Lesson 1: Computational Complexity

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS

DR. ANDREY TIMOFEYEV

OUTLINE

- Introduction.
- Big-O notation.
- Big-Omega notation.
- Big-Theta notation.
- Common computation complexities.
- Algorithm analysis examples.
- Proof by induction.
- Amortized complexity.

INTRODUCTION

- **Robust code** must be based on *"good"* **data structures** & **algorithms**.
  - **Data structures** - systematic way of organizing & accessing data.
  - **Algorithms** - step-by-step procedure for performing some task in a finite amount of time.
- **Computational complexity** – measure of the "goodness" of algorithms & data structures.
  - Analysis of **runtime** of **algorithms** & **data structure operations**.
    - Measured by the number of **primitive operations** required to be performed by the CPU.
- Algorithms are **characterized** by *f(n)*, where *n* = **input size**.
  - *f(n)* approximates **growth rate** of algorithm **running time** as a function of **input size** *n*.
    - A.k.a. *"asymptotic analysis"*.
  - Generally, interested in **worst case (upper bound)** – **Big-O notation**.
    - Additionally: **Big-Omega Ω** (lower bound) & **Big-Theta Θ** (tight bound).

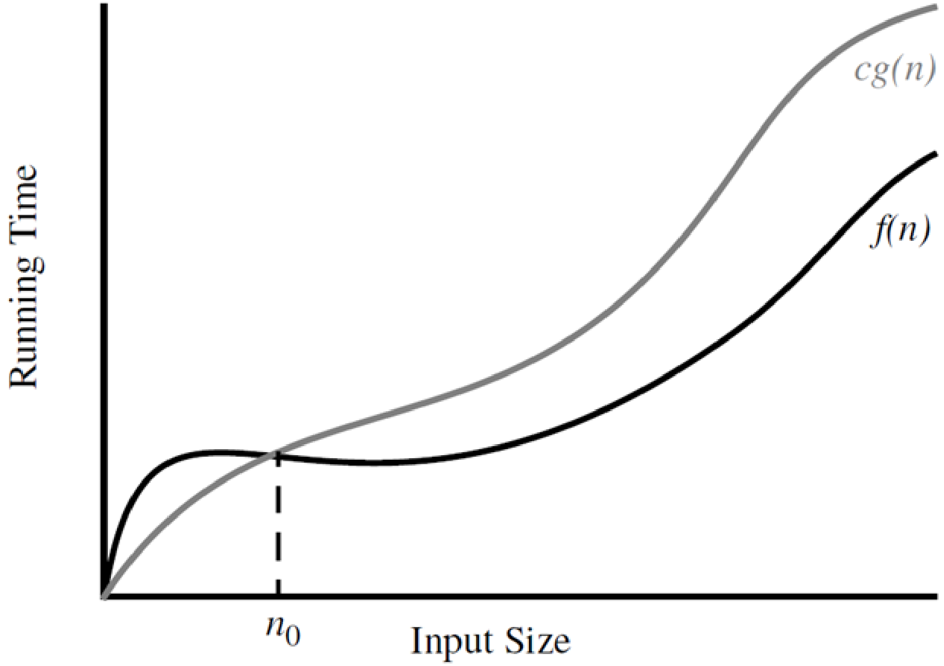
BIG-O NOTATION

- **Big-O notation** (upper bound) formal definition:
  - ***f(n)* is *O(g(n))***, if there is **real constant *c > 0*** and **integer constant $n_0 \geq 1$**, such that:

$$f(n) \leq c\, g(n), \text{ for } n \geq n_0$$

  - Now, in English:
    - f(n) is “less than or equal to” g(n) up to a constant factor and as n asymptotically grows toward infinity.

- **Big-O properties:**
  - Ignores **constant factors** & **lower order terms**.
    - $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$
  - Characterizes function in **simplest terms**.



Big-O illustration

ADDITIONAL NOTATIONS: BIG-OMEGA & BIG-THETA

- **Big-Omega notation** (lower bound) formal definition:
  - ***f(n)* is *Ω(g(n))***, if ***g(n)* is *O(f(n))***: there is **real constant *c > 0*** and **integer constant $n_0 \geq 1$**, such that:

$$f(n) \geq c\, g(n), \text{ for } n \geq n_0$$

  - Now, in English:
    - One function is asymptotically greater than or equal to another, up to a constant factor.

