Data Structures

Lecture 2: Algorithm Analysis

CHEN Zhongpu, Fall 2024

School of Computing and Artificial Intelligence, SWUFE

西南财经大学
SOUTHWESTERN UNIVERSITY OF FINANCE AND ECONOMICS

A Small Quiz

1. What does the following code snippet do?

```
a = [1] * 10
```

```
i, j = 10, 42
i, j = j, i
```

2. How to measure efficiency?

> A data structure is a data organization, and storage format that is usually chosen for efficient access to data.

```
import time
start = time.time()
# your program runs
end = time.time()
elapsed = end - start
```

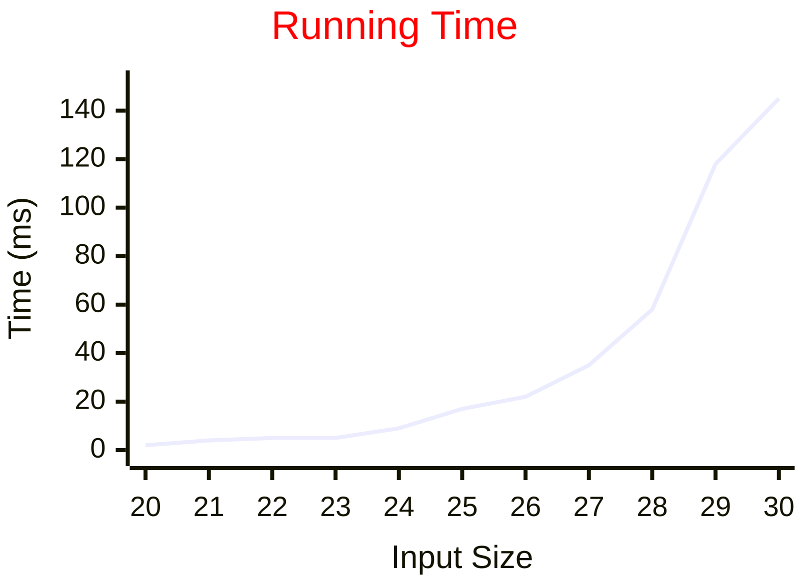
- `time.perf_counter()` is recommended for measuring elapsed time due to its high resolution.
- timeit is recommended measure execution time of small code snippets.

1. Empirical Analysis

```python
import time

def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

if __name__ == '__main__':
    ns = [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
    with open('fib_python.txt', 'w') as f:
        for n in ns:
            start = int(round(time.time() * 1000))
            fib(n)
            end = int(round(time.time() * 1000))
            f.write(f'{n}   {end - start}\n')
```



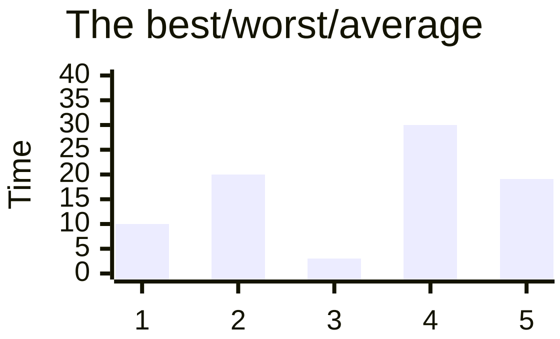
Which One is Faster?

`is_contains(lst_small, 42)` or `is_contains(lst_big, 42)`? We assume that `lst_small` is a short list, while `lst_big` is a long list.

```python
def is_contains(collection, target):
    for item in collection:
        if item == target:
            return True
    return False
```

Focusing on the Worst Case

An algorithm may fun faster on some inputs than it does on others of the same size.



- The best case
- The worst case
- The average case

Pitfalls of Empirical Analysis

- To compare different algorithms is feasible unless the hardware and software environments are the same.
- Experiments can be done only a limited set of test inputs.
- An algorithm must be fully implemented before the experiments.

