{k-1}
     - The larger n is, the more insignificant n^{k-1} is compared to n^k
     - Example (n^2 vs. n for different n): for n = 5, 25 vs. 5; for n = 10^6 it's 10^{12} vs. 10^6

  2. We can lose the constant – as n becomes larger, the constant factors become less significant also (the constants don't grow with n)
     - The constant operation cost does not affect the order of growth
     - $(a * n_1^{k)}) / (a * n_2^k)$ = $\mathbf{(n_1/n_2)^k}$ – as n is growing, the **increase in running time** doesn't depend on a

Content

- Analyzing algorithms
- Complexity abstractions
  - Rate/order of growth
- Big-O notation

Rate of growth and efficiency

- A_1 more efficient than A_2 **if**
  - **Worst case running** time of A_1 has a lower rate of growth than that of A_2

- **Worst case running time** (considering only rate of growth)
  - Denoted with symbol Θ (uppercased „theta”)
  - **Insert sort** has a worst case running time $T(n) = \mathbf{a}*n^2 + \mathbf{b}*n + \mathbf{c}$
    - But (only) n^2 drives the rate of growth of T(n)
  - So we say: it has the worst case running time $\Theta(n^2)$ („theta of n-squared”)
    - Also, colloquially, **insert sort** has „quadratic complexity” (or „complexity n-square”)

Rates of growth and complexity

- Growth rates for some common complexity functions
  - $\Theta(1)$ (constant)
  - $\Theta(\log n)$ (logarithmic)
  - $\Theta(n)$ (linear)
  - $\Theta(n \log n)$ (loglinear)
  - $\Theta(n^2)$ (quadratic complexity)
  - $\Theta(n^3)$ (cubic complexity)
    - … $\Theta(n^k)$ for $k \geq 0$ (polynomial)
  - $\Theta(2^n)$ (exponential)
  - $\Theta(n!)$ (factorial)

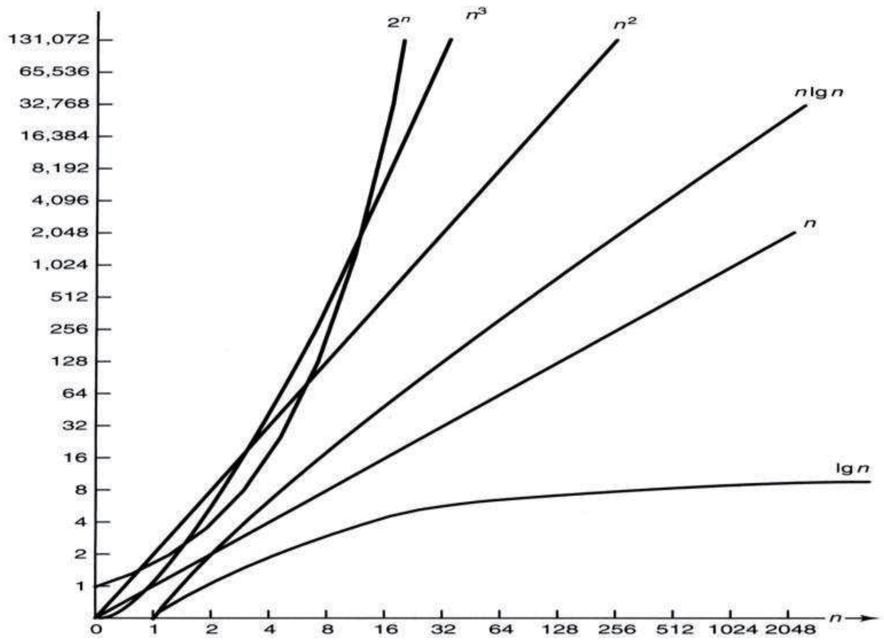


Image from https://tinyurl.com/46c3cssy

Asymptotic notation

- Say we have two algorithms A_1 and A_2
  - Worst-case running times: $T_1(n) = a * n + b$;
    $T_2(n) = c * n^2 + d * n + e$
- For some (small) values of n, depending on the values of constants (a, b, c, d, e), $T_2(n)$ may even be lower than $T_1(n)$
- But when we look **at input sizes large enough** to make **only** rate of growth of running time relevant, the *quadratic* running time will be larger than *linear*
  - There is a (large enough) value n_0 such that for all $\mathbf{n \geq n_0}$, $T_2(n) \geq T_1(n)$
- **Asymptotic efficiency** of algorithms: looking at input sizes so large that only rate of growth of the worst running time of the algorithm matters ($\mathbf{n \geq n_0}$)

Θ−notation

- For insert sort, we denoted the worst running time as $T(n) = \Theta(n^2)$
- Now we formally define the theta function

Θ-notation

For a given function $g(n)$, $\Theta(g(n))$ denotes a set of functions
$\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, and n_0 such that
$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

