Lecture 1: Computational Models, Time Complexity & An Example

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 7, 2023

Overview

- Computational Models
- Time Complexity & Efficiency
- Examples: 2SUM & 3SUM
- Acknowledgements

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write

Total time $\leftrightarrow$ # elementary operations

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write

Total time $\leftrightarrow$ # elementary operations

Finer distinction: word RAM and *unit cost* models.

- unit cost model $\leftrightarrow$ one assumes that words have *unbounded* size
- word RAM $\leftrightarrow$ words have a *pre-specified* size

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write
- *Assumptions*
  1. Alphabet fits into one word
  2. Input fits in memory
  3. No huge numbers in middle of computation

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write
- *Assumptions*
  1. Alphabet fits into one word
  2. Input fits in memory
  3. No huge numbers in middle of computation
- *Example*
  1. Input: graph with n vertices
  2. vertex labeled from set $\{1, \cdots, n\}$, edge with pair from $\{1, \cdots, n\}^2$

     $2 \log n$ bits to store vertex or edge (assume word size $O(\log n)$)
  3. basic operations (vertex comparison, accessing vertex/edge, etc.) constant time

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write
- **Bit Complexity (with word RAM):**
  1. when working with numerical algorithms, numbers may grow and no longer fit in one word - so need to account for that
  2. In this case, assume word is a bit (i.e. in $\{0, 1\}$)

$$\text{cost of operation} \leftrightarrow \# \text{ bit-operations}$$

Computational Models

Main Idea: a computational model should take into account all constraints of your machine, and account for the *scarcest resource(s)*.

- **Word RAM:**
  1. Memory modeled as array (access any position "unit time")
  2. Each entry of the array is a *word* with pre-specified size.
  3. Each word operation takes "unit time"
     - addition, multiplication, subtraction, division
     - read/write
- **Bit Complexity (with word RAM):**
  1. when working with numerical algorithms, numbers may grow and no longer fit in one word - so need to account for that
- Other models exist based on different resource constraints and assumptions (CS 365, CS 466 onwards)
  - Turing Machines
  - Circuits
  - Parallel computation
  - Online, streaming
  - many more

- Computational Models
- Time Complexity & Efficiency
- Examples: 2SUM & 3SUM
- Acknowledgements

Asymptotics recap

Given two functions $f, g : \mathbb{N} \to \mathbb{N}$

- $f(n) = O(g(n))$ if there is a constant C s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C$$

Examples:

- $\pi \cdot n^3 = O(n^3)$
- $10^{10} \cdot n^2 \log n = O(n^3)$
- $10n^3 + 100n^2 + n = O(n^3)$

Asymptotics recap

Given two functions $f, g : \mathbb{N} \to \mathbb{N}$

- $f(n) = O(g(n))$ if there is a constant C s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C$$

- $f(n) = \Omega(g(n))$ if there is a constant c s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$

Examples:

- $\pi \cdot n^3 = \Omega(n^3)$
- $10^{10} \cdot n^3 = \Omega(n^2 \log n)$
- $10n^3 + 100n^2 + n = \Omega(n^3)$

Asymptotics recap

Given two functions $f, g : \mathbb{N} \to \mathbb{N}$

- $f(n) = O(g(n))$ if there is a constant C s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq C$$

- $f(n) = \Omega(g(n))$ if there is a constant c s.t.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$

- $f(n) = \Theta(g(n))$ if $f(n) = O(n)$ and $f(n) = \Omega(g(n))$.
  Equivalently, there is constant C such that:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$$

Examples:

- $10^{10} \cdot n^3 = \Theta(n^3)$
- $10n^3 + 100n^2 + n = \Theta(n^3)$

Asymptotics recap

- $f(n) = o(g(n))$ if

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

Examples:

- $10^{10} \cdot n^2 = o(n^3)$
- $10n^3 + 100n^2 + n = o(2^n)$
- $10n^3 + 100n^2 + n = o(n^3 \log n)$

Asymptotics recap

- $f(n) = o(g(n))$ if
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
- $f(n) = \omega(g(n))$ if
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Examples:

- $10^{-10} \cdot n^3 = \omega(n^2)$
- $10n^3 + 100n^2 + n = \omega(n)$

Practice questions

Compare the following functions:

1. n^5 vs $n^5 / \log \log n$
2. $2^{\sqrt{n}}$ vs $n^{\log n}$
3. $n!$ vs 2^n
4. n^n vs $2^{n \log n}$

Worst case complexity

- An algorithm "runs in time" $O(f(n))$ if there is a constant $C > 0$ s.t., on inputs of size n, it requires at most $C \cdot f(n)$ elementary operations to output a correct answer.

