Analysis of Algorithms

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2020/21

version: 3941119 @2021-10-26

What are we analyzing?

- So far, we frequently asked: 'can we do better?'
- Now, we turn to the questions of
  - what is better?
  - how do we know an algorithm is better than the other?
- There are many properties that we may want to improve
  - correctness
  - robustness
  - simplicity
  - ...
  - In this lecture, *efficiency* will be our focus
    - in particular time efficiency/complexity

How to determine running time of an algorithm?

write the code, experiment

- A possible approach:
  - Implement the algorithm
  - Test with varying input
  - Analyze the results
- A few issues with this approach:
  - Implementing something that does not work is not productive (or fun)
  - It is often not possible cover all potential inputs
  - If your version takes 10 seconds less than a version reported 10 years ago, do you really have an improvement?
- A formal approach offers some help here

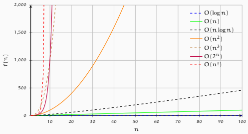
Some functions to know about

Family	Definition
Constant	$f(n) = c$
Logarithmic	$f(n) = \log_b n$
Linear	$f(n) = n$
N log N	$f(n) = n \log n$
Quadratic	$f(n) = n^2$
Cubic	$f(n) = n^3$
Other polynomials	$f(n) = n^k$, for $k > 3$
Exponential	$f(n) = b^n$, for $b > 1$
Factorial	$f(n) = n!$

- We will use these functions to characterize running times of algorithms

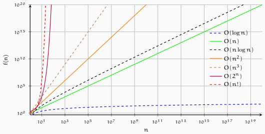
Some functions to know about

the picture - why we care about their difference



Some functions to know about

the bigger picture



A few facts about logarithms

- Logarithm is the inverse of exponentiation:

$$x = \log_b n \iff b^x = n$$

- We will mostly use base-2 logarithms. For us, no-base means base-2
- Additional properties:

$$\log xy = \log x + \log y$$
$$\log \frac{x}{y} = \log x - \log y$$
$$\log x^a = a \log x$$
$$\log_b x = \frac{\log_k x}{\log_k b}$$

- Logarithmic functions grow (much) slower than linear functions

Polynomials

- A degree-0 polynomial is a constant function ($f(n) = c$)
- Degree-1 is linear ($f(n) = n + c$)
- Degree-2 is quadratic ($f(n) = n^2 + n + c$)
- ...
- We generally drop the lower order terms (soon we'll explain why)
- Sometimes it will be useful to remember that

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

Combinations and permutations

- $n! = n \times (n-1) \times \ldots \times 2 \times 1$
- Permutations:

$$P(n,k) = n \times (n-1) \times \ldots \times (n-k-1) = \frac{n!}{(n-k)!}$$

- Combinations 'n choose k':

$$C(n,k) = \binom{n}{k} = \frac{P(n,k)}{P(k,k)} = \frac{n!}{(n-k)! \times k!}$$

Proof by induction

- Induction is an important proof technique
- It is often used for both proving the correctness and running times of algorithms
- It works if we can enumerate the steps of an algorithm (loops, recursion)
  - Show that base case holds
  - Assume the result is correct for n, show that it also holds for $n + 1$

Proof by induction

Example: show that $1 + 2 + 3 + \ldots + n = n(n+1)/2$

- Base case, for n=1

$$(1 \times 2)/2 = 1$$

- Assuming

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

we need to show that

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Formal analysis of running time of algorithms

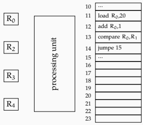
- We are focusing on characterizing running time of algorithms
- The running time is characterized as a function of input size
- We are aiming for an analysis method
  - independent of hardware / software environment
  - does not require implementation before analysis
  - considers all possible inputs

How much hardware independence?

quite, but not completely: we assume a RAM model of computing

- Characterized by random access memory (RAM) (e.g., in comparison to a sequential memory, like a tape)
- We assume the system can perform some primitive operations (addition, comparison) in constant time
- The data and the instructions are stored in the RAM
- The processor fetches them as needed, and executes following the instructions
- This is largely true for any computing system we use in practice