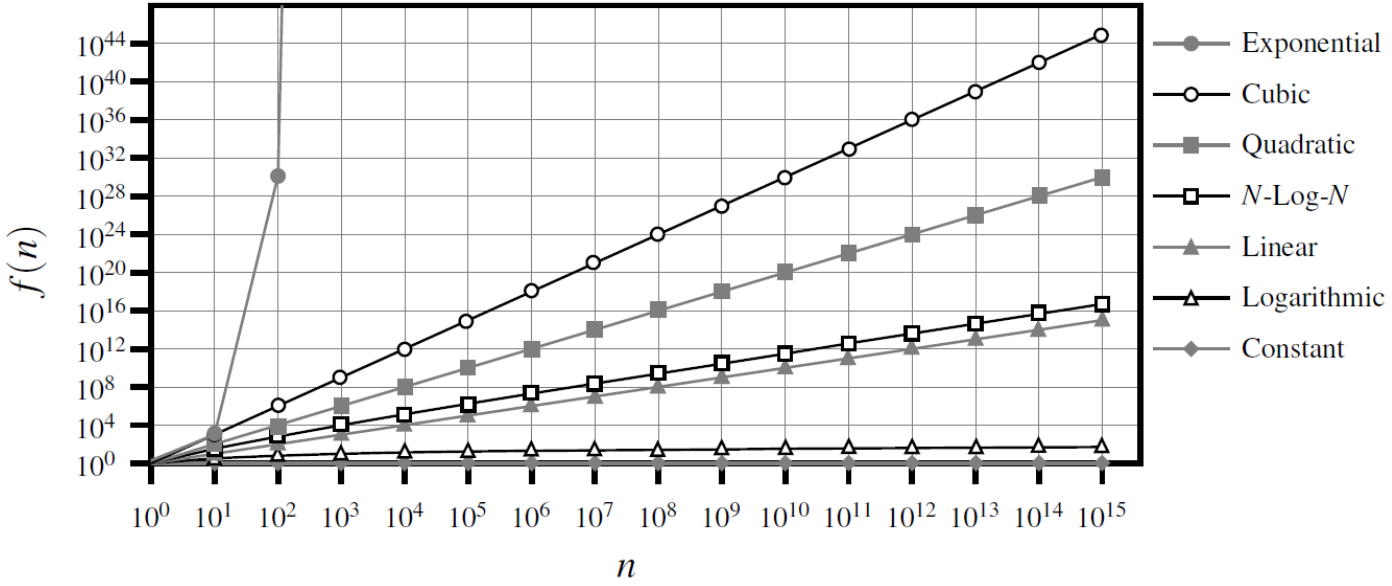
- **Big-Theta notation** (tight bound) formal definition:
  - ***f(n)* is *Θ(g(n))***, if ***f(n)* is *O(g(n))*** and ***f(n)* is *Ω(g(n))***: there are **real constants *c' > 0*** and ***c'' > 0***, and **integer constant $n_0 \geq 1$**, such that:

$$c'\, g(n) \leq f(n) \leq c''\, g(n), \text{ for } n \geq n_0$$

  - Now, in English:
    - Two functions grow asymptotically at the same rate, up to constant factors.

COMMON COMPUTATIONAL COMPLEXITIES

- **Commonly occurring computational complexities:**
  - **Constant:** *O(1)*.
    - Accessing an element in an array given its index.
    - Checking if a number is even or odd.
  - **Logarithmic:** *O(log n)*.
    - Binary search, binary tree operations.
  - **Linear:** *O(n)*.
    - Linear search, finding the maximum element in a list.
  - **Quasi-linear:** *O(n log n)*.
    - Advanced sorting algorithms (merge sort).
  - **Quadratic:** *O(n^2)*.
    - Basic sorting algorithms (bubble sort). Nested loops, 2D arrays.
  - **Cubic:** *O(n^3)*.
    - Nested loops, 3D arrays.
  - **Exponential:** *O(2^n)*.
    - Finding all subsets of a data collection. Towers of Hanoi.
  - **Factorial:** O(n!).
    - Figuring out a password given all the characters of the password.



Growth rate comparisons

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Classes of functions

EXAMPLES OF ALGORITHM ANALYSIS (1)

- **Finding largest element in the list.**

```
def find_max(data):
  """Return the maximum element from a nonempty list"""
  biggest = data[0] # The initial value to beat
  for val in data: # For each value:
      if val > biggest: # if it is greater than the best so far,
        biggest = val # we have found a new best (so far)
  return biggest # When loop ends, biggest is the max
```

EXAMPLES OF ALGORITHM ANALYSIS (2)

- **Prefix averages.**
  - Given sequence *S* consisting of *n* numbers, compute sequence *A*, such that *A[j]* is the average of elements *S[0], …, S[j]*, for *j = 0, …, n-1*: $A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$

```
def prefix_average1(S):
  n = len(S)
  # create new list of n zeros
  A = [0] * n
  for j in range(n):
    # begin computing S[0]+...+S[j]
    total = 0
    for i in range(j + 1):
      total += S[i]
    # record the average
    A[j] = total / (j+1)
  return A
```

```
def prefix_average2(S):
  n = len(S)
  # create new list of n zeros
  A = [0] * n
  for j in range(n):
    # record the average
    A[j] = sum(S[0:j+1]) / (j + 1)
  return A
```

```
def prefix_average3(S):
  n = len(S)
  # create new list of n zeros
  A = [0] * N
  # compute prefix sum as S[0]+S[1]+…
  total = 0
  for j in range(n):
    # update prefix sum to include S[j]
    total += S[j]
    # compute average based on current sum
    A[j] = total / (j + 1)
  return A
```

