

CS 341: Algorithms

Lecture 1: Introduction, review of asymptotics

Éric Schost

based on lecture notes by many other CS341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2024

Staff

Instructors

- Trevor Brown
- Éric Schost (office hours Thursday, 2-3pm)

ISC

- Sylvie Davies

Electronic communication

Course webpage:

- Course outline
- Lecture slides

Piazza

- Make sure you are signed up using your uwaterloo email address
- <http://piazza.com/uwaterloo.ca/fall2024/cs341>
- posting solutions to assignments is forbidden

email

- use your uwaterloo address

Assignments, exams, etc

- **4 assignments** (10% each)
- **Midterm** (20%)
 - Monday October 28, 4:30-6:20pm
- **Final** (40%)
 - TBA

References

- **Slides**
 - posted before the lecture (usually)
- **Textbooks**
 - **Introduction to Algorithms**, Cormen, Leiserson, Rivest, Stein [CLRS]
 - **Algorithm Design**, Kleinberg, Tardos [KT]
 - **Algorithms**, Dasgupta, Papadimitriou, Vazirani [DPV]

This course

What you should know

- CS240-level data structures and algorithms
- big-O notation
- maybe a bit of math (matrices, for instance)

What we will do

- a lot of algorithms
- pseudo-code
- proofs for correctness and runtime

What we will not do

- read/write code in class

Tentative syllabus

- divide-and-conquer, master theorem
- greedy algorithms
- dynamic programming
- breadth-first and depth-first search
- shortest paths in graphs
- flows and cuts
- NP-completeness

Cost of algorithms

Inputs

- parameterized by an integer n , called the **size**
- e.g., length of an array that we want to work with

$T(I)$ = runtime on input I

runtime of a particular instance

$T(n) = \max_{I \text{ of size } n} T(I)$

worst-case runtime

$T_{\text{avg}}(n) = \frac{\sum_{I \text{ of size } n} T(I)}{\text{number of inputs of size } I}$

average runtime, not used much in this course

Remark: we will sometimes use more than one parameter

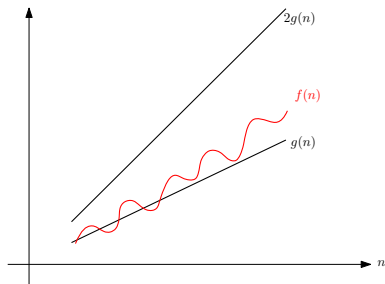
- numbers of rows and columns in a matrix
- vertices and edges in a graph

Asymptotic notation

Consider two functions $f(n), g(n)$ with values in $\mathbb{R}_{>0}$

big-O.

1. we say that $f(n) \in O(g(n))$ if
there exist $C > 0$ and n_0 , such that for $n \geq n_0$, $f(n) \leq Cg(n)$

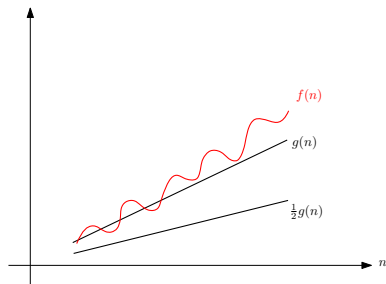


Asymptotic notation

Consider two functions $f(n), g(n)$ with values in $\mathbb{R}_{>0}$

big- Ω .

