Analysis and Runtime

COMP2611: Data Structures
Sept 2019

Outline for Lecture

- Efficiency
- Time taken for algorithm to complete
  - Empirical Analysis (Apostori)
  - Theoretical Analysis (Apriori)
    - Basic of Asymptoptic Analysis Big-O analysis

Q: What does it mean for a programme to be efficient?

Notions of Efficiency

- Efficiency has many facets
  - Time taken to compute answer (runtime)
  - Amount of memory used
  - Amount of energy consumed
  - Bandwidth consumed
  - Number of computing units used
  - Engineering effort
- Can't optimise across every dimension of efficiency, e.g. space-time trade-off
- Usually we care the most about time

Empirical (Experimental) Analysis

- Design a experiment to measure runtime
- Generate realistic scenarios of application of an algorithm of varying sizes and complexity
- Code the algorithm and data structure
- Run code on different scenarios
  - Run code on each scenario multiple times
  - then find the mean (and std. deviation)
- Plot or otherwise analyse the results

Example of an Experimental Analysis

- Fibonacci sequence is recursively defined on $\mathcal{N}$ as follows:

$$F_n \begin{cases} n & \text{if } n \leq 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Algorithms for Fibonacci

```
function fib1(n)
    if n ≤ 1
        return n
    return fib1(n - 1) + fib2(n - 1)
```

Directly uses recursive Definition

```
function fib2(n)
    arr = init_array(n + 1)
    arr[0] = 0
    arr[1] = 1
    for i = 1 to n
        arr[i] = arr[i - 1] + arr[i - 2]
    return arr[n]
```

Uses a technique called memoization

Scenario for Fibonacci

- Consider $0 \leq n \leq 20$ as our experimental input
- Now we need to code our algorithms
  - Let's use Python **:)**

Code for Fibonacci in Python

```
def fib1(n):
    if n <= 1:
        return n
    return fib1(n - 1) + fib1(n - 2)
```

```
def fib2(n):
    if n <= 1:
        return n
    arr = [0] * (n + 1)
    arr[1] = 1
    for i in range(2, n + 1):
        arr[i] = arr[i - 1] + arr[i - 2]
    return arr[n]
```

Empirical timing in Python

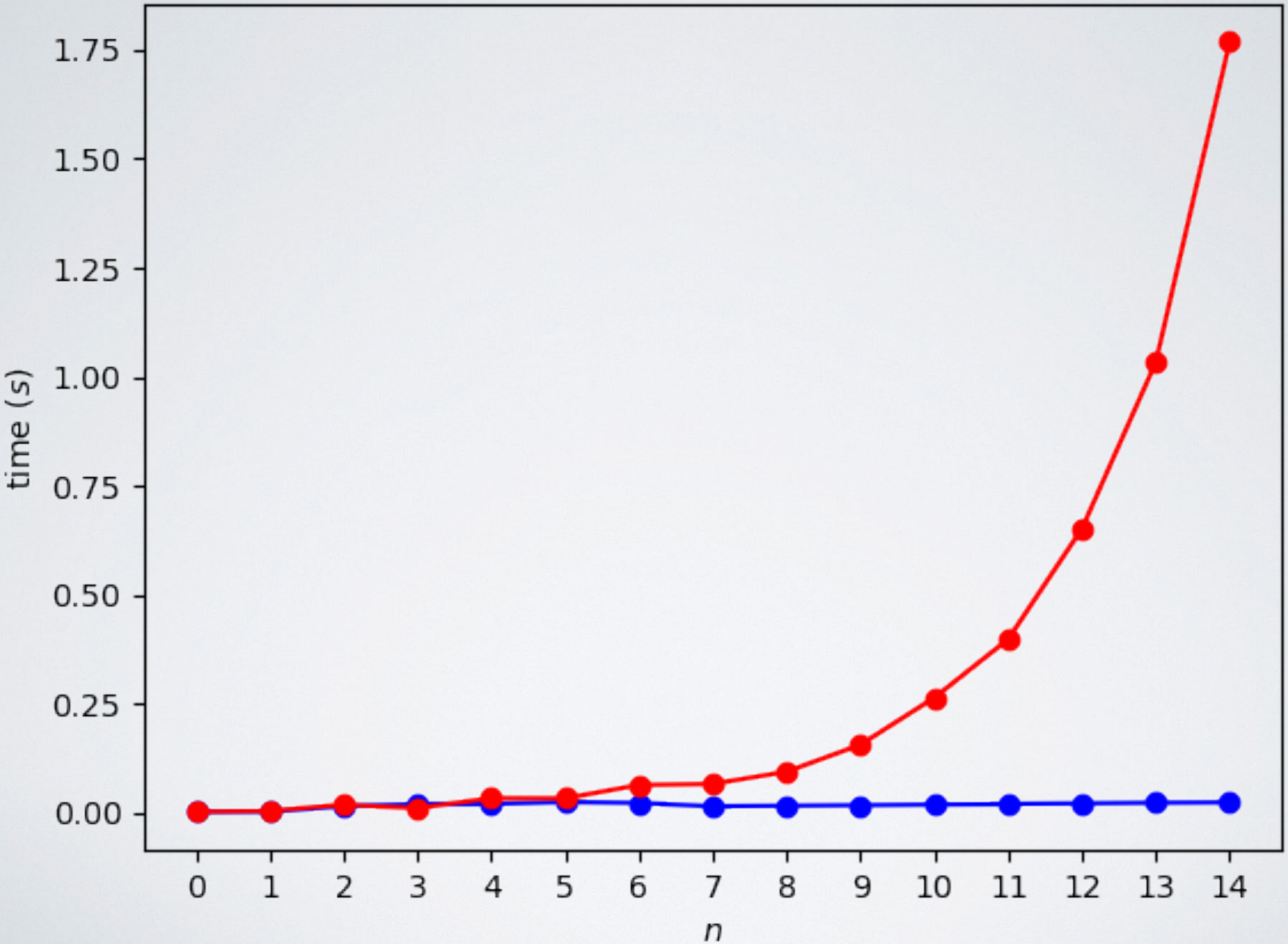
- Python has a useful module called *timeit*
- Has a function called *timeit* in *timeit* that runs a snippet multiple times and averages the time taken in **seconds**

```
timeit.timeit('fib2(10)', setup="from __main__ import fib2",number=10000)
```