RAM model: an example



- Processing unit performs basic operations in constant time
- Any memory cell with an address can be accessed in equal (constant) time
- The instructions as well as the data is kept in the memory
- There may be other, specialized registers
- Modern processing units also employ a 'cache'

Formal analysis of running time

- Simply count the number of primitive operations
- Primitive operations include:
  - Assignment
  - Arithmetic operations
  - Comparing primitive data types (e.g., numbers)
  - Accessing a single memory location
  - Function calls, return from functions
- Not primitive operations:
  - loops, recursion
  - comparing sequences

Focus on the worst case

- Algorithms are generally faster on certain input than others
- In most cases, we are interested in the *worst case* analysis
  - Guaranteeing worst case is important
  - It is also relatively easier: we need to identify the worst-case input
- Average case analysis is also useful, but
  - requires defining a distribution over possible inputs
  - often more challenging

Counting primitive operations

example: nearest points, the naive algorithm

```
def shortest_distance(points):
    n = len(points)                          # 1 (constant?)
    min = 0                                  # 1 (constant)
    for i in range(n):                       # n times
        for j in range(i):                   # i times
            d = distance(points[i], points[j])  # 1 (constant)
            if min > d:                      # 1 (constant)
                min =  d                     # 1 (constant)
    return min                               # 1 (constant)
```

$$
\begin{aligned}
T(n) &= 2 + (1 + 2 + 3 + \ldots + n - 1) \times 3 + 1 \\
&= 3 \times \frac{(n-1)(n-2)}{2} + 3
\end{aligned}
$$

Big-O notation

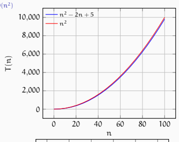
- Big-O notation is used for indicating an upper bound on running time of an algorithm as a function of running time
- If running time of an algorithm is $O(f(n))$, its running time grows proportional to $f(n)$ as the input size n grows
- More formally, given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and integer $n_0 \geqslant 1$ such that

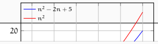
$$f(n) \leqslant c \times g(n) \text{ for } n \geqslant n_0$$

- Sometimes the notation $f(n) = O(g(n))$ is also used, but beware: this equal sign is not symmetric

Big-O example

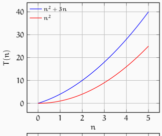
$T(n) = n^2 - 2n + 5$ is $O(n^2)$

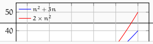




Big-O, another example

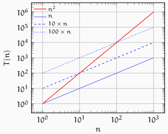
$T(n) = n^2 + 3n$ is $O(n^2)$





Big-O, yet another example

but n^2 is not $O(n)$ – proof by picture



Back to the function classes

Family	Definition
Constant	$f(n) = c$
Logarithmic	$f(n) = \log_b n$
Linear	$f(n) = n$
N log N	$f(n) = n \log n$
Quadratic	$f(n) = n^2$
Cubic	$f(n) = n^3$
Other polynomials	$f(n) = n^k$, for $k > 3$
Exponential	$f(n) = b^n$, for $b > 1$
Factorial	$f(n) = n!$

- None of these functions can be expressed as a constant factor of another

Rules of thumb

Drop the lower order terms

- In the big-O notation, we drop the constants and lower order terms
  - Any polynomial degree d is $O(n^d)$
    $10n^3 + 4n^2 + n + 100$ is $O(n^3)$
  - Drop any lower order terms:
    $2^n + 10n^3$ is $O(2^n)$
- Use the simplest expression:
  - $5n + 100$ is $O(5n)$, but we prefer $O(n)$
  - $4n^2 + n + 100$ is $O(n^3)$,
- Transitivity: if $f(n) = O(g(n))$, and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- Additivity: if both $f(n)$ and $g(n)$ are $O(h(n))$ $f(n) + g(n)$ is $O(h(n))$

