# CS 341: Algorithms

**University of Waterloo**
**Éric Schost**
**eschost@uwaterloo.ca**

**Lecture 1: welcome**

# Staff

**Instructors**

- Armin Jamshidpey
- Lap Chi Lau
- Éric Schost

**ISC**

- Sylvie Davies (sldavies)

**Office hours** (ÉS, DC 3627)

- Tuesday, 12:30 - 1:30
- Thursday 1 - 2pm

# Electronic communication

### Piazza

- sign up using your uwaterloo email address
- http://piazza.com/uwaterloo.ca/winter2023/cs341
- posting solutions to assignments is forbidden

### email

- use your uwaterloo address

# Assignments, exams, project, etc

- **5 assignments** (40%)
  - include programming problems
  - due on Friday (Jan27, Feb10, Mar10, Mar24, Apr07)
- **Midterm** (20%)
  - Friday February 17, 04:30-06:00pm
- **Final** (40%)
  - TBA

# References

- **Slides and notes**
  - posted before the lecture (expectedly) on learn
- **Textbooks**
  - **Introduction to Algorithms**, Cormen, Leiserson, Rivest, Stein
  - **Algorithm Design**, Kleinberg, Tardos
  - **Algorithms**, Dasgupta, Papadimitriou, Vazirani

# 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
- 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

# 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

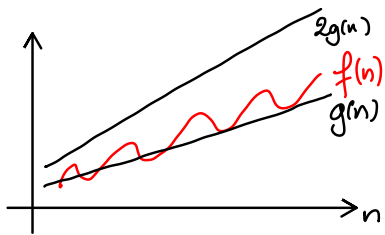
- numbers of rows and columns in a matrix
- vertices and edges in a graph

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

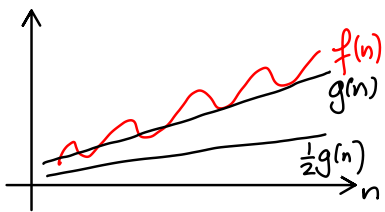


# Asymptotic notation

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

**big-$\Omega$.**

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

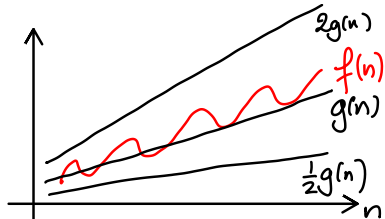


# Asymptotic notation

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

$\Theta$.

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 $0 < C < \infty$

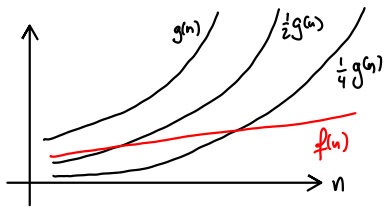


# Asymptotic notation

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

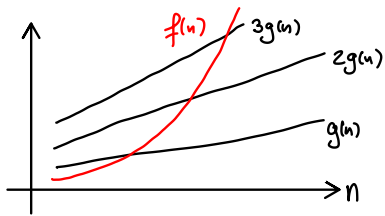


# Asymptotic notation

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

**little-$\omega$.**

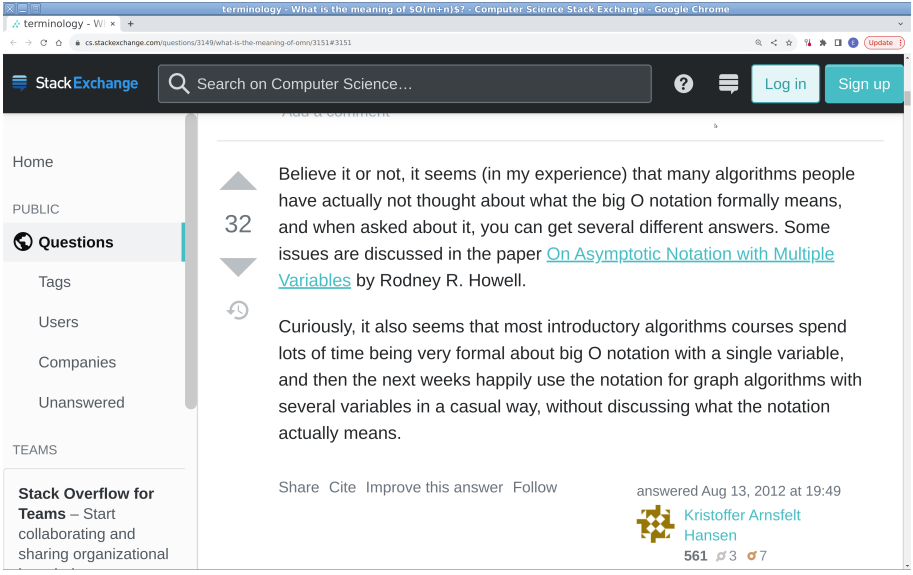
1. we say that $f(n) \in \omega(g(n))$ if
   for all $C > 0$, there exists $n_0$ such that for $n \geq n_0$, $f(n) > Cg(n)$
2. equivalent to $\lim_{n\to\infty} f(n)/g(n) = \infty$
3. equivalent to $g(n) \in o(f(n))$.