Functional call
To Time

Context Data

Number of times
to execute


1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
time (s)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n

Q: What are the problems with this sort of empirical testing?

Problem: Lack of control

- Many confounding factors:
  - Language and ecosystem used
  - Operating system
  - Other processes on the system
  - Hardware issues
  - Skill of implementer
- Solution: Need way to compare runtime **without** actually needed to code algorithm
- Solution: We need a method that depends on the **intrinsic** properties of an algorithm

Illustrative Example

```
function array_product(arr)
    if length(arr) == 0:
        // cannot get product of empty array
        error()
    prod = arr[0]
    for i = 1 to (length(arr) - 1)
        prod = prod * arr[i]
    return prod
```

Knuth's Observation

- Experimental running time can be determined using
  - Time of each operation & frequency of each operation
- Example:
  - run **`array_product`** on array of size **`100`**

```
time(sum_array) = time(read)·100 + time(mult)·99 + time(comp)·1
                = 3ms·100 + 100ms·99 + 10ms·1
                = 10.21s
```

- **Key insight!**
  - the time an operation takes depends on hardware but…
  - *the number of times an operation is repeated does not depend on hardware*
  - So let's ignore time and only focus on *number of times* an operation is repeated

Knuth's Observation

- How do we ignore time?
  - we'll assume each operation takes `1` unit of time
- Example:
  - array_product on array of size `100`

```
time(sum_array) = time(read)·100 + time(add)·99 + time(comp)·1
                = 1·100 + 1·99 + 1·1
                = 100 reads + 99 adds + 1 comp
```

- Let's simplify and just report total number of operations
  - time`(array_product) = 200 ops`

Towards **Algorithmic** Running Time

- Problem #1
  - experimental running time depends on hardware
  - solution: *focus on number of operations*

Elementary Operations

- Most algorithms make use of standard "elementary" operations:
  - Math: `+,-,*,/,max,min,log,sin,cos,abs,...`
  - Comparisons: `==,>,<,≤,≥`
  - Variable assignment
  - Variable increment or decrement
  - Array allocation
  - Creating a new object
  - Function calls and value returns
  - Careful: an object's constructor & function calls may have elementary ops too!
- In practice all these operations take different amounts of time but
  - **we will assume each operation takes 1 unit of time**

What is Running Time?