Rules of thumb

examples

$f(n)$	$O(f(n))$
$7n - 2$	n
$3n^3 - 2n^2 + 5$	n^3
$3 \log n + 5$	$\log n$
$\log n + 2^n$	2^n
$10n^5 + 2^n$	2^n
$\log 2^n$	n
$2^n + 4^n$	4^n
100×2^n	2^n
$n2^n$	$n2^n$
$\log n!$	$n \log n$

Big-O: back to nearest points

```
def shortest_distance(points):
    n = len(points)                               # 1 (constant?)
    min = 0                                       # 1 (constant)
    for i in range(n):                            # n times
        for j in range(i):                        # i times
            d = distance(points[i], points[j])    # 1 (constant)
            if min > d:                           # 1 (constant)
                min =  d                          # 1 (constant)
    return min                                    # 1 (constant)
```

$$
\begin{aligned}
T(n) &= 2 + (1 + 2 + 3 + \ldots + n - 1) \times 3 + 1 \\
&= 2 \times \frac{(n-1)(n-2)}{3} + 3 = 2/3(n^2 - 3n + 2) + 3 \\
&= O(n^2)
\end{aligned}
$$

Big-O examples

linear search

```
1 def linear_search(seq, val):
2     i, n = 0, len(seq)
3     while i < n:
4         if seq[i] == val:
5             return i
6         i += 1
7     return None
```

- What is the worst-case running time?
  2. 2 assignments
  3. $2n$ comparisons, n increment
  7. 1 return statemnt

  $T(n) = 3n + 3 = O(n)$
- What is the average-case running time?
  2. 2 assignments
  3. $2(n/2)$ comparisons, $n/2$ increment, 1 return

  $T(n) = 3/2n + 3 = O(n)$
- What about best case? $O(1)$

Note: do not confuse the big-O with the worst case analysis.

Recursive example

Recursive binary search

```
def rbs(a, x, L=0, R=n):
    if L >= R:
        return None
    M = (L + R) // 2
    if a[M] == x:
        return M
    if a[M] > x:
        return rbs(a, x, L,
         ↪  M - 1)
    else:
        return rbs(a, x, M +
         ↪  1, R)
```

- Counting is not easy, but realize that $T(n) = c + T(n/2)$
- This is a recursive formula, it means $T(n/2) = c + T(n/4)$, $T(n/4) = c + T(n/8), \ldots$
- So, $T(n) = 2c + T(n/4) = 3c + T(n/8)$
- More generally, $T(n) = ic + T(n/2^i)$
- Recursion terminates when $n/2^i = 1$, or $n = 2^i$, the good news: $i = \log n$
- $T(n) = c \log n + T(1) = O(\log n)$

You do not always need to prove: for most recurrence relations, there is a way to obtain quick solutions (we are not going to cover it further, see Appendix)

Why asymptotic analysis is important?

'maximum problem size'

- Assume we can solve a problem of size m in a given time on current hardware
- We get a better computer, which runs 1024 times faster
- New problem size we can solve in the same time

Complexity	new problem size
Linear (n)	$1024m$
Quadratic (n^2)	$32m$
Exponential (2^n)	$m + 10$

- This also demonstrates the gap between polynomial and exponential algorithms:
  - with a exponential algorithm fast hardware does not help
  - problem size for exponential algorithms does not scale with faster computers

Worst case and asymptotic analysis

pros and cons

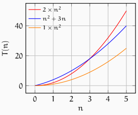
- We typically compare algorithms based on their worst-case performance
  - pro it is easier, and we get a (very) strong guarantee: we know that the algorithm won't perform worse than the bound
  - con a (very) strong guarantee: in some (many?) problems, worst case examples are rare
    - In practice you may prefer an algorithm that does better on average (we'll see examples from sorting)
- Our analyses are based on asymptotic behavior
  - pro for a 'large enough' input asymptotic analysis is correct
  - con constant or lower order factors are not always unimportant
    - A constant factor of 100^{100} should probably not be ignored