# 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 $\Omega(n)$
- $2^{n-1}$ is in $\Theta(2^n)$
- $(n-1)!$ is **not** in $\Theta(n!)$, it is in $o(n!)$

# Definitions for several parameters



# Definitions for several parameters

Consider two functions $f(n), g(n)$ 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 $\boldsymbol{b \geq \log(n)}$ for input size $n$
- Random Access Memory: can access any memory location at unit cost
- basic operations on words have unit costs

No need to worry about loop counters, array indices, ..., as long as they are $n^{O(1)}$.

# 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 $n$
- Random Access Memory: can access any memory location at unit cost
- basic operations on words have unit costs

No need to worry about loop counters, array indices, ..., as long as they are $n^{O(1)}$.

```
Sum(A[1..n])
1.    s ← 0
2.    for i = 1, ..., n
3.        s ← s + A[i]
```

If all entries of $A$ fit in a word, time $\boldsymbol{O(n)}$

# 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 $n$
- Random Access Memory: can access any memory location at unit cost
- basic operations on words have unit costs

No need to worry about loop counters, array indices, ..., as long as they are $n^{O(1)}$.

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

All entries of $A$ fit in a word. Runtime?

# 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 $n$
- Random Access Memory: can access any memory location at unit cost
- basic operations on words have unit costs

No need to worry about loop counters, array indices, ..., as long as they are $n^{O(1)}$.

**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 $\Theta$'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 also allow $A[i..j]$ empty, with sum equal to zero.

# Brute force algorithm

```
Test2(A)
1.    max ← 0
2.    for i ← 1 to n do
3.        for j ← i to n do
4.            sum ← 0
5.            for k ← i to j do
6.                sum ← A[k]
7.    return max
```

# Brute force algorithm

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

# Brute force algorithm

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

**Runtime**: $\Theta(n^3)$

**Remark**: could also return the indices $i, j$ giving the max

# Improved brute force algorithm

**Observation:** we recompute the same sum many times in the $j$ loop.

# Improved brute force algorithm

**Observation:** we recompute the same sum many times in the $j$ loop.

```
BetterBruteForce(A)
1.    opt ← 0
2.    for i ← 1 to n do
3.        sum ← 0
4.        for j ← 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).

# 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.

**Exercise:** can we use $A[i] + \cdots + A[j] = A[i] + \cdots + A[n] - A[n] - \cdots - A[j+1]$?

# Maximizing half-sums

```
MaximizeLowerHalf(A)
1.    opt ← A[n/2]
2.    sum ← A[n/2]
3.    for i = n/2 − 1, . . . , 1 do
4.        sum ← sum + A[i]
5.        if sum > opt
6.            opt ← sum
7.    return opt
```

**Runtime**: $\Theta(n)$

# Maximizing half-sums

```
MaximizeLowerHalf(A)
1.    opt ← A[n/2]
2.    sum ← A[n/2]
3.    for i = n/2 − 1, . . . , 1 do
4.        sum ← sum + A[i]
5.        if sum > opt
6.            opt ← 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. $\text{opt}_{\text{lo}} \leftarrow$ 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 nested subarrays $A[1..j]$ of sizes $1, \ldots, n$.

# Dynamic programming

**Idea:** solve the problem in nested 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 nested 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

$$M(j) = \max(M(j-1), \overline{M}(j)) \qquad j \geq 2$$

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:** $\overline{M}(j) = \max(\overline{M}(j-1) + A[j], A[j]) = A[j] + \max(\overline{M}(j-1), 0)$ for $j \geq 2$

# Dynamic programming

How to implement this?

- **direct translation:** two recursive functions. Many repeated operations.
  **Exercise:** $\Theta(n^2)$ for $M(n)$

# Dynamic programming

How to implement this?

- **direct translation:** two recursive functions. Many repeated operations.
  **Exercise:** $\Theta(n^2)$ for $M(n)$
- **better:** store all values $M(j)$ and $\overline{M}(j)$: $\Theta(n)$

# Dynamic programming

How to implement this?

- **direct translation:** two recursive functions. Many repeated operations.
  **Exercise:** $\Theta(n^2)$ for $M(n)$
- **better:** store all values $M(j)$ and $\overline{M}(j)$: $\Theta(n)$
- **best:** $\Theta(n)$ with no extra storage, using a loop.

```
DynamicProgramming(A[1..n])
1.    M̄ ← A[1]
2.    M ← max(M̄, 0)
3.    for i = 2, ..., n do
4.        M̄ ← A[i] + max(M̄, 0)
5.        M ← max(M, M̄)
6.    return M
```