Worst case complexity

- An algorithm "runs in time" $O(f(n))$ if there is a constant $C > 0$ s.t., on inputs of size n, it requires at most $C \cdot f(n)$ elementary operations to output a correct answer.
- "Mathematically:" given algorithm A and input x, let $T_A(x)$ be running time of algorithm A on input x.

Worst-case running time is:

$$T_A(n) = \max_{size(x)=n} T_A(x)$$

Worst case complexity

- An algorithm "runs in time" $O(f(n))$ if there is a constant $C > 0$ s.t., on inputs of size n, it requires at most $C \cdot f(n)$ elementary operations to output a correct answer.
- "Mathematically:" given algorithm A and input x, let $T_A(x)$ be running time of algorithm A on input x.

  Worst-case running time is:

$$T_A(n) = \max_{size(x)=n} T_A(x)$$

- Asymptotic notation allows us to focus on main growth of complexity
  - ignore leading constant
  - ignore lower order terms

Worst case complexity

- An algorithm "runs in time" $O(f(n))$ if there is a constant $C > 0$ s.t., on inputs of size n, it requires at most $C \cdot f(n)$ elementary operations to output a correct answer.
- "Mathematically:" given algorithm A and input x, let $T_A(x)$ be running time of algorithm A on input x.

  Worst-case running time is:

$$T_A(n) = \max_{size(x)=n} T_A(x)$$

- Asymptotic notation allows us to focus on main growth of complexity
  - ignore leading constant
  - ignore lower order terms
- For instance:
  - binary search runs in time $O(\log n)$
  - sorting (using say merge-sort) runs in time $O(n \log n)$

Efficient algorithms

- with concept of asymptotic analysis, when will an algorithm be "efficient"?

An algorithm is "efficient" when there is a constant $\gamma > 0$ such that the algorithm runs in time $O(n^\gamma)$

Polynomial time.

Efficient algorithms

- with concept of asymptotic analysis, when will an algorithm be "efficient"?

An algorithm is "efficient" when there is a constant $\gamma > 0$ such that the algorithm runs in time $O(n^\gamma)$

Polynomial time.

- Of course, the smaller the constant γ, the more efficient our algorithm will be.

Efficient algorithms

- with concept of asymptotic analysis, when will an algorithm be "efficient"?

  An algorithm is "efficient" when there is a constant $\gamma > 0$ such that the algorithm runs in time $O(n^\gamma)$

  Polynomial time.

- Of course, the smaller the constant γ, the more efficient our algorithm will be.
- Why care so much about polynomial time?
  - Composition (i.e. can use subroutines)
  - For many problems, "trivial" algorithms run in exponential time (i.e. $2^{n^{O(1)}}$)

"Practical" algorithms

- "Practice" depends on the setting that one is working on, thus it is loosely defined
  - some settings this means nearly linear time ($O(n \log^c n)$)
  - sometimes even *sub-linear* time! (CS 466)
  - other times fast for *most* inputs
  - other times for small enough inputs (leading constant matters)
  - etc.

“Practical” algorithms

- “Practice” depends on the setting that one is working on, thus it is loosely defined
  - some settings this means nearly linear time ($O(n \log^c n)$)
  - sometimes even *sub-linear* time! (CS 466)
  - other times fast for *most* inputs
  - other times for small enough inputs (leading constant matters)
  - etc.

But in all the above, always taking care of the *leading constants!*

"Practical" algorithms

- "Practice" depends on the setting that one is working on, thus it is loosely defined
  - some settings this means nearly linear time ($O(n \log^c n)$)
  - sometimes even *sub-linear* time! (CS 466)
  - other times fast for *most* inputs
  - other times for small enough inputs (leading constant matters)
  - etc.

But in all the above, always taking care of the *leading constants!*

For instance, an algorithm running in time $100n^3$ is much better (in practice) than one which runs in time $2^{1000} \cdot n$.

- Computational Models
- Time Complexity & Efficiency
- Examples: 2SUM & 3SUM
- Acknowledgements

3-SUM problem

- **Input:** Set of integers $\{a_1, \ldots, a_n\}$, integer c
- **Output:** $\begin{cases} YES, & \text{if } \exists\ i, j, k \in [n] \text{ such that } a_i + a_j + a_k = c \\ \text{NO, otherwise} \end{cases}$

3-SUM problem

- **Input:** Set of integers $\{a_1, \ldots, a_n\}$, integer c
- **Output:** $\begin{cases} YES, & \text{if } \exists\ i, j, k \in [n] \text{ such that } a_i + a_j + a_k = c \\ \text{NO, otherwise} \end{cases}$
- Naive algorithm: for each triple i, j, k, check whether $a_i + a_j + a_k = c$

  Running time: $O(n^3)$ (4 ops to check each triple)

Can we do better?