Big-O relatives

- Big-O (upper bound): $f(n)$ is $O(g(n))$
  if $f(n)$ is asymptotically *less than or equal to* $g(n)$
  $$f(n) \leqslant cg(n) \text{ for } n > n_0$$
- Big-Omega (lower bound): $f(n)$ is $\Omega(g(n))$
  if $f(n)$ is asymptotically *greater than or equal to* $g(n)$
  $$f(n) \geqslant cg(n) \text{ for } n > n_0$$
- Big-Theta (upper/lower bound): $f(n)$ is $\Theta(g(n))$
  if $f(n)$ is asymptotically *equal to* $g(n)$
  $$f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n))$$

Big-O, Big-Ω, Big-Θ: an example

$T(n) = n^2 + 3n$ is $\Theta(n^2)$



O for $c = 2$ and $n_0 = 3$
$$T(n) \leqslant cg(n) \text{ for } n > n_0$$
Ω for $c = 1$ and $n_0 = 0$
$$T(n) \geqslant cg(n) \text{ for } n > n_0$$
Θ for $c = 2$, $n_0 = 3$, $c' = 1$ and $n'_1 = 0$
$$T(n) \leqslant cg(n) \text{ for } n > n_0 \quad \textbf{and}$$
$$T(n) \geqslant c'g(n) \text{ for } n > n'_0$$

Summary

- Algorithmic analysis mainly focuses on worst-case asymptotic running times
- *Sublinear* (e.g., *logarithmic*), *Linear* and *n log n* algorithms are good
- *Polynomial* algorithms may be acceptable in many cases
- *Exponential* algorithms are bad
- We will return to concepts from this lecture while studying various algorithms
- Reading for this lectures: **goodrich2013**

Next:

- Common patterns in algorightms
- Sorting algorithms
- Reading: **goodrich2013** – up to 12.7

Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.

A(nother) view of computational complexity

P, NP, NP-complete and all that

- A major division of complexity classes according to Big-O notation is between
  - P polynomial time algorithms
  - NP non-deterministic polynomial time algorithms
- A big question in computing is whether P = NP
- All problems in NP can be reduced in polynomial time to a problem in a subclass of NP (*NP-complete*)
  - Solving an NP complete problem in P would mean proving

  P = NP

Video from `https://www.youtube.com/watch?v=YX40hbAHx3s`

Exercise

Sort the functions based on asymptotic order of growth

$\log n^{1000}$	$\log 5^n$
$n \log(n)$	$\binom{n}{n/2}$
5^n	$\log \log n!$
$\log n$	$\sqrt{n}$
$\log n^{1/\log n}$	n^2
$\log n$	2^n
$\log 2^n/n$	$\binom{n}{2}$
$\log n!$	
$\log 2^n$	

Recurrence relations

the master theorem

- Given a recurrence relation:
  $$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$
  - a number of sub-problems
  - b reduction factor or the input
  - n^d amount of work to create and combine sub-problems

  $$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a = b^d \end{cases}$$
- The theorem is more general than most cases where $a = b$
- But the theorem is not general for all recurrences: it requires equal splits

Big-O example with recurrence

an informal sketch of complexity of segmentation

```python
def segment_r(seq):
  if len(seq) == 1:
    yield [seq]
  else:
    for seg in segment_r(seq[1:]):
      yield [seq[0]] + seg
      yield [seq[0] + seg[0]] +
        ↪ seg[1:]
```

- Intuition:
  - if $n = 1$, time is constant: c
  - for $n = 2$ we make two recursive calls $2c$
  - for $n = 3$ we make two recursive calls with size 2 (ignoring size 1 calls) $2 \times 2c$
  - for $n = 4$ we make more calls, at least including $2 \times 2 \times 2c$
  - for $n = 5$ we make even more calls, at least including $2 \times 2 \times 2 \times 2c$
  - for n we make at least $2^{n-1}c$ calls

Note that the master theorem is not useful for this algorithm.