Discrete Mathematics and Its Application 2 (CS147)

Lecture 3: Worst-case asymptotic running time

Fanghui Liu

Department of Computer Science, University of Warwick, UK

Target

Analyzing the runtime of an algorithm for a computational problem using Big-O notation.

Target

Analyzing the runtime of an algorithm for a computational problem using Big-O notation.

- Q1: What is a computational problem?

Target

Analyzing the runtime of an algorithm for a computational problem using Big-O notation.

- Q1: What is a computational problem?
- Q2: What is an algorithm?

Target

Analyzing the runtime of an algorithm for a computational problem using Big-O notation.

- Q1: What is a computational problem?
- Q2: What is an algorithm?
- Q3: How to define an algorithm's runtime?

Computational problem (decision problem)

Example

Input - An array $A[1, 2, \ldots, n]$ of n numbers.
$\Leftrightarrow A[1], A[2], A[3], \cdots, A[n]$
For each $i \in \{1, 2, \cdots, n\}$, $A[i]$ is a real number.

Computational problem (decision problem)

Example

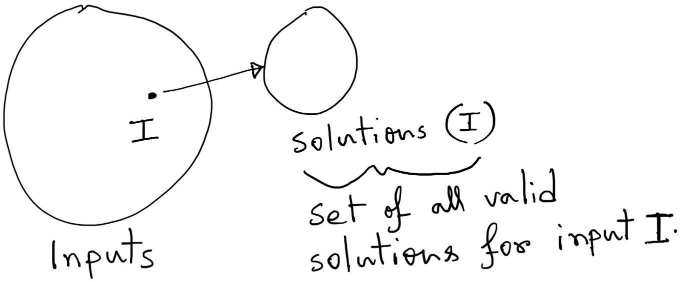
Input - An array $A[1, 2, \ldots, n]$ of n numbers.
Output

- `Yes` if there exist indices $i, j \in \{1, 2, \cdots, n\}$ with $i \neq j$ such that $A[i] + A[j] = 0$.
- `No` otherwise.

Computational problem `P` (say)
○ the class of decision problems that are solvable in polynomial time

Computational problem (decision problem)

In the previous example, for all inputs I, either solutions(I)=`Yes` or Solutions(I) = `No`. A computational problem `P` consists of:



- decision problem: the answer for every input is either yes or no
- search problem, counting problem, optimization problem...

Algorithm

Definition

An algorithm for a computational problem $\mathtt{P}$ is a step by step procedure such that given any input I, outputs a valid solution for I.

Algorithm

Definition

An algorithm for a computational problem P is a step by step procedure such that given any input I, outputs a valid solution for I.

- Input: I
- Output: A valid solution for I

An algorithm for our example problem

An example algorithm (Pseudocode)

For $i = 1, 2, \ldots, n$
    For $j = i+1, i+2, \ldots, n$
        If $A[i] + A[j] = 0$
            Return `YES`.
Return `No`.

You can implement this algorithm by writing a computer program in C, C++, Java etc.

An algorithm for our example problem

An example algorithm (Pseudocode)

For $i = 1, 2, \ldots, n$
  For $j = i + 1, i + 2, \ldots, n$
    If $A[i] + A[j] = 0$
      Return `YES`.
Return `No`.

You can implement this algorithm by writing a computer program in C, C++, Java etc.

c/c++

```
//Using for-loops to add numbers 1 - 5
int sum = 0;
for (int i = 1; i <= 5; ++i) {
    sum += i;
}
```

Python

```
for i in range(1, 6):  # gives i values from 1 to 5 inclusive (but not 6)
    # statements
    print(i)
# if we want 6 we must do the following
for i in range(1, 6 + 1):  # gives i values from 1 to 6
    # statements
    print(i)
```

Algorithms vs. Programs

- The same algorithm can be implemented in different programming languages.
- An algorithm is an abstract mathematical object, independent of
  - the programming language it has been implemented in
  - the machine (computer) it is running on.

Runtime of an program and algorithm

Statement

Runtime of a computer program = the actual time (say, in microseconds) if it takes to finish execution.

- It depends on
  - the input
  - the programming language
  - the machine (computer)

Example: Runtime of an algorithm

An example algorithm (Pseudocode)
(take c_1 time) **1.** For $i = 1, 2, \ldots, n$
(take c_2 time) **2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time) **3.** If $A[i] + A[j] = 0$
(take c_4 time) **4.** Return $\texttt{YES}$.
(take c_5 time) **5.** Return $\texttt{No}$.

Example: Runtime of an algorithm

An example algorithm (Pseudocode)
(take c_1 time) **1.** For $i = 1, 2, \ldots, n$
(take c_2 time) **2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time) **3.** If $A[i] + A[j] = 0$
(take c_4 time) **4.** Return `YES`.
(take c_5 time) **5.** Return `No`.