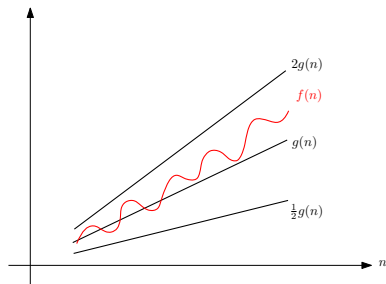
1. we say that $f(n) \in \Omega(g(n))$ if
there exist $C > 0$ and n_0 such that for $n \geq n_0$, $f(n) \geq Cg(n)$
2. equivalent to $g(n) \in O(f(n))$



Asymptotic notation

Consider two functions $f(n), g(n)$ with values in $\mathbb{R}_{>0}$

- Θ .
1. we say that $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.
 2. in particular true if $\lim_{\infty} f(n)/g(n) = C$ for some $0 < C < \infty$

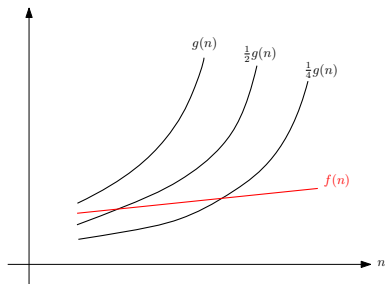


Asymptotic notation

Consider two functions $f(n), g(n)$ with values in $\mathbb{R}_{>0}$

little-o.

1. we say that $f(n) \in o(g(n))$ if
for all $C > 0$, there exists n_0 such that for $n \geq n_0$, $f(n) \leq Cg(n)$
2. equivalent to $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$.



Examples

- $n^k + c_{k-1}n^{k-1} + \dots + c_0$ is in $\Theta(n^k)$
- $n^{O(1)}$ means **(at most) polynomial in n**

c_i and k constant!

True/False

2^{n-1} is in $\Theta(2^n)$

True/False

$(n-1)!$ is in $\Theta(n!)$

Definitions for several parameters

The screenshot shows a web browser window displaying a Stack Exchange question. The browser's address bar shows the URL: `cs.stackexchange.com/questions/3149/what-is-the-meaning-of-omv/3151#3151`. The page title is "terminology - What is the meaning of $SO(m+n)$? - Computer Science Stack Exchange". The Stack Exchange logo and a search bar are visible at the top. A left sidebar contains navigation links: Home, PUBLIC, Questions (selected), Tags, Users, Companies, Unanswered, and TEAMS. The main content area shows a question with 32 answers. The selected answer, by Kristoffer Arnfelt Hansen, is highlighted. The answer text reads: "Believe it or not, it seems (in my experience) that many algorithms people have actually not thought about what the big O notation formally means, and when asked about it, you can get several different answers. Some issues are discussed in the paper [On Asymptotic Notation with Multiple Variables](#) by Rodney R. Howell. Curiously, it also seems that most introductory algorithms courses spend lots of time being very formal about big O notation with a single variable, and then the next weeks happily use the notation for graph algorithms with several variables in a casual way, without discussing what the notation actually means." Below the answer text are options to "Share", "Cite", "Improve this answer", and "Follow". The answer was posted on "Aug 13, 2012 at 19:49" and has 561 votes, 3 comments, and 7 flags.

terminology - What is the meaning of $SO(m+n)$? - Computer Science Stack Exchange - Google Chrome

terminology - W | x +

cs.stackexchange.com/questions/3149/what-is-the-meaning-of-omv/3151#3151

StackExchange Search on Computer Science... Log in Sign up

Home

PUBLIC

Questions

Tags

Users

Companies

Unanswered

TEAMS

Stack Overflow for Teams – Start collaborating and sharing organizational

32

Believe it or not, it seems (in my experience) that many algorithms people have actually not thought about what the big O notation formally means, and when asked about it, you can get several different answers. Some issues are discussed in the paper [On Asymptotic Notation with Multiple Variables](#) by Rodney R. Howell.

Curiously, it also seems that most introductory algorithms courses spend lots of time being very formal about big O notation with a single variable, and then the next weeks happily use the notation for graph algorithms with several variables in a casual way, without discussing what the notation actually means.

