1: Algorithm Analysis **Fall 2021**

Lecture 1: Computational Complexities of Algorithms

Lecturer: None *Scribes: None*

In this note, we will explore some examples in analyzing computational complexities of algorithms, especially of those mathematics-oriented optimization algorithms and data structure algorithms. The readers are encouraged to first familiarize them-selves with the so-called Big-$\mathcal{O}$ notation, which has been elaborated in the book [Cormen09] and this Wikipedia page. This Stack Overflow page may also help.

1.1 Computational Complexity Analysis: A Case-by-Case Study

Example 1.1 ("Hello World!") Algorithm 1 below has a computational complexity of $\mathcal{O}(n)$.

Algorithm 1 "Hello World!"

Input: a number $n \in \mathbb{N}_+$
Output: $\varnothing$

```
1: initialize i as 0
2: while i < n do
3:     print("Hello World!")        ▷ This operation can be done in O(1) time!
4: end while
```

Explanation: It is obvious that the expression "print(Hello World!)" will be executed n times, and print$(\cdot)$ has a computational complexity of $\mathcal{O}(1)$, hence the claim holds. ■

Remark 1.2 The readers may still be confused about why n times of an $\mathcal{O}(1)$ operation yields an $\mathcal{O}(n)$ operation. In fact, $\underbrace{\mathcal{O}(1) + \mathcal{O}(1) + \ldots + \mathcal{O}(1)}_{n \text{ times}} \subseteq \mathcal{O}(n)$. This is because

- "$\subseteq$": $\forall f(n) \in \underbrace{\mathcal{O}(1) + \mathcal{O}(1) + \ldots + \mathcal{O}(1)}_{n \text{ times}}$, $f(n) = f_n(n) + \ldots + f_1(n)$, where $f_i(n) \in \mathcal{O}(1)$, $\forall i = 1, \ldots, n$, with for each i, $|f_i(n)| \leq c_i 1$ whenever $n \geq N_i$. Hence, for $n \geq \max(\{N_n, \ldots, N_1\})$, we have

$$|f(n)| \underset{\text{(triangle inequality)}}{\leq} |f_n(n)| + \ldots + |f_1(n)| \underset{\text{(by definition)}}{\leq} c_n 1 + c_{n-1} 1 + \ldots + c_1 1 \leq \left(\max_{i=1,\ldots,n} c_i\right)\left(\underbrace{1 + 1 + \ldots + 1}_{n \text{ times}}\right) = \left(\max_{i=1,\ldots,n} c_i\right) n,$$

  implying $f(n) \in \mathcal{O}(n)$. Hence, $\underbrace{\mathcal{O}(1) + \mathcal{O}(1) + \ldots + \mathcal{O}(1)}_{n \text{ times}} \subseteq \mathcal{O}(n)$.

Therefore, our claim holds. □

Example 1.3 Algorithm 2 below has a computational complexity of $\mathcal{O}(mn)$.

Algorithm 2 An algorithm with double loops

Input: two numbers $(m, n) \in \mathbb{N}_+^2$
Output: $\varnothing$

for $i = 1, \ldots, m$ **do**
  for $j = 1, \ldots, n$ **do**
    do something in $\mathcal{O}(1)$ time
  end for
end for

Explanation: Obviously, the $\mathcal{O}(1)$ expression will be executed mn times. ■

Exercise 1.4 Verify by yourself that the computational complexity of matrix multiplication of $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{B} \in \mathbb{R}^{m \times p}$, whose pseudo-code is given in Algorithm 3 below, is $\mathcal{O}(nmp)$.

Algorithm 3 Matrix multiplication

Input: $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{B} \in \mathbb{R}^{m \times p}$
Output: $\boldsymbol{AB}$

initialize $\boldsymbol{C} \in \{0\}^{n \times p}$
for $i = 1, \ldots, n$ **do**
  for $j = 1, \ldots, p$ **do**
    for $k = 1, \ldots, m$ **do**
      $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$
    end for
  end for
end for
return $\boldsymbol{C}$

Next, we step into some more complicated complexity analysis cases.