- Line **1.** total time $\leq c_1 n$

Example: Runtime of an algorithm

An example algorithm (Pseudocode)
(take c_1 time) **1.** For $i = 1, 2, \ldots, n$
(take c_2 time) **2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time) **3.** If $A[i] + A[j] = 0$
(take c_4 time) **4.** Return `YES`.
(take c_5 time) **5.** Return `No`.

- Line **1.** total time $\leq c_1 n$
- Line **2.** total time $\leq \sum_{i=1}^{n} c_2(n-i)$

Example: Runtime of an algorithm

An example algorithm (Pseudocode)	
(take c_1 time)	**1.** For $i = 1, 2, \ldots, n$
(take c_2 time)	**2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time)	**3.** If $A[i] + A[j] = 0$
(take c_4 time)	**4.** Return `YES`.
(take c_5 time)	**5.** Return `No`.

- Line **1.** total time $\leq c_1 n$
- Line **2.** total time $\leq \sum_{i=1}^{n} c_2(n-i)$
- Line **3.** total time $\leq \sum_{i=1}^{n} c_3(n-i)$

Example: Runtime of an algorithm

An example algorithm (Pseudocode)
(take c_1 time) **1.** For $i = 1, 2, \ldots, n$
(take c_2 time) **2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time) **3.** If $A[i] + A[j] = 0$
(take c_4 time) **4.** Return `YES`.
(take c_5 time) **5.** Return `No`.

- Line **1.** total time $\leq c_1 n$
- Line **2.** total time $\leq \sum_{i=1}^{n} c_2(n-i)$
- Line **3.** total time $\leq \sum_{i=1}^{n} c_3(n-i)$
- Line **4.** total time $\leq \sum_{i=1}^{n} c_4(n-i)$

Example: Runtime of an algorithm

An example algorithm (Pseudocode)	
(take c_1 time)	**1.** For $i = 1, 2, \ldots, n$
(take c_2 time)	**2.** For $j = i+1, i+2, \ldots, n$
(take c_3 time)	**3.** If $A[i] + A[j] = 0$
(take c_4 time)	**4.** Return `YES`.
(take c_5 time)	**5.** Return `No`.

- Line **1.** total time $\leq c_1 n$
- Line **2.** total time $\leq \sum_{i=1}^{n} c_2(n-i)$
- Line **3.** total time $\leq \sum_{i=1}^{n} c_3(n-i)$
- Line **4.** total time $\leq \sum_{i=1}^{n} c_4(n-i)$
- Line **5.** total time $\leq c_5$

Example: Runtime of an algorithm

An example algorithm (Pseudocode)
(take c_1 time) **1.** For $i = 1, 2, \dots, n$
(take c_2 time) **2.** For $j = i+1, i+2, \dots, n$
(take c_3 time) **3.** If $A[i] + A[j] = 0$
(take c_4 time) **4.** Return `YES`.
(take c_5 time) **5.** Return `No`.

- Line **1.** total time $\le c_1 n$
- Line **2.** total time $\le \sum_{i=1}^{n} c_2(n-i)$
- Line **3.** total time $\le \sum_{i=1}^{n} c_3(n-i)$
- Line **4.** total time $\le \sum_{i=1}^{n} c_4(n-i)$
- Line **5.** total time $\le c_5$

Total time spent on an input of size n

$$\le c_1 n + \sum_{i=1}^{n} c_2(n-i) + \sum_{i=1}^{n} c_3(n-i) + \sum_{i=1}^{n} c_4(n-i) + c_5 = \Theta(n^2)$$

- On every input of size n, the algorithm spends **at most** $\Theta(n^2)$ time.

- On every input of size n, the algorithm spends **at most** $\Theta(n^2)$ time.
- **Input sensitivity:** There are inputs of size n on which the algorithm spends only $\Theta(1)$ time. [e.g., $A[1] + A[2] = 0$]

- On every input of size n, the algorithm spends **at most** $\Theta(n^2)$ time.
- **Input sensitivity:** There are inputs of size n on which the algorithm spends only $\Theta(1)$ time. [e.g., $A[1] + A[2] = 0$]
- **Input sensitivity:** There are inputs of size n on which the algorithm spends $\Theta(n^2)$ time. [e.g., $A[1] = 1, A[2] = 2, \ldots, A[n-1] = n-1$ and $A[n] = -(n-1)$]

- On every input of size n, the algorithm spends **at most** $\Theta(n^2)$ time.
- **Input sensitivity:** There are inputs of size n on which the algorithm spends only $\Theta(1)$ time. [e.g., $A[1] + A[2] = 0$]
- **Input sensitivity:** There are inputs of size n on which the algorithm spends $\Theta(n^2)$ time. [e.g., $A[1] = 1, A[2] = 2, \ldots, A[n-1] = n-1$ and $A[n] = -(n-1)$]

We will say that the algorithm has a runtime of $\Theta(n^2)$.

