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 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
- 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

$T(I) = \text{runtime on input } I$	**runtime of a particular instance**
$T(n) = \max_{I \text{ of size } n} T(I)$	**worst-case runtime**
$T_{\text{avg}}(n) = \dfrac{\sum_{I \text{ of size } n} T(I)}{\text{number of inputs of size } I}$	**average runtime, not used much in this course**

Remark: we will sometimes use more than one parameter

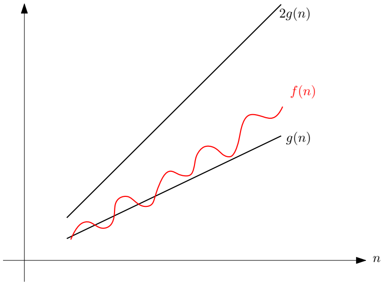
- numbers of rows and columns in a matrix
- vertices and edges in a graph

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

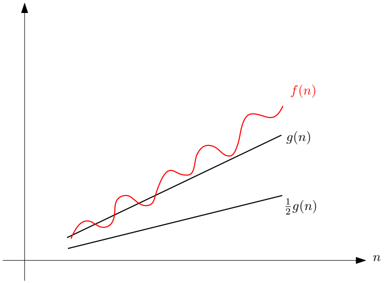


Asymptotic notation

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

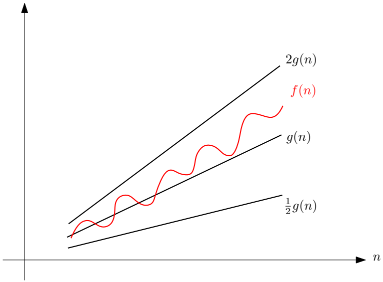


Asymptotic notation

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 $0 < C < \infty$

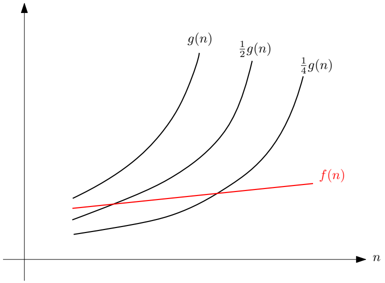


Asymptotic notation

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



Examples

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

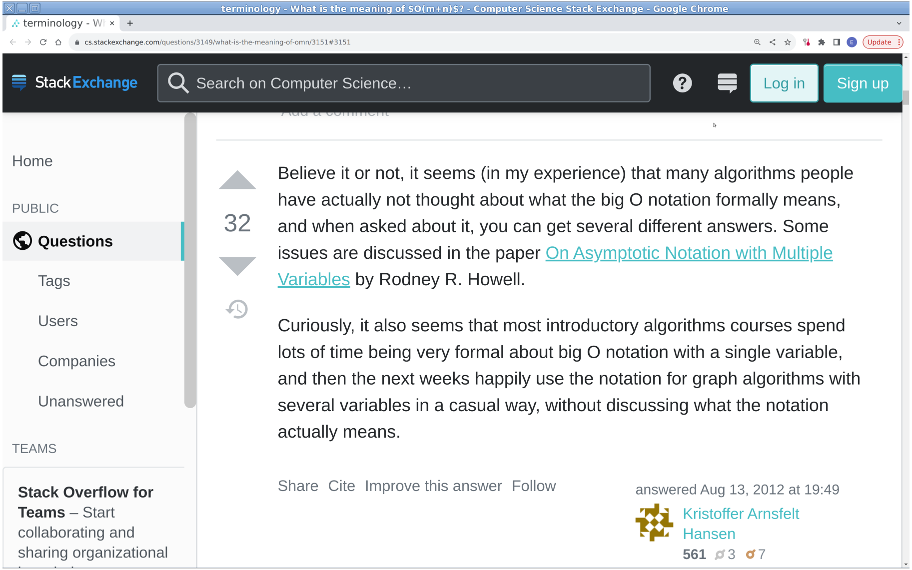
True/False

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

True/False

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

Definitions for several parameters



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 $\boldsymbol{b}$ **bits** each
- assume $\boldsymbol{b \geq \log(n)}$ 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 $\boldsymbol{b}$ **bits** each
- assume $\boldsymbol{b \geq \log(n)}$ 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 ← 0
2.    for i = 1, ..., n
3.        s ← s + A[i]
```

Exercise

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

Computational model: word RAM

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

$s \leftarrow 1$

for $i = 1, \ldots, n$

  $s \leftarrow s \times A[i]$

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** $i = 1, \ldots, n$
3. $\quad s \leftarrow s \times A[i]$

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 $A[i] + \cdots + A[j]$. 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 $10 + \cdots + 6 = 13$. 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)
    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: $\boldsymbol{\Theta(n^3)}$

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: $\boldsymbol{\Theta(n^2)}$

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: $\boldsymbol{F(i,j) = f(i) + g(j)}$, 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: $\boldsymbol{F(i,j) = f(i) + g(j)}$, 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: $\boldsymbol{\Theta(n)}$

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: $\boldsymbol{\Theta(n)}$

MaximizeUpperHalf(A)

```
...
```

Runtime: $\boldsymbol{\Theta(n)}$

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 $\boldsymbol{T(n) \in \Theta(n \log(n))}$

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 $1, \ldots, n$. 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 (time permitting)

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 (time permitting)

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.

```
1.    M̄ ← A[1]
2.    for i = 2, ..., n do
3.        M̄ ← A[i] + max(M̄, 0)
```

Main algorithm (time permitting)

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: $\boldsymbol{\Theta(n)}$