Example 1.5 (Factorization Machine) The objective function of the canonical Factorization Machine (with intercept term w_0 ignored) is given as follows

$$f_{\mathrm{FM}}(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{P}) := \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{j_2 > j_1} \left\langle \boldsymbol{p}_{j_1}, \boldsymbol{p}_{j_2} \right\rangle x_{j_1} x_{j_2},$$

where $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{w} \in \mathbb{R}^d$ and $\boldsymbol{P} \in \mathbb{R}^{k \times d}$. Here, $\boldsymbol{x}$ is a sparse vector, and we denote its number of non-zero elements as $\mathrm{nnz}(\boldsymbol{x})$. By mathematical manipulations, $f_{\mathrm{FM}}(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{P})$ can be re-arranged as

$$f_{\mathrm{FM}}(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{P}) := \langle \boldsymbol{w}, \boldsymbol{x} \rangle + \sum_{j_2 > j_1} \left\langle \boldsymbol{p}_{j_1}, \boldsymbol{p}_{j_2} \right\rangle x_{j_1} x_{j_2} = \sum_{j=1}^{d} w_j x_j + \frac{1}{2} \sum_{f=1}^{k} \left[\left(\sum_{j=1}^{d} p_{fj} x_j \right)^2 - \sum_{j=1}^{d} p_{fj}^2 x_j^2 \right].$$

(If one wants to derive this by his/her own, then the following figure may help.)

We claim that the computational complexity for evaluating the right-most objective function of FM is $\mathcal{O}(\mathrm{nnz}(\boldsymbol{x})k)$.

Explanation: The evaluation consists of two steps.

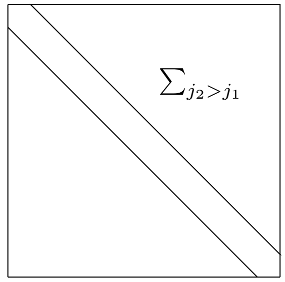

Figure 1.1: A big matrix divided into three parts.

1. **Evaluating the linear regression term $\sum_{j=1}^d w_j x_j$:** This expression is an inner product. Note that $\boldsymbol{x}$ is sparse, hence we can compute this in the most efficiency way as in Algorithm 4, where $\text{supp}(\boldsymbol{x}) = \{i \mid x_i \neq 0\}$. Hence, the computation complexity in this part is $\mathcal{O}(\text{nnz}(\boldsymbol{x}))$.

Algorithm 4 Inner product of $\boldsymbol{x}$ (which is sparse) and $\boldsymbol{w}$

Input: $\boldsymbol{x} \in \mathbb{R}^d$ and $\boldsymbol{w} \in \mathbb{R}^d$
Output: $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$

```
initialize c = 0
for i ∈ supp(x) do
    c ← c + x_i w_i
end for
return c
```

---

2. **Evaluating the quadratic regression term $\frac{1}{2}\sum_{f=1}^k [(\sum_{j=1}^d p_{fj} x_j)^2 - \sum_{j=1}^d v_{fj}^2 x_j^2]$:** In a similar vein, we can figure out the computational complexity in this part as follows (to avoid redundancy, pseudo-code is omitted here). Recall that $\boldsymbol{x}$ is sparse, hence $\sum_{j=1}^d p_{fj} x_j$ and $\sum_{j=1}^d v_{fj}^2 x_j^2$ can both be computed in $\mathcal{O}(\text{nnz}(\boldsymbol{x}))$ time, and thus the time complexity of computing $(\sum_{j=1}^d p_{fj} x_j)^2 - \sum_{j=1}^d v_{fj}^2 x_j^2$ is $\mathcal{O}(\text{nnz}(\boldsymbol{x}))$. Therefore, the total computational complexity in this part is $\mathcal{O}(\text{nnz}(\boldsymbol{x})k)$.

Notice that the complexity family $\mathcal{O}(\text{nnz}(\boldsymbol{x})) \subseteq \mathcal{O}(\text{nnz}(\boldsymbol{x})k)$, the claim thus holds. ■

Remark 1.6 The readers may still be confused about why $g(n) \leq h(n)$ implies $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$. This is because $\forall f(n) \in \mathcal{O}(g(n))$, for all $n \geq N_0$, we have

$$|f(n)| \leq \underbrace{cg(n)}_{\text{(by definition)}} \leq ch(n),$$

implying $f(n) \in \mathcal{O}(h(n))$. Thence, $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$. Similarly, one can also discover that $g(n) \leq h(n)$ implies $\mathcal{O}(g(n)) + \mathcal{O}(h(n)) \in \mathcal{O}(h(n))$.

The above relation can be easily extended to the cases of multiple functions. □

Below, we will encounter some examples involving recursion. Many of them are borrowed from [Jordi].