EXAMPLES OF ALGORITHM ANALYSIS (3)

- **Three-way set disjointness.**
  - Determine if the intersection of the three sequences is empty.
    - No element *x* such that $x \in A$, $x \in B$, $x \in C$.

```
def disjoint1(A, B, C):
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          # we found a common value
          return False
  # if we reach this, sets are disjoint
  return True
```

```
def disjoint2(A, B, C):
  for a in A:
    for b in B:
      # only check C if we found match from A and B
      if a == b:
        for c in C:
          # (and thus a == b == c)
          if a == c:
            # we found a common value
            return False
  # if we reach this, sets are disjoint
  return True
```

EXAMPLES OF ALGORITHM ANALYSIS (4)

- **Element uniqueness.**
  - Check if elements of a given collection are distinct from each other.

```
def unique1(S):
  for j in range(len(S)):
    for k in range(j + 1, len(S)):
      if S[j] == S[k]:
        # found duplicate pair
        return False
  # if we reach this, elements were unique
  return True
```

```
def unique2(S):
  # create a sorted copy of S
  temp = sorted(S)
  for j in range(1, len(temp)):
    if S[j-1] == S[j]:
      # found duplicate pair
      return False
  # if we reach this, elements were unique
  return True
```

PROOF BY INDUCTION (1)

- **Computational complexity** expressed by the **Big-O notation** that is defined for the input *size n* as it grows towards the **infinity**.
  - Justifying **exhaustively** (e.g. for every possible value of n = 1, 2, 3, 4, …) is not feasible.
  - **Proof by mathematical induction** is used instead.
- **Proof by induction process:**
  - Step 1: Base case.
    - Prove the statement is true for the **base case** (*trivial value*).
  - Step 2: Inductive case.
    - **Inductive hypothesis:** assume the statement is true for *n – 1*.
    - Show the **statement is valid** for n.

PROOF BY INDUCTION (2)

- **Examples:**
  - Prove that given nested loop has $O(n^2)$ complexity.

```
n = 5
for j in range(n):
  print(f"Outer iteration: {j}“) # goes through n iterations
  for i in range(j + 1):
    print(f"\tInner iteration: {i}") # iterations grow by 1 for each outer iteration
  print()
```

PROOF BY INDUCTION (2)

- **Examples:**
  - Prove that given nested loop has $O(n^2)$ complexity.

```
n = 5
for j in range(n):
  print(f"Outer iteration: {j}") # goes through n iterations
  for i in range(j + 1):
    print(f"\tInner iteration: {i}") # iterations grow by 1 for each outer iteration
  print()
```

- Proof by induction: $1 + 2 + 3 + 4 + 5 + \ldots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  - Step 1: Base case.
    - $n = 1$
    - $1 = \frac{n(n+1)}{2} = \frac{1(1+1)}{2} = \frac{2}{2} = 1$
  - Step 2: Inductive case.
    - Assume valid for n – 1 and show valid for n.
    - $\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n$ (by summation definition).
    - By induction hypothesis:
      - $\sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)n}{2}$
    - Finally:
      - $\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n = \frac{(n-1)n}{2} + \mathrm{n} = \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}$

PROOF BY INDUCTION (3)

- **Examples (cont.):**
  - Prove that Fibonacci sequence is $< 2^n$

    $F(1) = 1,$

    $F(2) = 2,$

    $F(n) = F(n-2) + F(n-1)$, for $n > 2$.

    $F(n) < 2^n$

PROOF BY INDUCTION (3)

- **Examples (cont.):**
  - Prove that Fibonacci sequence is $< 2^n$

    $F(1) = 1,$

    $F(2) = 2,$

    $F(n) = F(n-2) + F(n-1)$, for $n > 2$.