"Running time"
=
Number of elementary operations

Running time ≠ Experimental time

A Simple Algorithm

```
function array_product(arr)
    if length(arr) == 0:
        return error
    prod = 1
    for i = 0 to (length(arr) - 1)
        prod = prod * arr[i]
    return prod
```

1op

1op

1op

1op per

3ops per

1op

- Do we count **`return error`**?
  - If we have any empty array, we have **`2 ops`**
  - If we have an array of size **`n`**, we have **`4n+4 ops`**

Towards **Algorithmic** Running Time

- Problem #1
  - experimental running time depends on hardware
  - solution: *focus on number of operations*
- Problem #2
  - number of operations depends on input
  - solution: *focus on number of operations for worst-case input*

A Simple Algorithm

```
function array_product(arr)
    if length(arr) == 0:
       return error
    prod = arr[0]
    for i = 1 to (length(arr) - 1)
       prod = prod * arr[i]
    return prod
```

1op

1op

1op

1op per

3ops per

1op

- What is the worst-case input for our algorithm?
  - any array that is non-empty
  - we'll just ignore "**`return error`**"

What is Running Time?

Worst-case running time
=
Number of elementary operations on worst-case input

A Simple Algorithm

```
function array_product(arr)
    if length(arr) == 0:
        return error
    prod = arr[0]
    for i = 1 to (length(arr) - 1)
        prod = prod * arr[i]
    return prod
```

1op

1op

1op

1op per

3ops per

1op

- How many times does the loop run?
  - Depends on size of the array

Towards an **Algorithmic** Running Time

- Problem #1
  - experimental running time depends on hardware
  - solution: *focus on* ***number*** *of operations (Knuth's observation)*
- Problem #2
  - number of operations depends on input
  - solution: *focus on number of operations on* ***worst-case*** *input! Why?*
- Problem #3
  - number of operations depends on input size
  - solution: *focus on number of operations as a function of* ***input size n***.

A Simple Algorithm

```
function array_product(arr)
    if length(arr) == 0:
        return error
    prod = arr[0]
    for i = 1 to (length(arr) - 1)
        prod = prod * arr[i]
    return prod
```

1op

1op

1op

1op per

3ops per

1op

- How many times does the loop run?
  - Depends on size of the array
  - Let `n = length(array)`
  - **`array_product`** takes `4n+4 ops`

What is Running Time?

Worst-case running time
=
T(n): *Number of elementary operations*
on worst-case input
as a function of input size n

Constant Time

```
function foobar(arr)
    return (arr[0] * 2)
```

3 ops

- How many basic operations are performed
  - `T(n) = 3 ops`
  - Number of operations independent of array size

Attempt

```
function argmin(arr)
    if length(array) == 0:
        return error
    min_index = 0
    for i = 1 to length(arr):
        if arr[min_index] > arr[i]:
            min_index = i
    return min_index
```

- How many basic operations are performed
  - `T(n) = ?`

Attempt

```
function argmin(arr)
    if length(array) == 0:
       return error
    min_index = 0
    for i = 1 to length(arr):
       if arr[min_index] > arr[i]:
          min_index = i
    return min_index
```

1op

1op

1op

1 op per

3 op per

1 op

1 op

- How many basic operations are performed
  - `T(n) =`

Attempt

```
function argmin(arr)
    if length(array) == 0:
       return error
    min_index = 0
    for i = 1 to length(arr):
       if arr[min_index] > arr[i]:
          min_index = i
    return min_index
```

1op

1op

1op

1 op per

3 op per

1 op

1 op

- How many basic operations are performed
  - `T(n) =`

Attempt

```
function argmin(arr)
    if length(array) == 0:
       return error
    min_index = 0
    for i = 1 to length(arr):
       if arr[min_index] > arr[i]:
          min_index = i
    return min_index
```

1op

1op

1op

1 op per

3 op per

1 op

1 op

- How many basic operations are performed
  - `T(n) =`

Linear Runtime

```
function argmin(arr)
    if length(array) == 0:
        return error
    min_index = 0
    for i = 1 to length(arr):
        if arr[min_index] > arr[i]:
            min_index = i
    return min_index
```

1op

1op

1op

1 op per

3 op per

1 op

1 op

- How many basic operations are performed
  - $T(n) = (1 + 1 + 1) + (1 + 3 + 1)n$
    $= 5n + 3$