Mathematical Model

$$T = f(n)$$

How long will my take, as a function of the input size (in the worst case)?

2. Mathematical Analysis

Knuth's Insight 💡

The total running time of a program is determined by two primary factors:

1. The cost of executing each statement
2. The frequency of execution of each statement

$$total\ time = \sum (cost \times frequency)$$

```python
def foo()
  i = 0
  result = i * (i + 1) + (i + 1) * (i + 2) + 42
```

```python
def bar(S):
  n = len(S)
  total = 0
  for j in range(n):
    total += S[j]
  return total
```

Primitive Operations ⚛

We assume that primitive operations take constant time to execute, such as the following:

- Assigning an identifier to an object
- Performing an arithmetic operation (for example, *adding two numbers*)
- Comparing two numbers
- Accessing a single element of a Python with an identifier
- Calling a function (excluding operations executed within the function)
- Returning from a function

Building Complexity on Simplicity

Given a list `lst` whose size is `N`, how many primitive operations are there?

- `sum(lst)`
- `len(lst)`
- `lst.append(1)`

Order of Growth

We can simplify the cost model by taking a "big-picture" approach: it is the order of growth (rate of growth) of the running time as a function of the input size. See plots at Overleaf.

Function	Approximation	Oder of growth
$n^3/6 - n^2/2 + n/3$	$n^3/6$	n^3
$n^2/2 - n/2$	$n^2/2$	n^2
$2\lg n + 1$	$2\lg n$	$\lg n$
3	3	1

- Ignores leading coefficient
- Ignores lower-order terms

3. Big O Notation

The order of growth is often described by an asymptotic notation big O.

Description	Time complexity
constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
linearithmic	$O(n \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(2^n)$

Big O

Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We write

$$f(x) = O(g(x))$$

if and only if there exists constants N and C such that

$$f(x) \leq Cg(x), \forall x > N$$

Intuitively, this means that f does not grow faster than g (g is the upper bound of f).

Informal Big O 👀

Big-O denotes the **less-than-or-equal-to** concept:

$$7n^3 + 100n^2 - 20n + 6$$

We can say its order of growth is n^3. To put it in another way, this function grows no faster than n^3, so we can write that it is $O(n^3)$. Note that in some research papers, it may be written in a fancy way, $\mathcal{O}(n^3)$.

Examples: True or False 💪

- The function $8n + 5$ is $O(n)$.
- The function $n^2 + 2n + 1$ is $O(n^2)$.
- The function $5n^2 + 3n \log n + 2n + 5$ is $O(n^2)$.
- The function $8n + 5$ is $O(8n + 5)$.
- The function $8n + 5$ is $O(n^2)$.

We should use big-O in tightest and simplest terms.

Some Words of Caution ⚠

Does the following statement make sense?

- Since $n^2 - n = O(n^2)$ and $n^2 - 1 = O(n^2)$, we can say $n^2 - n = n^2 - 1$.
- An algorithm in $O(n)$ is always faster than one in $O(n^3)$.
- To search an item in an array, the time complexity is $O(1)$ in the best case.

Examples

Given a sequence S consisting of n numbers, we want to compute a sequence A such that $A[j]$ is a prefix average, that is

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$$

What is the time complexity of the following two algorithms?

```python
def prefix_average_1(S):
  n = len(S)
  A = [0] * n
  for j in range(n):
    total = 0
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j + 1)
  return A
```

```python
def prefix_average_2(S):
  n = len(S)
  A = [0] * n
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j + 1)
  return A
```

4. Other Notations

A comprehensive about algorithm analysis is out of the scope of this course.

- $\Theta(n)$
- $\Omega(n)$

Revisit Fibonacci

Conclusion

- Big O notation
- Evaluation through visualization

Homework 2 ✨✏️

- R-3.14
- R-3.25
- Plot the execution time as the function of n for R-3.25.