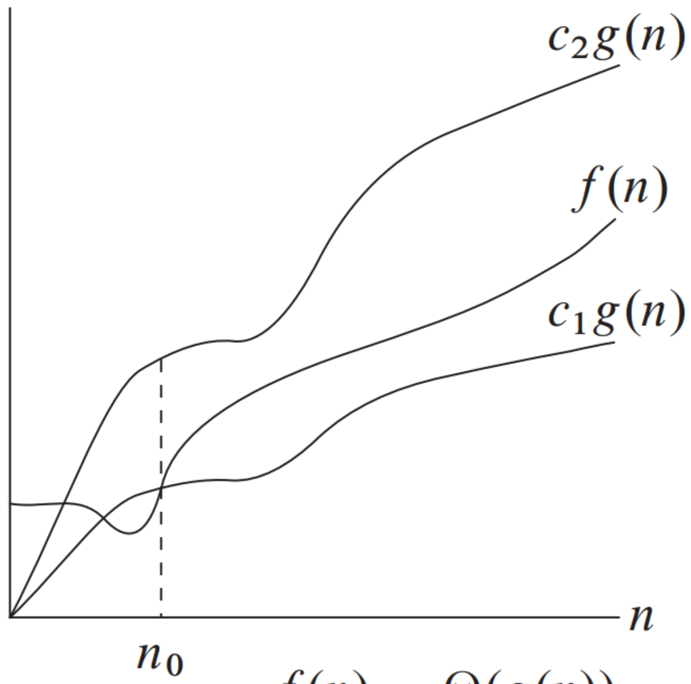
- **Q**: is the above satisfied for $g(n) = n^2$ and $f(n) = ½\, n^2 + 2n$?
  - Give one set of valid values for c_1, c_2, and n_0
  - If, for example, $c_1 = ½$, $c_2 = 1$, what is then n_0?

Θ−notation

Θ-notation

For a given function $g(n)$, $\Theta(g(n))$ denotes a set of functions
$\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, and n_0 such that
$0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n \ge n_0$

- $f(n)$ „sandwiched” between $c_1 g(n)$ and $c_2 g(n)$
  - Gurantee that this is true for all $n \ge n_0$
- $g(n)$ **asymptotically tight bound** for $f(n)$
  - Both **upper** ($c_2 g(n)$) and **lower** asymptotic bound ($c_1 g(n)$)
- For polynomials: $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$
  - $f(n) = \Theta(n^d)$
  - Constants are polynomials of 0-th degree: $\Theta(n^0) = \Theta(1)$



$f(n) = \Theta(g(n))$

(Big) O-notation

- Θ-notation bounds a function from both above and below
- In algorithmic efficiency, we are typically interested more (only) in the **asymptoptic upper bound**

O-notation

For a given function $g(n)$, $O(g(n))$ denotes a set of functions
$O(g(n)) = \{ f(n)$: there exists positive constants c, and n_0 such that
$0 \leq f(n) \leq c*g(n)$ for all $n \geq n_0$

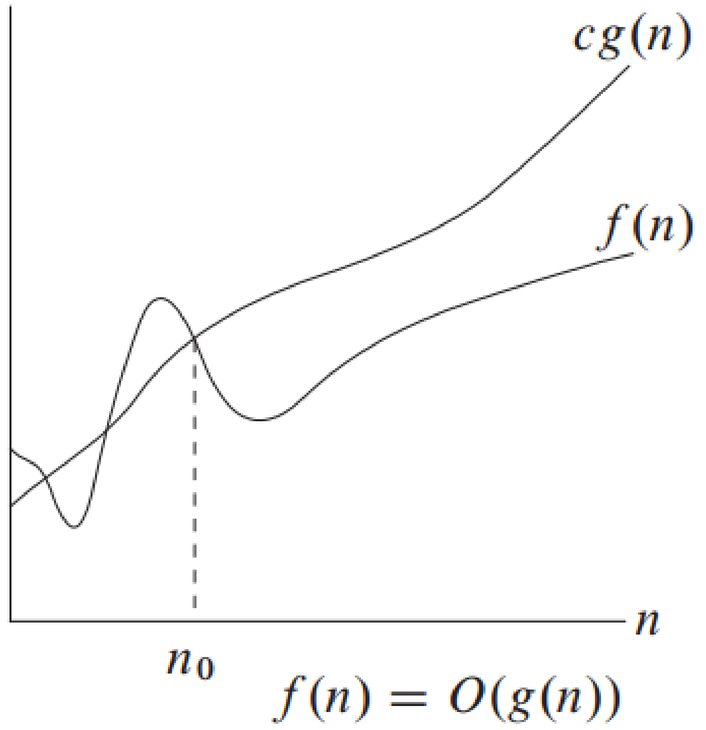
- O-notation: an upper bound of a function *to within a constant factor*
  - Any $f(n)$ in $\Theta(g(n))$ is surely also in $O(g(n))$
  - The opposite not true: e.g., linear functions are in $O(n^2)$ but not in $\Theta(n^2)$