Attempt

```
function all_pairs_prod_sum(arr)
    if length(array) == 0:
        return error
    acc = 0
    for i = 1 to length(arr):
        for j = 1 to length(arr):
            acc = acc + (arr[i] * arr[j])
    return arr
```

- How many basic operations are performed
  - `T(n) = ?`

Attempt

```
function all_pairs_prod_sum(arr)
    if length(array) == 0:
       return error
    acc = 0
    for i = 1 to length(arr):
       for j = 1 to length(arr):
          acc = acc + (arr[i] * arr[j])
    return arr
```

1op

1op

1op

1op

1 op per

1 op per

4 op per

1op

- How many basic operations are performed
  - `T(n) = ?`

Attempt

```
function all_pairs_prod_sum(arr)
    if length(array) == 0:
       return error
    acc = 0
    for i = 1 to length(arr):
       for j = 1 to length(arr):
          acc = acc + (arr[i] * arr[j])
    return arr
```

1op

1op

1op

1 op per

1 op per

4 op per

1op

- How many basic operations are performed
  - $T(n) = (1 + 1 + 1) + (1 + 4)n^2 + n$
  - $= 5n^2 + n + 3$

Running Times

Constant

independent of input size

Linear

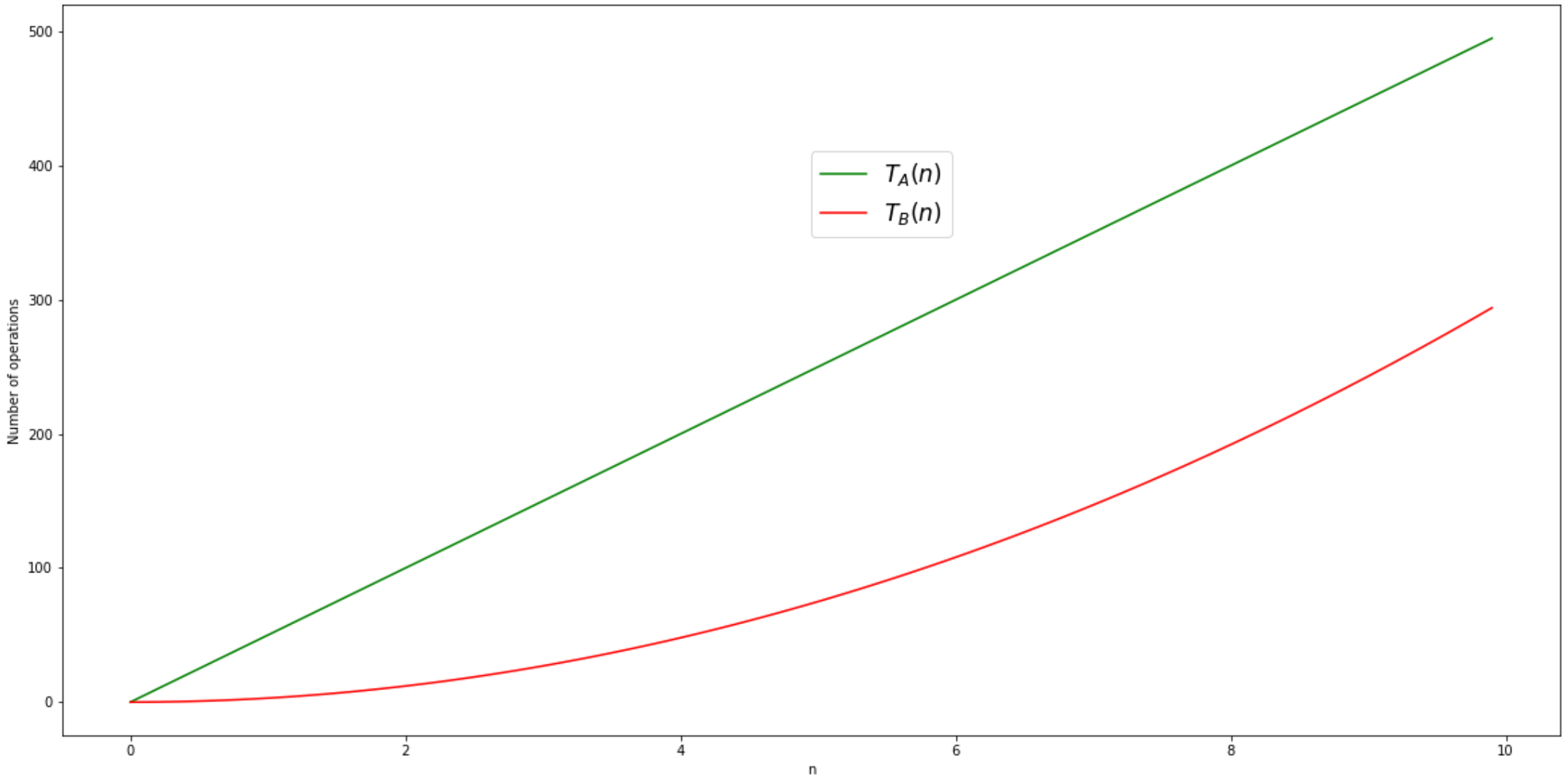
depends on input size

Quadratic

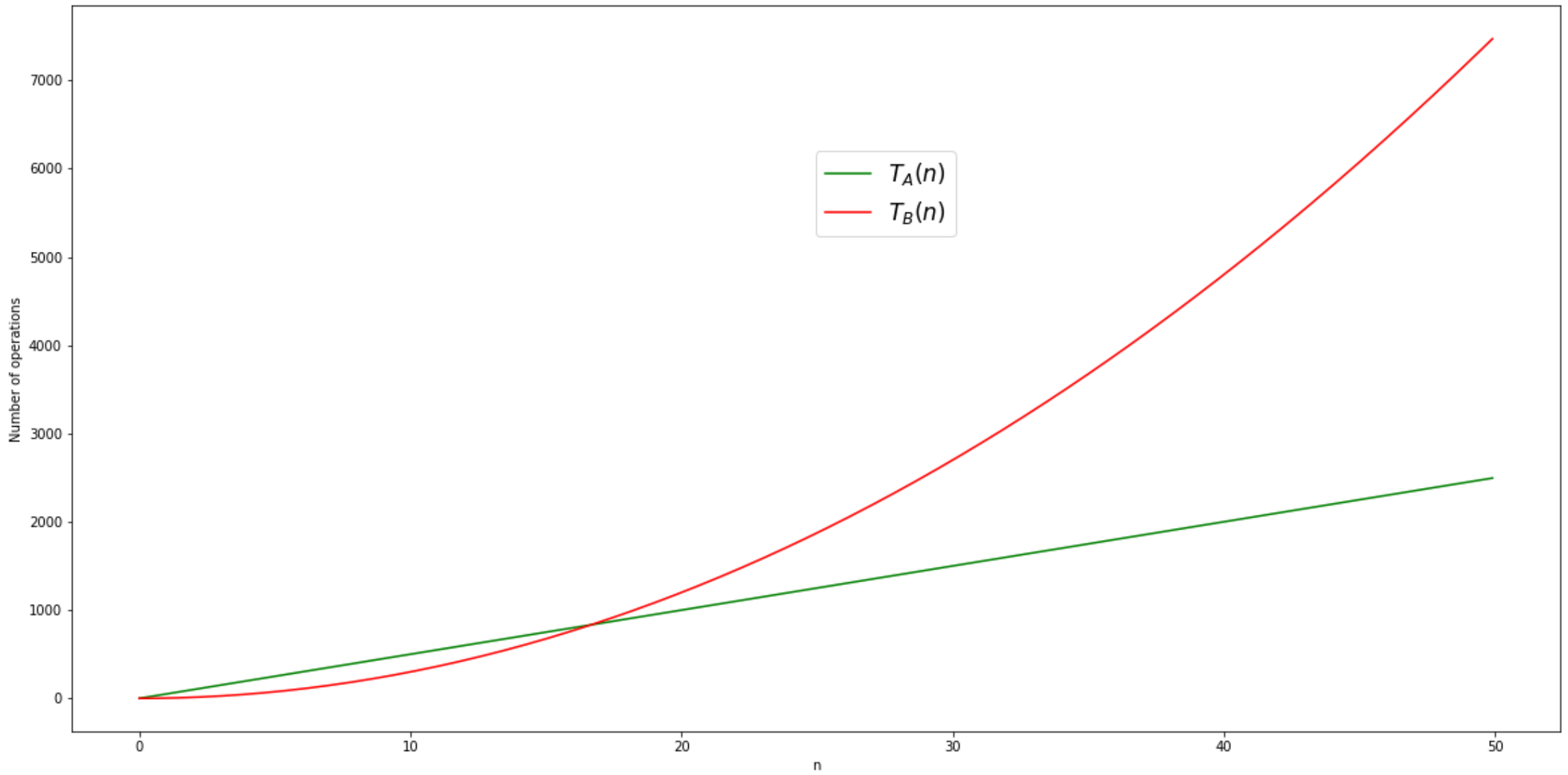
depends on square of input size

Comparing runtimes

- Suppose that algorithm a has runtime $T_A(n)$
- Suppose that algorithm a has runtime $T_B(n)$
- How do we compare them?
  - Number of operations depends on n
- Consider $T_A(n) = 50n$
- Consider $T_B(n) = 3n^2$



When n is small, $T_B(n) < T_A(n)$



When **n** gets larger, $T_B(n) > T_A(n)$

Comparing runtimes

- Suppose that algorithm a has runtime $T_A(n)$
- Suppose that algorithm a has runtime $T_B(n)$
- How do we compare them?
  - Number of operations depends on n
- Consider $T_A(n) = 50n$
- Consider $T_B(n) = 3n^2$
- We care about when n grows larger. Why?

What is Running Time?

Asymptotic worst-case running time
=
Number of elementary operations
on worst-case input
as a function of input size n
when n tends to infinity

In CS "running time" usually means asymptotic worst-case running time…but not always!
we will learn about other kinds of running times

Comparing Running Times

Comparing *asymptotic* running times
=
$T_A(n)$ is better than $T_B(n)$ if
for large enough n
$T_A(n)$ grows slower than $T_B(n)$

Q: can we formalize all this mathematically?

Big-O

Definition (Big-O): $T_A(n)$ is $O(T_B(n))$ if there exists positive constants c and n_0 such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s order of growth is at most $T_B(n)$'s order of growth
- Examples
  - $2n+10$ is $O(n)$
  - $n^{10}+2019$ is $O(n^{10})$ and also $O(n^{50})$
  - We want **tight** bounds. Loose bounds are uninformative

Big-O

- How do we find "the Big-O of something"?
  - Usually you "eyeball" it
  - Then you try to prove it (not in this course)

Big-O Examples

Definition (Big-O): $T_A(n)$ is $O(T_B(n))$ if there exists positive constants c and n_0 such that:

