Algorithms and Complexity / (Adv)

Algorithm Analysis

Julián Mestre

School of Information Technologies
The University of Sydney

THE UNIVERSITY OF SYDNEY

Three abstractions

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

A computational problem

Motivation

- We have collected information about the daily fluctuation of a stock's price, which we have recently bought and sold
- We want to evaluate our performance against the best possible outcome

Input:

- An array with n integer values $A[0], A[1], \ldots, A[n-1]$

Task:

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

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

Naive algorithm

```
def naive(A):

  def evaluate(A,a,b)
     return A[a] + ... + A[b]

  n = size of A
  answer = (0,0)
  for i = 0 to n-1
    for j = i to n-1
      if evaluate(A,i,j) > evaluate(A,answer[0],answer[1])
         answer = (i,j)
 return answer
```

Questions:

- how efficient is this algorithm?
- is this the best algorithm for this task?

Efficiency

Def. 1: An algorithm is efficient if it runs quickly on real input instances

Not a good definition because it depends on

- how big our instances are
- how restricted/general our instance are
- implementation details
- hardware it runs on

A better definition would be implementation independent:

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

Efficiency

<u>*Def. 2*</u>*:* An algorithm is *efficient* if it achieves (analytically) qualitatively better worst-case performance than a brute-force approach.

This is better but still has some issues:

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

Efficiency

<u>Def. 3</u>: An algorithm is efficient if it runs in polynomial time; that is, on an instance of size n, it performs $p(n)$ steps for some polynomial $p(x)= a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0$

Notice that if we double the size of the input, then the running time would roughly increase by a factor of 2^d.

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

Comparison of running times

size	n	n log n	n^2	n^3	2^n	n!
10	<1 s	<1 s	<1 s	<1 s	<1 s	3 s
30	<1 s	<1 s	<1 s	<1 s	17 m	WTL
50	<1 s	<1 s	<1 s	<1 s	35 y	WTL
100	<1 s	<1 s	<1 s	1 s	WTL	WTL
1000	<1 s	<1 s	1 s	15 m	WTL	WTL
10,000	<1 s	<1 s	2 m	11 d	WTL	WTL
100,000	<1 s	1 s	2 h	31 y	WTL	WTL
1,000,000	1 s	10 s	4 d	WTL	WTL	WTL

WTL = way too long

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of "size" n. We say that $T(n) = O(\ f(n)\ )$ if

there exist n_0 and $c > 0$ such that $T(n) \leq c\ f(n)$ for all $n > n_0$

Also, we say that $T(n) = \Omega(\ f(n)\ )$ if

there exist n_0 and $c > 0$ such that $T(n) > c\ f(n)$ for all $n > n_0$

Finally, we say that $T(n) = \Theta(\ f(n)\ )$ if

$T(n) = O(\ f(n)\ )$ and $T(n) = \Omega(\ f(n)\ )$

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

Properties of asymptotic growth

Let $T(n) = a_d n^d + \cdots + a_0$ be a poly. with $a_d > 0$, then $T(n) = \Theta(n^d)$

Let $T(n) = \log_a n$ for constant $a > 1$, then $T(n) = \Theta(\log n)$

For every $b > 1$ and $d > 0$, we have $n^d = O(b^n)$

The reason we use asymptotic analysis is that allows us to ignore unimportant details and focus on what's important, on what dominates the running time of an algorithm.

Survey of common running times

Let n be the size of the input, and let T(n) be the running time of our algorithm.

We say T(n) is…	if…
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
"almost" linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$ for some $c > 1$

Recap: Asymptotic analysis

Establish the asymptotic number of "steps" our algorithm performs in the worst case

Each "step" represents constant amount of real computation

Asymptotic analysis provides the right level of detail

Efficiency = polynomial running time

Keep in mind hidden constants inside your O-notation

Naive algorithm

```
def naive(A):

  def evaluate(A,a,b)
     return A[a] + ... + A[b]

  n = size of A
  answer = (0,0)
  for i = 0 to n-1
    for j = i to n-1
      if evaluate(A,i,j) > evaluate(A,answer[0],answer[1])
        answer = (i,j)
  return answer
```

Obs.

`naive` runs in $\Theta(n^3)$ time

Pre-processing

Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

The rest is as before

```
def preprocessing(A):

  def evaluate(B,a,b)
     return B[a] - B[b+1]

  n = size of A
  B = array of size n+1
  for i in 0 to n-1
    B[i] = A[i] + ... A[n-1]
  B[n] = 0
  ⋮
```

Obs.

`preprocessing` runs in $\Theta(n^2)$ time

Reuse computation

Imagine trying to find the best index i for a fixed index j:

$$OPT[j] = \text{argmax}_{\,i \leq j}\, B[i]$$

But we can also express $OPT[j]$ recursively in a way that allows us to compute, in $O(n)$ time, $OPT[j]$ for all j

Finally, in $O(n)$ time, find j maximizing $B[OPT[j]] - B[j+1]$

Obs.

There is an $\Theta(n)$ time algorithm for finding the optimal investment window

Experimental algorithms

Some times we can get a rough idea of the asymptotic running of an algorithm by doing doubling experiments.

First run the algorithm on instances whose size are powers of 2

If we suspect that $T(n)$ is polynomial of unknown degree d, then plot $T(2n)/T(n)$. It should converge to 2^d

If you suspect that $T(n) = \Theta(f(n))$, then plot $T(n)/f(n)$. It should converge to a constant > 0

Recap: Algorithm analysis

`naive` runs in $\Theta(n^3)$ time

`preprocessing` runs in $\Theta(n^2)$ time

With a bit of ingenuity we can solve the problem in $\Theta(n)$ time

Some times experiments can confirm asymptotic analysis

Why we separate problem, algorithm, and analysis?

- somebody can design a better algorithm to solves a given problem
- somebody can give a tighter analysis of an old algorithm

This week

Quiz 0

- 15 minutes long, during tutorial
- It won't count as assessment. It's just to learn about your math background.

Tutorial Sheet 1:

- posted on Monday 27 July
- make sure you work on it before the tutorial

Assignment 1:

- posted on Monday 27 July, due next Monday