Share Cite Improve this answer Follow

answered Aug 13, 2012 at 19:49

Kristoffer Arnfelt Hansen

561 3 7

Definitions for several parameters

Consider two functions $f(n, m), g(n, m)$ with values in $\mathbb{R}_{>0}$

Definition

$f(n, m)$ is in $O(g(n, m))$ if there exist C, n_0, m_0 such that
 $f(n, m) \leq Cg(n, m)$ for $n \geq n_0$ **or** $m \geq m_0$ (i.e. finitely many exceptions)

Remark:

- weaker definition: there exist C, n_0, m_0 such that $f(n, m) \leq Cg(n, m)$ for $n \geq n_0$ **and** $m \geq m_0$
- will not matter too much which one we choose

Computational model: word RAM

Rough definition:

- memory locations contain **integer words** of **b bits** each
- assume $b \geq \log(n)$ for input size n
(an integer M uses $\log |M|/b$ words, integers in $n^{O(1)}$ fit in $O(1)$ words)
- Random Access Memory: can **access any memory location** at unit cost, **basic operations on words** have unit costs

Computational model: word RAM

Rough definition:

- memory locations contain **integer words** of **b bits** each
- assume $b \geq \log(n)$ for input size n
(an integer M uses $\log |M|/b$ words, integers in $n^{O(1)}$ fit in $O(1)$ words)
- Random Access Memory: can **access any memory location** at unit cost, **basic operations on words** have unit costs

Sum($A[1..n]$)

1. $s \leftarrow 0$
2. **for** $i = 1, \dots, n$
3. $s \leftarrow s + A[i]$

Exercise

If all entries of A fit in a word, what is the cost?

Computational model: word RAM

Product($A[1..n]$)

1. $s \leftarrow 1$
2. **for** $i = 1, \dots, n$
3. $s \leftarrow s \times A[i]$

Exercise

All entries of A fit in a word. Does this have the same runtime as the Sum algorithm (previous slide)?

Computational model: word RAM

Product($A[1..n]$)

1. $s \leftarrow 1$
2. **for** $i = 1, \dots, n$
3. $s \leftarrow s \times A[i]$

Exercise

All entries of A fit in a word. Does this have the same runtime as the Sum algorithm (previous slide)?

More examples

- matrix multiplication algorithms (with word-size inputs) are OK
- other matrix algorithms (Gaussian elimination) need more care
- (weighted) graph algorithms (weights fit in a word) are usually OK

Case study: maximum subarray

Question

Given an array $A[1..n]$, find a contiguous subarray $A[i..j]$ that maximizes the sum $A[i] + \dots + A[j]$. All entries fit in a word.

Example. Given

$$A = [10, -5, 4, 3, -5, 6, -1, -1]$$

the subarray

$$A[1..6] = [10, -5, 4, 3, -5, 6]$$

has sum $10 + \dots + 6 = 13$. It is the best we can do.

Convention. We can take $j < i$, so $A[i..j]$ is empty, and the sum is zero.

Brute force algorithm

BruteForce(A)

1. $\text{opt} \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n **do**
3. **for** $j \leftarrow i$ **to** n **do**
4. $\text{sum} \leftarrow 0$
5. **for** $k \leftarrow i$ **to** j **do**
6. $\text{sum} \leftarrow \text{sum} + A[k]$
7. **if** $\text{sum} > \text{opt}$
8. $\text{opt} \leftarrow \text{sum}$
9. **return** opt

Brute force algorithm

BruteForce(A)

```
1.  opt ← 0
2.  for  $i \leftarrow 1$  to  $n$  do
3.      for  $j \leftarrow i$  to  $n$  do
4.          sum ← 0
5.          for  $k \leftarrow i$  to  $j$  do
6.              sum ← sum +  $A[k]$ 
7.          if sum > opt
8.              opt ← sum
9.  return opt
```

Runtime: $\Theta(n^3)$

Improved brute force algorithm

Idea: we recompute the same sum many times in the j loop.

Improved brute force algorithm

Idea: we recompute the same sum many times in the j loop.