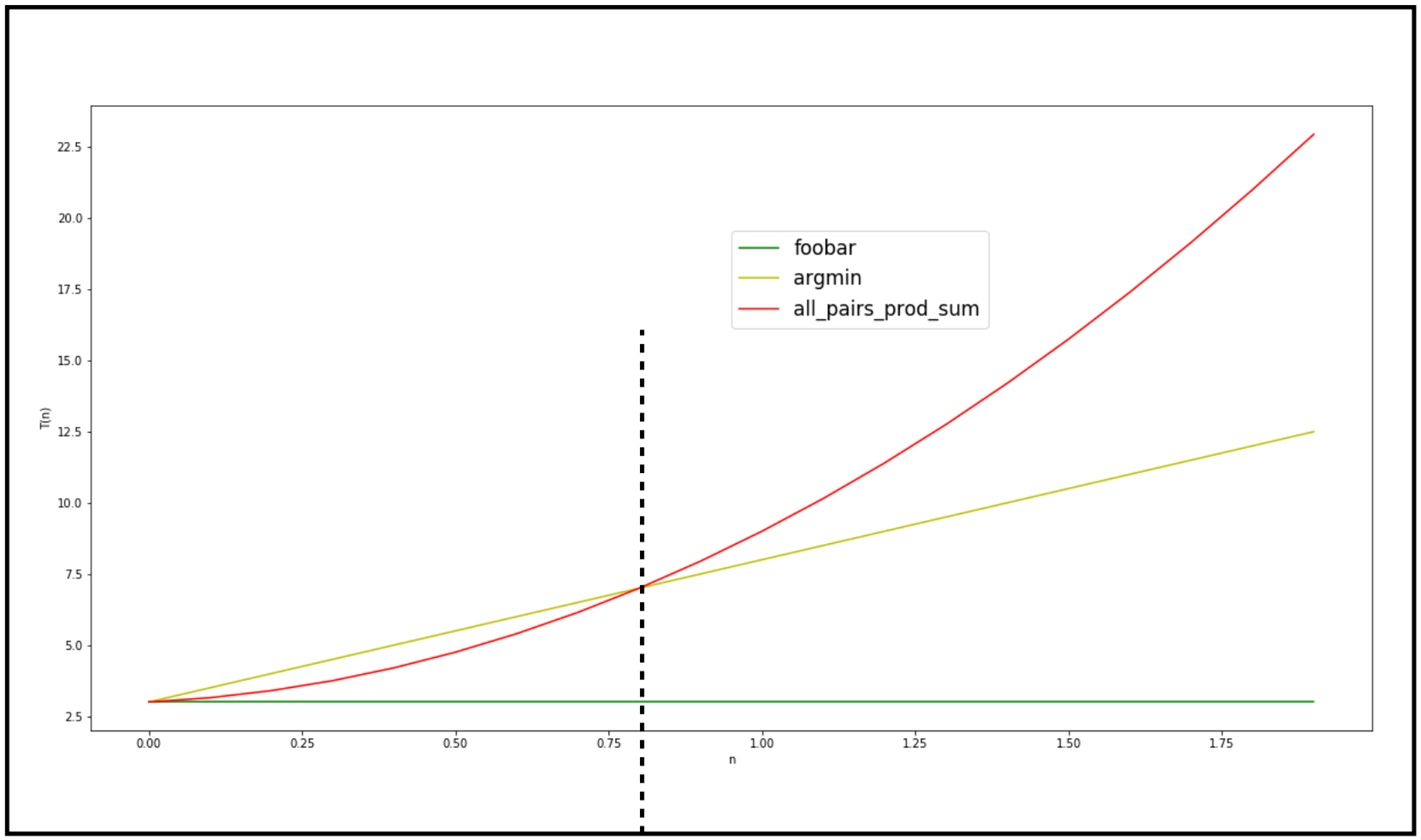
$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $2n+10$ is $O(n)$
  - for example, choose $c=3$ and $n_0=10$
- Why? because
  - $2n+10 \leq 3 \cdot n$ when $n \geq 10$
  - for example, $2 \cdot 10+10 \leq 3 \cdot 10$

Plotting Running Times

T(n)



n_0

n

We don't care what happens here

We only care what happens here

Eyeballing Big-O

- If **`T(n)`** is a polynomial of degree **`d`** then **`T(n)`** is **$O(n^d)$**
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - **$1000n^2+400n+739$** is **$O(n^2)$**
  - **$n^{80}+43n^{72}+5n+1$** is **$O(n^{80})$**
- *For the Big-O, use the smallest upper bound*
  - **`2n`** is **$O(n^{50})$** but that's not really a useful bound
  - instead it is better to say that **`2n`** is **`O(n)`**

Example Big-O Analysis

- Given algorithm, find number of ops as a function of input size
  - foobar: $T(n)=3$
  - argmin: $T(n)=5n+2$
  - all_pairs_prod_sum: $T(n)=5n^2+n+3$
- Replace constants with "c" (they are irrelevant as n grows)
  - first: $T(n)=c$
  - argmax: $T(n)=c_0n+c_1$
  - possible_products: $T(n)=c_0n^2+n+c_1$

Example Big-O Analysis

- Discard constants & use smallest possible degree
  - first: $T(n)=c$ is $O(1)$
  - argmax: $T(n)=c_0n+c_1$ is $O(n)$
  - possible_products: $T(n)=c_0n^2+n+c_1$ is $O(n^2)$
- The convention for $T(n)=c$ is to write $O(1)$

Big-O

Definition (Big-O): $T_A(n)$ is $O(T_B(n))$ if there exists positive constants c and n_0 such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s growth rate is upper bounded by $T_B(n)$'s growth rate
- But what if we need to express a lower bound?
  - we use Big-Ω notation!

Running Times

O(1)
independent of input size

O(n)
depends on input size

O(n²)
depends on square of input size

O(n³)
depends on cube of input size

O(n⁷⁰)
depends on 70th power of input size

O(2ⁿ)
exponential in input size

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Readings

- Asymptotic runtime and Big-O
  - Dasgupta et al. section 0.3 (pp. 15-17)
  - Sedwick

References

- Some material adapted from Brown's CS16
- Some material drawn from the Cormen and Sedwick