(Big) O−notation

O-notation

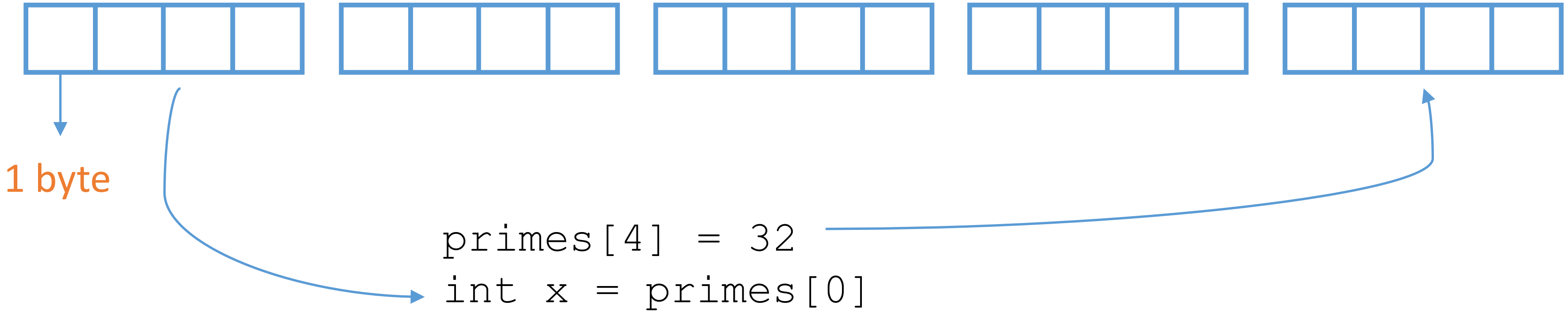
For a given function $g(n)$, $O(g(n))$ denotes a set of functions
$O(g(n)) = \{ f(n)$: there exists positive constants c, and n_0 such that
$0 \leq f(n) \leq c*g(n)$ for all $n \geq n_0$

- $f(n)$ limited from above with $c\, g(n)$
  - Gurantee that this is true for all $n \geq n_0$
- $g(n)$ **asymptotic upper bound** for $f(n)$
- For polynomials: $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$
  - $f(n) = O(n^k)$, for each $k \geq d$



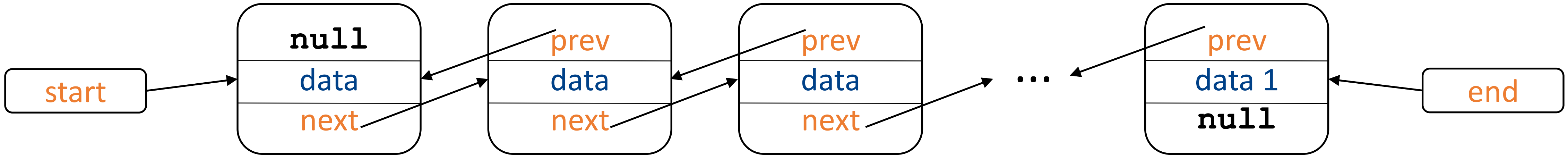
Lists: Arrays vs. Linked Lists

- ADT: **List** – a linear sequence of elements, ordered collection of values
  - When we design algorithms, we typically think in terms of ADTs
- **Array** (as **data structure**, not ADT)
  - Writing values into and reading values from an array is **fast**
  - Example in *C* (as Arrays in Python or Java are implemented differently)
    - `int primes[5];` // allocate 5 x size of int (typically 4 bytes) of contiguous memory



Lists: Arrays vs. Linked Lists

- ADT: **List** – a **linear** sequence of elements
  - When we design algorithms, we typically think in terms of ADTs
- **Linked List**
  - Consists of **nodes**: nodes contain both the data (values) and a **pointer** to the next node in the list
  - Nodes can contain values of different types
  - Dynamic data structure: „resizable" at run time
    - Non-contiguous memory allocation possible, space for **new nodes** can be allocated dynamically (on „per-need" basis)



Set element of List – running time

- List as **Array**
  - **L:** the memory address of the first element of the list (array)
  - **size** – the number of bytes for storing one value

```
set_element(L, ind, val)
   L = L + ind*size
   write(L, val)
```

n = size of the list
Worst running time
(Big-O)?

- List as **Linked List**
  - **L:** pointer (memory address) to the first node of the list

```
set_element(L, ind, val)
   for i = 0 to ind
      L = L.next
   write(L, val)
```

n = size of the list
Worst running time
(Big-O)?

Stacks and queues – running time

- Stack

```
push(S, x)
  if S.top == len(S.elements) - 1
    error „overflow”
  else
    S.elements[S.top] = x
    S.top = S.top + 1
```

- Queue

```
enqueue(Q, x)
  if is_full(Q)
    error „overflow”
  else
    Q.elements[Q.tail] = x
    Q.tail = (Q.tail + 1) % len(Q.elements)
```

n = size of the stack/queue

Worst running time (Big-O)?

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