BetterBruteForce(A)

```
1.   opt ← 0
2.   for  $i \leftarrow 1$  to  $n$  do
3.       sum ← 0
4.       for  $j \leftarrow i$  to  $n$  do
5.           sum ← sum +  $A[j]$ 
6.           if sum > opt
7.               opt ← sum
8.   return opt
```

Runtime: $\Theta(n^2)$

Divide-and-conquer

Idea: solve the problem twice in size $n/2$ (we assume n is a power of 2). Then the optimal subarray (if not empty)

1. is completely in the left half $A[1..n/2]$
2. or is completely in the right half $A[n/2 + 1..n]$
3. or contains **both** $A[n/2]$ **and** $A[n/2 + 1]$

(cases mutually exclusive.)

Divide-and-conquer

Idea: solve the problem twice in size $n/2$ (we assume n is a power of 2). Then the optimal subarray (if not empty)

1. is completely in the left half $A[1..n/2]$
2. or is completely in the right half $A[n/2 + 1..n]$
3. or contains **both** $A[n/2]$ **and** $A[n/2 + 1]$

(cases mutually exclusive.)

To find the optimal subarray in case **3**, write

$$A[i] + \cdots + A[j] = A[i] + \cdots + A[n/2] + A[n/2 + 1] + \cdots + A[j]$$

Divide-and-conquer

Idea: solve the problem twice in size $n/2$ (we assume n is a power of 2). Then the optimal subarray (if not empty)

1. is completely in the left half $A[1..n/2]$
2. or is completely in the right half $A[n/2 + 1..n]$
3. or contains **both** $A[n/2]$ **and** $A[n/2 + 1]$

(cases mutually exclusive.)

To find the optimal subarray in case **3**, write

$$A[i] + \cdots + A[j] = A[i] + \cdots + A[n/2] + A[n/2 + 1] + \cdots + A[j]$$

more abstractly: $F(i, j) = f(i) + g(j)$, for i in $1, \dots, n/2$ and j in $n/2 + 1, \dots, n$

To maximize $F(i, j)$, maximize $f(i)$ and $g(j)$ **independently**.

Divide-and-conquer

Idea: solve the problem twice in size $n/2$ (we assume n is a power of 2). Then the optimal subarray (if not empty)

1. is completely in the left half $A[1..n/2]$
2. or is completely in the right half $A[n/2 + 1..n]$
3. or contains **both** $A[n/2]$ **and** $A[n/2 + 1]$

(cases mutually exclusive.)

To find the optimal subarray in case **3**, write

$$A[i] + \cdots + A[j] = A[i] + \cdots + A[n/2] + A[n/2 + 1] + \cdots + A[j]$$

more abstractly: $F(i, j) = f(i) + g(j)$, for i in $1, \dots, n/2$ and j in $n/2 + 1, \dots, n$

To maximize $F(i, j)$, maximize $f(i)$ and $g(j)$ **independently**.

Maximizing half-sums

MaximizeLowerHalf(A)

1. $\text{opt} \leftarrow A[n/2]$
2. $\text{sum} \leftarrow A[n/2]$
3. **for** $i = n/2 - 1, \dots, 1$ **do**
4. $\text{sum} \leftarrow \text{sum} + A[i]$
5. **if** $\text{sum} > \text{opt}$
6. $\text{opt} \leftarrow \text{sum}$
7. **return** opt

Runtime: $\Theta(n)$

Maximizing half-sums

MaximizeLowerHalf(A)

1. $\text{opt} \leftarrow A[n/2]$
2. $\text{sum} \leftarrow A[n/2]$
3. **for** $i = n/2 - 1, \dots, 1$ **do**
4. $\text{sum} \leftarrow \text{sum} + A[i]$
5. **if** $\text{sum} > \text{opt}$
6. $\text{opt} \leftarrow \text{sum}$
7. **return** opt

Runtime: $\Theta(n)$

MaximizeUpperHalf(A)

1. ...

