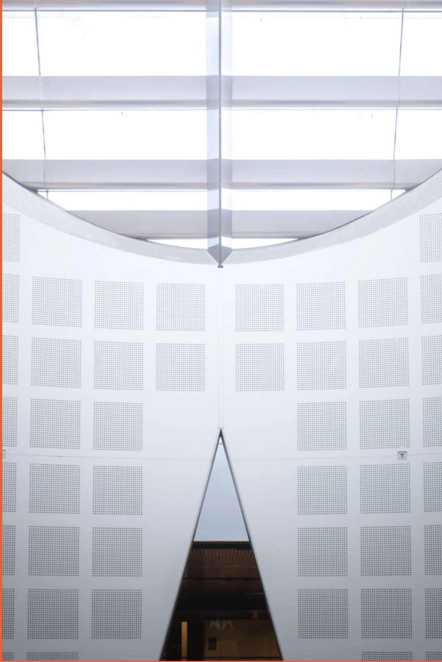
Data Structures and Algorithms

Algorithm analysis

Presented by

Julián Mestre
School of Computer Science



THE UNIVERSITY OF SYDNEY

Three abstractions

Computational problem:

- defines a computational task
- specifies what the input is and what the output should be

Algorithm:

- a step-by-step recipe to go from input to output
- different from implementation

Correctness and complexity analysis:

- a formal proof that the algorithm solves the problem
- analytical bound on the resources it uses

Example computational problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], ..., A[n-1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i+1] + \cdots + A[j]$$

Efficiency

Definition (first attempt)

An algorithm is efficient if it runs quickly on real input instances

Not a good definition beause it is not easy to evaluate:

- instances considered
- implementation details
- hardware it runs on

Our definition should implementation independent:

- count number of "steps"
- bound the algorithm's worst-case performance

Efficiency

Definition (second attempt)

An algorithm is efficient if it achieves qualitatively better worst-case performance than a brute-force approach

Not a good definition because it is subjective:

- brute-force approach is ill-defined
- qualitatively better is ill-defined

Our definition should be objective:

- not tied to a strawman baseline
- independently agreed upon

Efficiency

Definition
An algorithm is efficient if it runs in polynomial time; that is, on an instance of size n, it performs no more than $p(n)$ steps for some polynomial $p(x) = a_d x^d + \cdots + a_1 x + a_0$.

This gives us some information about the expected behavior of the algorithm and is useful for making predictions and comparing different algorithms.

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of size n.

If $T(n)$ is a polynomial of degree d, then doubling the size of the input should roughly increase the running time by a factor of 2^d.

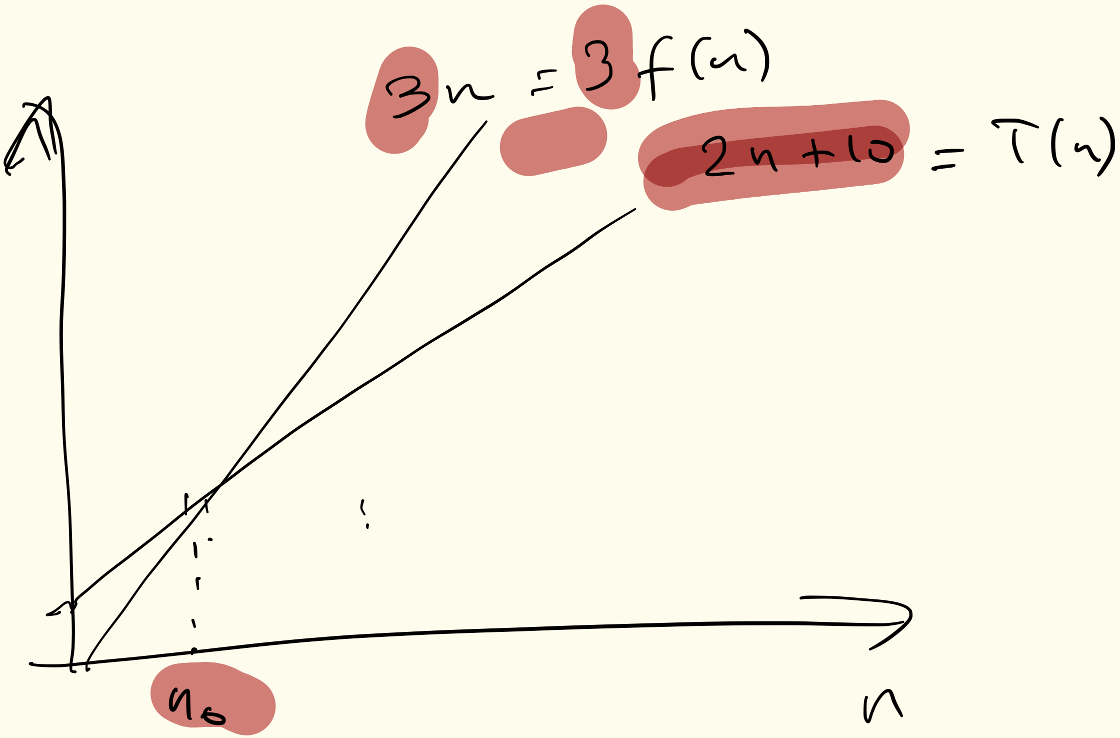
Asymptotic growth analysis gives us a tool for focusing on the terms that make up $T(n)$, which dominate the running time

