CS 341: Algorithms
Lecture 1: Course Introduction

Armin Jamshidpey Collin Roberts

Based on lecture notes by Éric Schost and many previous CS 341 instructors

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

Winter 2025

Staff

Instructors

- Armin Jampshidpey
- Mark Petrick
- Collin Roberts

ISC

- Sylvie Davies (sldavies)

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/winter2025/cs341`
- posting solutions to assignments is forbidden

email

- use your uwaterloo address

Assignments, exams, project, etc

- **5 assignments** (20%)
- **2 programming questions** (4%)
- **Midterm** (30%)
  - Monday, Feb 24, 7:00-8:50pm.
- **Final** (46%)
  - TBA

In order to pass the course, you **must**:

- earn half of the written assignment points and
- earn half of the exam points.

If you don't meet the above requirement, your final mark will be the smaller of the normal calculation and a mark of 46.

References

- **Slides**
  - posted before the lecture (expectedly)
- **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 strucures 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
- breadth-first and depth-first search
- greedy algorithms
- dynamic programming
- 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

$T(I) = \text{runtime on input } I$	**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

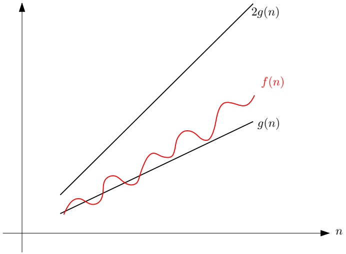
- numbers of rows and columns in a matrix
- vertices and edges in a graph

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 $n \geq n_0$, $f(n) \leq Cg(n)$

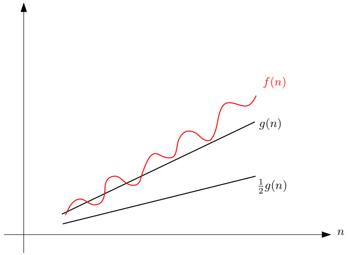


Asymptotic notation

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 $g(n) \in O(f(n))$

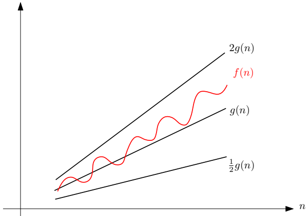


Asymptotic notation

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 there exist $C, C' > 0$ and n_0 such that for $n \geq n_0$, $C'g(n) \leq f(n) \leq Cg(n)$
2. equivalent to $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.
3. in particular true if $\lim_\infty f(n)/g(n) = C$ for some $0 < C < \infty$

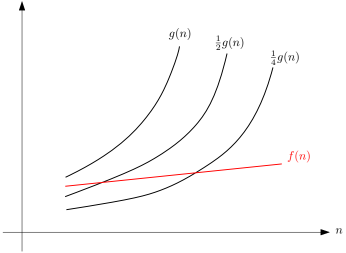


Asymptotic notation

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 $\lim_{n\to\infty} f(n)/g(n) = 0$.

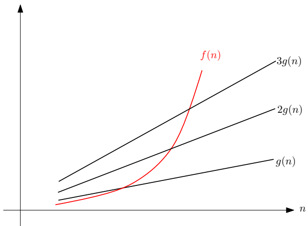


Asymptotic notation

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

little-ω.

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



Examples

- $n^k + c_{k-1}n^{k-1} + \cdots + c_0$ is in $\Theta(n^k)$ c_i and k constant!
- $n^{O(1)}$ means **(at most) polynomial in n**
- $n \log(n)$ is in $O(n^2)$ and $\Omega(n)$

True/False

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

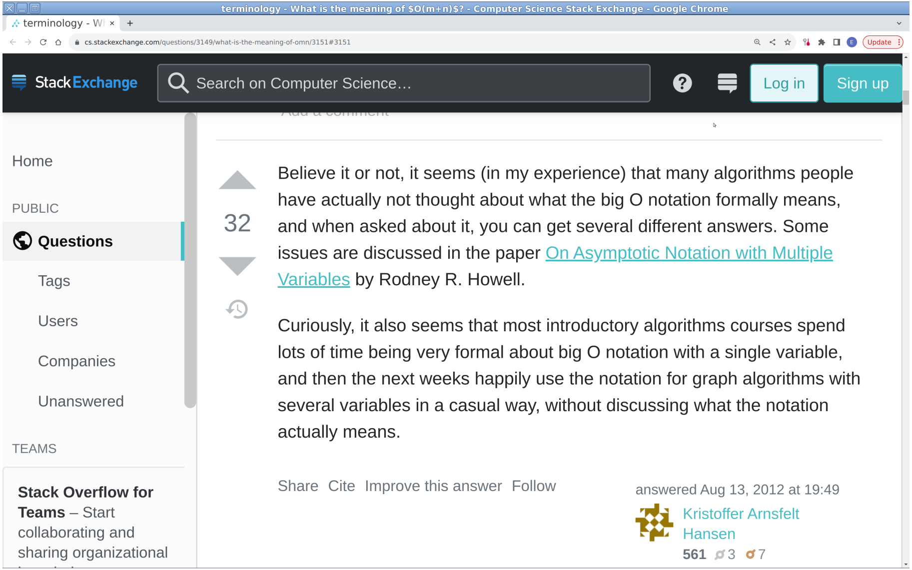
https://padlet.com/arminjamshidpey/CS341