Runtime: $\Theta(n)$

Main algorithm

DivideAndConquer($A[1..n]$)

1. **if** $n = 1$ **return** $\max(A[1], 0)$
2. $\text{opt}_{\text{lo}} \leftarrow \text{DivideAndConquer}(A[1..n/2])$
3. $\text{opt}_{\text{hi}} \leftarrow \text{DivideAndConquer}(A[n/2 + 1..n])$
4. $\text{opt}_{\text{middle}} \leftarrow \text{MaximizeLowerHalf}(A) + \text{MaximizeUpperHalf}(A)$
5. **return** $\max(\text{opt}_{\text{lo}}, \text{opt}_{\text{hi}}, \text{opt}_{\text{middle}})$

Main algorithm

DivideAndConquer($A[1..n]$)

1. **if** $n = 1$ **return** $\max(A[1], 0)$
2. $\text{opt}_{\text{lo}} \leftarrow \text{DivideAndConquer}(A[1..n/2])$
3. $\text{opt}_{\text{hi}} \leftarrow \text{DivideAndConquer}(A[n/2 + 1..n])$
4. $\text{opt}_{\text{middle}} \leftarrow \text{MaximizeLowerHalf}(A) + \text{MaximizeUpperHalf}(A)$
5. **return** $\max(\text{opt}_{\text{lo}}, \text{opt}_{\text{hi}}, \text{opt}_{\text{middle}})$

Runtime: $T(n) = 2T(n/2) + \Theta(n)$ so $T(n) \in \Theta(n \log(n))$

Proof: same as MergeSort. Details in next module.

Dynamic programming (time permitting)

Idea: solve the problem in subarrays $A[1..j]$ of sizes $1, \dots, n$. The optimal subarray

1. is either a subarray of $A[1..n - 1]$,
2. or contains $A[n]$

(cases mutually exclusive!)

Dynamic programming (time permitting)

Idea: solve the problem in subarrays $A[1..j]$ of sizes $1, \dots, n$. The optimal subarray

1. is either a subarray of $A[1..n - 1]$,
2. or contains $A[n]$

(cases mutually exclusive!)

Translation: write $M(j) = \max$ sum for subarrays of $A[1..j]$. Then

$$M(n) = \max(M(n - 1), \overline{M}(n))$$

with $\overline{M}(j) = \max$ sum for subarrays of $A[1..j]$, that include j .

Dynamic programming (time permitting)

How can we compute $\overline{M}(1), \dots, \overline{M}(n)$?

Idea. As before: the optimal subarray that contains $A[n]$

1. is of the form $A[i..n-1, n]$, for some $i \leq n-1$

2. or is exactly $[A[n]]$

(cases mutually exclusive)

Dynamic programming (time permitting)

How can we compute $\overline{M}(1), \dots, \overline{M}(n)$?

Idea. As before: the optimal subarray that contains $A[n]$

1. is of the form $A[i..n-1, n]$, for some $i \leq n-1$
2. or is exactly $[A[n]]$

(cases mutually exclusive)

Translation: $\overline{M}(n) = \max(\overline{M}(n-1) + A[n], A[n]) = A[n] + \max(\overline{M}(n-1), 0)$

Can eliminate recursive calls, and write as a loop.

- | |
|--|
| <ol style="list-style-type: none">1. $\overline{M} \leftarrow A[1]$2. for $i = 2, \dots, n$ do3. $\overline{M} \leftarrow A[i] + \max(\overline{M}, 0)$ |
|--|

Main algorithm (time permitting)

DynamicProgramming(A)

1. $\bar{M} \leftarrow A[1]$
2. $M \leftarrow \max(\bar{M}, 0)$
3. **for** $i = 2, \dots, n$ **do**
4. $\bar{M} \leftarrow A[i] + \max(\bar{M}, 0)$
5. $M \leftarrow \max(M, \bar{M})$
6. **return** M

Runtime: $\Theta(n)$