Runtime of an algorithm

Formally, let

$$f(n) = \max_{\text{input } I \text{ of size } n} (\text{runtime of the algorithm on input } I)$$

We will focus on how $f(n)$ grows with input size n, asymptotically.
$\Leftrightarrow$ Worst-case asymptotic running time of an algorithm.

Worst-case asymptotic running time

- Is a feature of an algorithm for a given computational problem

Worst-case asymptotic running time

- Is a feature of an algorithm for a given computational problem
- Independent of
  - programming language
  - specific input
  - machine (computer)

Worst-case asymptotic running time

- Is a feature of an algorithm for a given computational problem
- Independent of
  - programming language
  - specific input
  - machine (computer)
- Allows us to compare two different algorithms for the same problem.

Worst-case asymptotic running time

- An $\mathcal{O}(n^2)$ time algorithm is better than an $\mathcal{O}(n^3)$ time algorithm
- An $\mathcal{O}(n \log n)$ time algorithm is better than an $\mathcal{O}(n^2)$ time algorithm

Goal

Given a computational problem, our goal will be to find an algorithm for it with smallest possible worst-case asymptotic runtime.

Runtime of an algorithm

Statement

- *constant time $\mathcal{O}(1)$: arithmetic/logic operation, access one element in an array*
- *linear time $\mathcal{O}(n)$: merge two sorted arrays (Lecture 5)*
- *quadratic time $\mathcal{O}(n^2)$: bubble sort (Lecture 4)*
- *logarithmic time $\mathcal{O}(\log n)$: binary search (Lecture 6)*
- *linearithmic time $\mathcal{O}(n \log n)$: merge sort (Lecture 5)*

Beyond the worst case runtime analysis

- the best case: find one input that the algorithm can perform the best
- the average case: **averaged** over all possible inputs for *randomized* algorithm
- runtime analysis vs. memory analysis

*Examples in TCS, ML theory

Theorem 6 (arbitrary loss)**.** *From random initialization, with probability at least* $1 - e^{-\Omega(\log^2 m)}$*, gradient descent with appropriate learning rate satisfy the following.*

- *If* f *is nonconvex but* σ*-gradient dominant (a.k.a. Polyak-Łojasiewicz), GD finds* ε*-error minimizer in*[14]
$$\boxed{T = \widetilde{O}\big(\tfrac{\mathsf{poly}(n,L)}{\sigma\delta^2} \cdot \log \tfrac{1}{\varepsilon}\big) \text{ iterations}}$$
as long as $\boxed{m \geq \widetilde{\Omega}\big(\mathsf{poly}(n, L, \delta^{-1}) \cdot d\sigma^{-2}\big)}$.
- *If* f *is convex, then GD finds* ε*-error minimizer in*
$$T = \widetilde{O}\big(\tfrac{\mathsf{poly}(n,L)}{\delta^2} \cdot \tfrac{1}{\varepsilon}\big) \text{ iterations}$$
as long as $m \geq \widetilde{\Omega}\big(\mathsf{poly}(n, L, \delta^{-1}) \cdot d \log \varepsilon^{-1}\big)$.

Figure: time complexity and parameter complexity [AZLS19].

Corollary 1.3. *Let* $\mathcal{D}$ *be the distribution over pairs* $(x, y) \in \mathbb{R}^d \times \mathbb{R}$ *where* $x \sim \mathcal{N}(0, \mathrm{Id})$ *and* $y = F(x)$ *for a size-*S *ReLU network* F *for which the product of the spectral norms of its weight matrices is a constant.*

Then there is an algorithm that draws $N = d\log(1/\delta)\exp(O(k^3/\varepsilon^2 + kS))$ *samples, runs in time* $\widetilde{O}(d^2\log(1/\delta))\exp(O(k^3S^2/\varepsilon^2 + kS^3))$*, and outputs a ReLU network* $\widetilde{F}$ *such that* $\mathbb{E}[(y - \widetilde{F}(x))^2] \leq \varepsilon$ *with probability at least* $1 - \delta$.

Figure: Recall the example in Lecture 1: sample complexity and time complexity [CKM22].

References I

[0] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song, *A convergence theory for deep learning via over-parameterization*, International Conference on Machine Learning, PMLR, 2019, pp. 242–252.
(Cited on page 33.)

[0] Sitan Chen, Adam R Klivans, and Raghu Meka, *Learning deep relu networks is fixed-parameter tractable*, 2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS), IEEE, 2022, pp. 696–707.
(Cited on page 33.)