    $F(n) < 2^n$

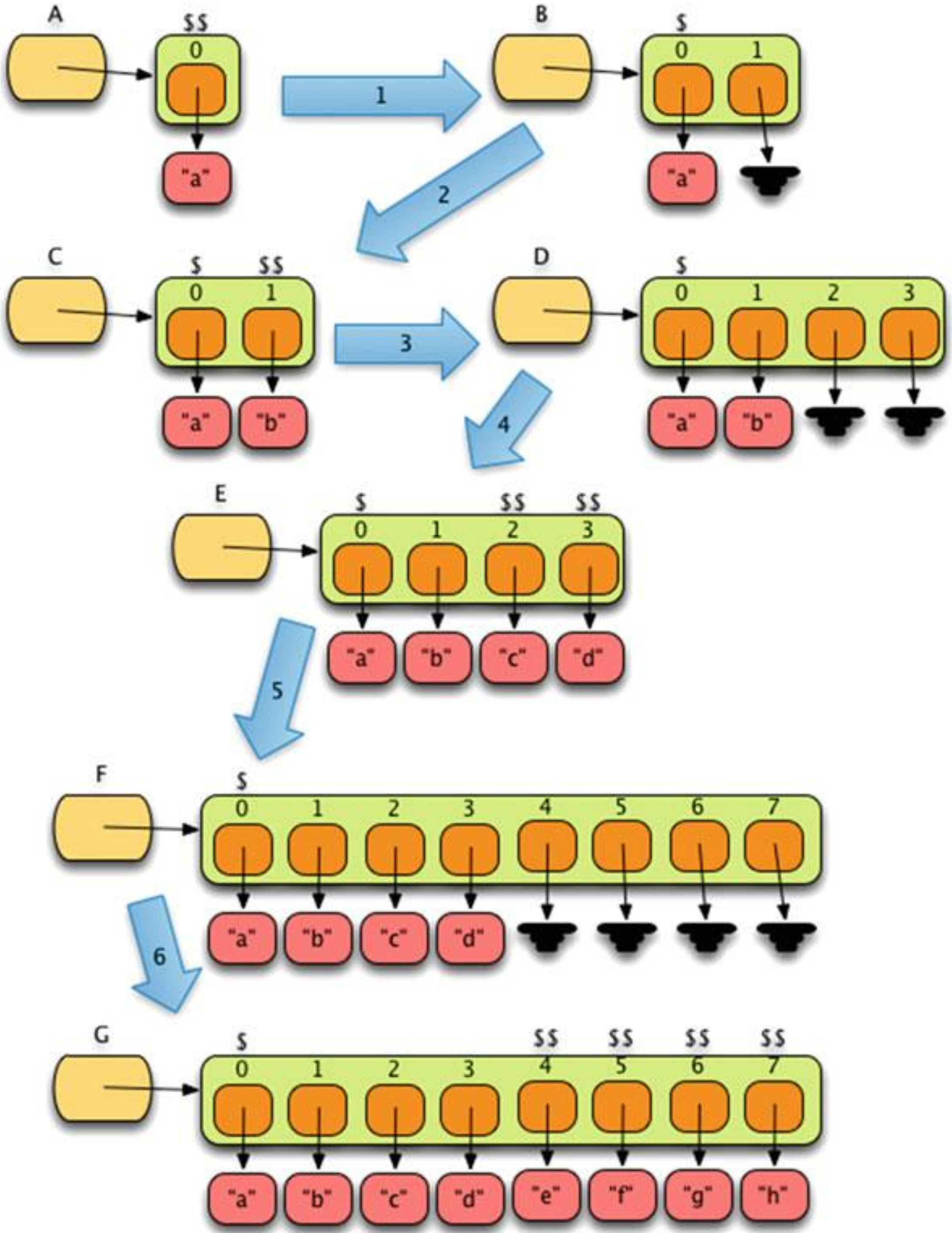
- Proof by induction: $F(n) < 2^n$
  - Step 1: Base case ($n \leq 2$).
    - $F(1) = 1 < 2^1 = 2$
    - $F(2) = 2 < 2^2 = 4$
  - Step 2: Inductive case ($n > 2$).
    - Assume valid for $n - 2$ & $n - 1$ and show valid for n.
      - $F(n) = F(n-2) + F(n-1)$
    - By induction hypothesis:
      - $F(n) < 2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1} = 2*2^{n-1} = 2^n$

AMORTIZED COMPLEXITY (1)

- **Amortized complexity** - estimates more **accurate** algorithm runtime.
  - Considers the **tightest upper bound** of the **worst-case running time** of sequence of operations.
  - **Divides** by number of operation to get **average** (*amortized*) running time of each operation.
- **Example:**
  - Python list allows **appending** values with **O(1) complexity**.
  - Python list object is **implemented** in **C** programming language.
    - C only allows allocating **fixed** size lists (*arrays*).
    - **New array** must be **allocated** & values must be **copied** every time the list is out of space - **O(n) behavior**.
  - Python achieving this in O(1) is an **amortized complexity**.

AMORTIZED COMPLEXITY (2)

- The strategy is to **double** the **size** of a newly allocated array once **run out of space**.
  - Allows achieving an **amortized** O(1) complexity.



SUMMARY

- Computational complexity & asymptotic analysis.
- Big-O, Big-Omega, Big-Theta.
- Common computational complexities.
- Algorithm analysis.
- Proof by induction.
- Amortized complexity.