3-SUM problem

- **Input:** Set of integers $\{a_1, \ldots, a_n\}$, integer c
- **Output:** $\begin{cases} YES, & \text{if } \exists\ i, j, k \in [n] \text{ such that } a_i + a_j + a_k = c \\ \text{NO, otherwise} \end{cases}$
- Naive algorithm: for each triple i, j, k, check whether $a_i + a_j + a_k = c$

  Running time: $O(n^3)$ (4 ops to check each triple)

Can we do better?

- Less naive:
  1. Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
  2. For each pair i, j, let $b_{i,j} = c - a_i - a_j$
  3. Binary search to check if there is k such that $a_k = b_{i,j}$

  Running time: $O(n^2 \log n + n \log n) = O(n^2 \log n)$

Can we do better?

Last attempt

- Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
- For each $k \in [n]$, let $b_k := c - a_k$
- Decide if there are $i, j \in [n]$ such that $a_i + a_j = b_k$

Last attempt

- Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
- For each $k \in [n]$, let $b_k := c - a_k$
- Decide if there are $i, j \in [n]$ such that $a_i + a_j = b_k$
- **2-SUM problem:** given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

Last attempt

- Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
- For each $k \in [n]$, let $b_k := c - a_k$
- Decide if there are $i, j \in [n]$ such that $a_i + a_j = b_k$
- **2-SUM problem:** given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

if we can solve the 2-SUM problem, then can solve 3-SUM by "calling" 2-SUM for each $k \in [n]$

Reduction!

Last attempt

- Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
- For each $k \in [n]$, let $b_k := c - a_k$
- Decide if there are $i, j \in [n]$ such that $a_i + a_j = b_k$
- **2-SUM problem:** given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

  if we can solve the 2-SUM problem, then can solve 3-SUM by "calling" 2-SUM for each $k \in [n]$

  Reduction!
- Running time $= O(n \times (\text{running time for 2-SUM}) + n \log n)$

Last attempt

- Sort the set of numbers, so can assume we have $a_1 \leq a_2 \leq \cdots \leq a_n$
- For each $k \in [n]$, let $b_k := c - a_k$
- Decide if there are $i, j \in [n]$ such that $a_i + a_j = b_k$
- **2-SUM problem:** given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

  if we can solve the 2-SUM problem, then can solve 3-SUM by "calling" 2-SUM for each $k \in [n]$

  Reduction!
- Running time $= O(n \times \text{(running time for 2-SUM)} + n \log n)$

  Can we do 2-SUM with running time better than $O(n \log n)$?

2-SUM

- given $a_1 \le a_2 \le \cdots \le a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

2-SUM

- given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?
- Idea: see board

2-SUM

- given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

- Algorithm:
  1. Write $\beta_i := b - a_i$ for each $i \in [n]$, and let $j, t \in [n]$ be counters, initially set to $j = 1$ and $t = n$.
  2. While $t > 0$:
     - if $\beta_j > a_t$, then $j \leftarrow j + 1$
     - if $\beta_j = a_t$, then return YES
     - else (i.e. $\beta_j < a_t$), then $t \leftarrow t - 1$
  3. Return NO

2-SUM

- given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

- Algorithm:
  1. Write $\beta_i := b - a_i$ for each $i \in [n]$, and let $j, t \in [n]$ be counters, initially set to $j = 1$ and $t = n$.
  2. While $t > 0$:
     - if $\beta_j > a_t$, then $j \leftarrow j + 1$
     - if $\beta_j = a_t$, then return YES
     - else (i.e. $\beta_j < a_t$), then $t \leftarrow t - 1$
  3. Return NO
- Running time: n executions of the loop, each loop iteration takes at most 2 operations.

Thus: $O(n)$

2-SUM

- given $a_1 \leq a_2 \leq \cdots \leq a_n$ and b, are there $i, j \in [n]$ such that $a_i + a_j = b$?

- Algorithm:
  1. Write $\beta_i := b - a_i$ for each $i \in [n]$, and let $j, t \in [n]$ be counters, initially set to $j = 1$ and $t = n$.
  2. While $t > 0$:
     - if $\beta_j > a_t$, then $j \leftarrow j + 1$
     - if $\beta_j = a_t$, then return YES
     - else (i.e. $\beta_j < a_t$), then $t \leftarrow t - 1$
  3. Return NO
- Running time: n executions of the loop, each loop iteration takes at most 2 operations.

$$\text{Thus: } O(n)$$

- So the running time of our last 3-SUM algorithm is

$$O(n^2 + n \log n) = O(n^2)$$

Conclusion

- Computational models (basis for modeling computation)
- Running time dependent on the model
- Efficient algorithms (beating exhaustive search)
- Reductions
- flavour of course

Acknowledgement

- Based on Lap Chi's first lecture

  https://cs.uwaterloo.ca/~lapchi/cs341/notes/L01.pdf

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)

Introduction to Algorithms, third edition.

MIT Press