Algorithms and Complexity

Algorithm Analysis Review

Julián Mestre

School of Information Technologies
The University of Sydney

Efficiency

First attempt: An algorithm is efficient if it can be implemented to run quickly on the instances we are trying to solve

Performance of our implementation will depend on

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

A better definition would be implementation independent

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

Efficiency

An algorithm is *efficient* if it achieves (analytically) qualitatively better worst-case performance than a brute-force approach.

We consider an algorithm efficient if it runs in polynomial; that is, on an instance of size n, the algorithm perform $p(n)$ number of step for some fixed polynomial $p(x)= a_d\, x^d + a_{d-1}\, x^{d-1} + \cdots + a_0$, where $a_d > 0$

If we double the size of the input, then the running time would roughly increase by a factor of 2^d

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 order of growth

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 $a > 0$, then $T(n) = \Theta(\log n)$

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

The main reason we use asymptotic analysis is that allows us to ignore unimportant details and focus on what's important.

Survey of common running times

Let n be the size of the input, and let $T(n)$ be the running time of our algorithm. Then we say that $T(n)$ is

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

Recap: Asymptotic analysis

Find out 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

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

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

First algorithm: Brute force

```
def find_optimal_window(A):

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

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

Obs.

This algorithm runs in $O(n^3)$ time

Second attempt: Pre-processing

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

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

Run the rest of the algorithm as before

```
def find_optimal_window_faster(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.

This algorithm runs in $O(n^2)$ time

Third attempt: 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

Recap: Algorithm analysis

Given fluctuation number, find optimal window for investment

Brute force algorithm runs in $\Theta(n^3)$ time

Brute force with twist runs in $\Theta(n^2)$ time

But a bit of ingenuity the problem can be solved in $\Theta(n)$ time

Doubling experiments can give us an indication of the asymptotic running time of an algorithm

Data structures

A data structure is a way of organizing your data so as to allow us to perform certain computational tasks more efficiently

Example: In Python we have been using "lists", which supports:

- `list(A)` : returns a new list with the items of `A`
- `len(A)` : returns the length of the list
- `A[i]` : returns the `(i+1)`st element in the list # make sure `i < len(A)` !
- `A.append(x)` : adds `x` at the end of `A`
- `A.pop(x)` : removes and returns last element of `A`

Priority queue

A data structure for storing n numbers supporting:

- `priority_queue(A)` : returns a new queue with the items of `A`
- `insert(x,Q)`: adds `x` to the queue `Q`
- `get_min(Q)` : removes and returns the minimum element in `Q`
- `empty(Q)` : returns `True` if the queue is empty

Naive implement using an array:

- `priority_queue(A)` runs in $\Theta(|A|)$ time
- `Q.insert(x)` runs in $\Theta(1)$ time
- `Q.get_min()` runs in $\Theta(|Q|)$ time
- `empty(Q)` runs in $O(1)$ time

Applications: Sorting

Priority queues have many applications, among them, sorting

The algorithm runs in $\Theta(n^2)$ time.

Can we do better than that?

- Use different algorithm
- Use better data structure

```
def sort(A):

  Q = priority_queue(A)
  i = 0
  while not Q.empty():
    A[i] = min(Q)
    i = i + 1
```

O(n)

$O(n^2)$

O(n)

Priority queue with heaps

We could implement a priority queue with a heap:

- `priority_queue(A)` would run in $\Theta(|A|)$ time
- `Q.insert(x)` runs in $\Theta(\log |Q|)$ time
- `Q.get_min()` runs in $\Theta(\log |Q|)$

```
def sort(A):

  Q = priority_queue(A)
  i = 0
  while not Q.empty():
    A[i] = min(Q)
    i = i + 1
```

O(n)

O(n log n)

O(log n)

Recap: Data structure

DS encourage code modularity and re-usability

There are libraries offering highly tuned implementations

There are many other data structures for priority queues:

- Heap (the one you should know)
- Binomial heap
- Fibonacci heap
- Pairing heaps
- Etc

This week

Quiz 1

- 15 minutes long
- during your tutorial session

Problem set 1:

- will be posted later today (Monday 30 July)
- make sure you work on it before the tutorial

Assignment 1:

- will be posted tomorrow (Tuesday 1 August)
- will be due one week later