True/False

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

https://padlet.com/arminjamshidpey/CS341

Definitions for several parameters



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

cs.stackexchange.com/questions/3149/what-is-the-meaning-of-omn/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 Arnsfelt Hansen

561 3 7

Definitions for several parameters

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

$f(n,m)$ 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** $m \geq m_0$

Remark:

- less strict 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 $\boldsymbol{b}$ **bits** each
- assume $\boldsymbol{b} \geq \mathbf{log}(\boldsymbol{n})$ for input size n
- 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
- 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** $i = 1, \ldots, n$
3. $s \leftarrow s + A[i]$

Padlet

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

https://padlet.com/arminjamshidpey/CS341

Computational model: word RAM

Product$(A[1..n])$
1. $s \leftarrow 1$
2. **for** $i = 1, \ldots, n$
3. $\quad s \leftarrow s \times A[i]$

Padlet

All entries of A fit in a word. Does it have the same runtime as the Sum algorithm (on the previous slide)?
https://padlet.com/arminjamshidpey/CS341

Computational model: word RAM

Product$(A[1..n])$
1. $s \leftarrow 1$
2. **for** $i = 1, \ldots, n$
3. $\quad s \leftarrow s \times A[i]$

Padlet

All entries of A fit in a word. Does it have the same runtime as the Sum algorithm (on the previous slide)?
https://padlet.com/arminjamshidpey/CS341

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

Practical relevance?

1. **big-O is only an upper bound**
   - typical example: 1 is in $O(n^2)$ and n is in $O(n)$, but ...
   - try to give Θ's if possible
2. **big-anything hides constants**
   - this is by design
   - a $\Theta(n^2)$ will beat a $\Theta(n^3)$ algorithm **eventually**
   - **galactic algorithms**: become practically relevant for astronomical input sizes (fast matrix or integer multiplication)
3. **we use a simplified model**
   - artificial computational model
   - focus on "operations", forget memory requirements, data locality, ...

Case study: maximum subarray

Task

Given an array $A[1..n]$, find a contiguous subarray $A[i..j]$ that maximizes the sum $A[i] + \cdots + A[j]$.

Example. Given

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

the subarray

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

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

Convention. We can take $j = i - 1$, so $A[i..j]$ is empty, and the sum is zero.

Note: To make things simpler, here we just try to find the maximum sum and not the array.

Brute force algorithm

Test2(A)

```
max ← 0
for i ← 1 to n do
    for j ← i to n do
        sum ← 0
        for k ← i to j do
            sum ← A[k]
return max
```

Brute force algorithm

```
BruteForce(A)
    opt ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← sum + A[k]
            if sum > opt
                opt ← sum
    return opt
```

Brute force algorithm

```
BruteForce(A)
    opt ← 0
    for i ← 1 to n do
        for j ← i to n do
            sum ← 0
            for k ← i to j do
                sum ← sum + A[k]
            if sum > opt
                opt ← sum
    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)
    opt ← 0
    for i ← 1 to n do
        sum ← 0
        for j ← i to n do
            sum ← sum + A[j]
            if sum > opt
                opt ← sum
    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).

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 $1, \ldots, n/2$ and j in $n/2+1, \ldots, n$

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 $1, \ldots, n/2$ and j in $n/2+1, \ldots, n$

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

Maximizing half-sums

MaximizeLowerHalf(A)

```
opt ← A[n/2]
sum ← A[n/2]
for i = n/2 − 1, ..., 1 do
    sum ← sum + A[i]
    if sum > opt
        opt ← sum
return opt
```

Runtime: $\Theta(n)$

Maximizing half-sums

MaximizeLowerHalf(A)

```
opt ← A[n/2]
sum ← A[n/2]
for i = n/2 − 1, . . . , 1 do
    sum ← sum + A[i]
    if sum > opt
        opt ← sum
return opt
```

Runtime: $\Theta(n)$

MaximizeUpperHalf(A)

```
. . .
```

Runtime: $\Theta(n)$

Main algorithm

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

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

Idea: solve the problem in subarrays $A[1..j]$ of sizes $1, \ldots, n$.

Dynamic programming

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

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

(cases mutually exclusive!)

Dynamic programming

Idea: solve the problem in subarrays $A[1..j]$ of sizes $1, \ldots, n$. 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

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

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

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

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.

$\overline{M} \leftarrow A[1]$
for $i = 2, \ldots, n$ **do**
    $\overline{M} \leftarrow A[i] + \max(\overline{M}, 0)$

Main algorithm

DynamicProgramming(A)

$\overline{M} \leftarrow A[1]$
$M \leftarrow \max(\overline{M}, 0)$
for $i = 2, \ldots, n$ **do**
  $\overline{M} \leftarrow A[i] + \max(\overline{M}, 0)$
  $M \leftarrow \max(M, \overline{M})$
return M

Runtime: $\Theta(n)$