Asymptotic growth analysis

Definition
We say that $T(n) = O(f(n))$ if
there exists $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$

Definition
We say that $T(n) = \Omega(f(n))$ if
there exists $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$

Definition
We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$



We say $\left.\begin{array}{l} T(n) = O(n) \\ T(n) = \Omega(n) \end{array}\right\} T(n) = \Theta(n)$

Examples of asymptotic growth

Polynomial

Logarithmic

Exponential

Comparison of running times

size	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$<$ 1s	$<$ 1s	$<$ 1s	<1s	<1s	3s
50	$<$ 1s	$<$ 1s	$<$ 1s	<1s	17m	-
100	$<$ 1s	$<$ 1s	$<$ 1s	1s	35y	-
1,000	$<$ 1s	$<$ 1s	1s	15m	-	-
10,000	$<$ 1s	$<$ 1s	2s	11d	-	-
100,000	$<$ 1s	1s	2h	31y	-	-
1,000,000	1s	10s	4d	-	-	-

Properties of asymptotic growth

Transitivity:

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$

Sums of functions:

- If $f = O(g)$ and $g = O(h)$ then $f + g = O(h)$
- If $f = \Omega(h)$ then $f + g = \Omega(h)$

Asymptotic analysis is a powerful tool that allows us to ignore unimportant details and focus on what's important.

$$T(n) = 4n^2 + 10n + 20$$

Survey of common running times

Let $T(n)$ be the running time of our algorithm.

We say that $T(n)$ is ...	if ...
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

Recall stock trading problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], ..., A[n-1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i+1] + \cdots + A[j]$$

Naive algorithm

```
def naive(A)
    n ← length of A
    best_so_far = 0
    buy ← sell ← -1
    for i in range(0, n)                  ← n times
        for j in range(i, n)              ← ≤ n times
            if best_so_far < eval(i,j)
                buy ← i
                sell ← j
                best_so_far ← eval(i,j)
    return buy, sell
```

$O(n)$

```
def eval(i, j)
    return A[i] + ... + A[j]
```

$O(n^2)$ time w/ preproc

Overall it runs in $O(n^3)$ time

Naive with preprocessing

```
def eval(i, j)
    return B[j] - B[i-1]
```

$\}\ O(1)$

```
B = copy of A
for i in range(1, n)
    B[i] = B[i-1] + A[i]
```

$\}\ O(n)$

$$B[i] = A[0] + A[1] + \dots + A[i]$$

$$A[i] + A[i+1] + \dots + A[j] = -B[i-1] + B[j]$$

Reuse computation

Recap

Asymptotic time complexity gives us some information about the expected behavior of the algorithm. It is useful for making predictions and comparing different algorithms.

Why do we make a distinction between problem, algorithm, implementation and analysis?

- somebody can design a better algorithm for a given problem
- somebody can come up with better implementation
- somebody can come up with better analysis

This week

Tutorial sheet 1:

- posted on Wed 27 Feb
- make sure you work out a few problems before the tutorial

- posted on Wed 27 Feb
- due on Tue 5 Mar