Example 1.7 Assume $n \geq 1$, Algorithm 5 below has a computational complexity of $\mathcal{O}(n^2)$.

Explanation: It is easy to observe that the computational complexity of Algorithm 5 is $\mathcal{O}(n) + \mathcal{O}(n-1) + \ldots + \mathcal{O}(1) \subseteq \mathcal{O}(n(n+1)/2) = \mathcal{O}(n^2)$, hence the claim holds. ■

Algorithm 5 Recursion 1

Input: a number $n \in \mathbb{Z}$
Output: $\varnothing$

if $n \geq 1$ **then**
  do something in $\mathcal{O}(n)$ time ▷ Note that the n here is consistent with the input.
  call Algorithm 5 (i.e. itself) with input as $n - 1$
end if

Remark 1.8 The inclusion $\mathcal{O}(n) + \mathcal{O}(n-1) + \ldots + \mathcal{O}(1) \subseteq \mathcal{O}(n(n+1)/2)$ holds, because

- "$\subseteq$": $\forall f(n) \in \mathcal{O}(n) + \mathcal{O}(n-1) + \ldots + \mathcal{O}(1)$, $f(n) = f_n(n) + \ldots + f_1(n)$, where $f_i(n) \in \mathcal{O}(i)$, $\forall i = 1, \ldots, n$, with for each i, $|f_i(n)| \leq c_i i$ whenever $n \geq N_i$. Hence, for $n \geq \max(\{N_n, \ldots, N_1\})$, with the triangle inequality $|f(n)| \leq |f_n(n)| + \ldots + |f_1(n)|$, we have

$$|f(n)| \leq \underbrace{c_n n + c_{n-1}(n-1) + \ldots + c_1 1}_{\text{(by definition)}} \leq \left(\max_{i=1,\ldots,n} c_i\right)(n + (n-1) + \ldots + 1) = \left(\max_{i=1,\ldots,n} c_i\right)\left(\frac{n(n+1)}{2}\right),$$

  implying $f(n) \in \mathcal{O}(n(n+1)/2)$. Hence, $\mathcal{O}(n) + \mathcal{O}(n-1) + \ldots + \mathcal{O}(1) \subseteq \mathcal{O}(n(n+1)/2)$.

Therefore, our claim holds. In the sequel, we will omit these intermediate steps. □

Example 1.9 Assume $n \geq 1$, Algorithm 6 below has a computational complexity of $\mathcal{O}(n)$.

Algorithm 6 Recursion 2

Input: a number $n = 2^k$ for some $k \in \mathbb{Z}$
Output: $\varnothing$

if $n \geq 1$ **then**
  do something in $\mathcal{O}(n)$ time ▷ Note that the n here is consistent with the input.
  call Algorithm 6 (i.e. itself) with input as $n/2$
end if

Explanation: Obviously, the computational complexity is $\mathcal{O}(n) + \mathcal{O}(n/2) + \ldots + \mathcal{O}(1) \subseteq \mathcal{O}(n((1/2^0) + (1/2^1) + (1/2^2) + \ldots + (1/2^{\log_2(n)}))) = \mathcal{O}(n(2 - (1/n))) = \mathcal{O}(2n - 1) = \mathcal{O}(n)$. ■

Exercise 1.10 Assume $n \geq 1$, what is the computational complexity of Algorithm 7 below?

Algorithm 7 Recursion 3

Input: a number $n = 2^k$ for some $k \in \mathbb{Z}$
Output: $\varnothing$

if $n \geq 1$ **then**
  do something in $\mathcal{O}(n)$ time ▷ Note that the n here is consistent with the input.
  call Algorithm 7 (i.e. itself) with input as $n/2$
  call again Algorithm 7 (i.e. itself) with input as $n/2$
end if

Example 1.11 (Inorder Traversal) The *inorder traversal* algorithm for iterating over a tree is given in Algorithm 8 below. Assume there are n nodes in the tree, then the computational complexity of Algorithm 8 is $\mathcal{O}(n)$.

Algorithm 8 Inorder traversal

Input: the root node r of a tree (or sub-tree)

Output: $\varnothing$

if $r = \varnothing$ **then**
  return
end if
call Algorithm 8 with input as the left child of r
do something with r in $\mathcal{O}(1)$ time ▷ Such as printing r.
call Algorithm 8 with input as the right child of r

Explanation: This is because the $\mathcal{O}(1)$ time expression will be executed exactly n times (i.e., every node r will and only will be visited once). ■

Example 1.12 (Deep-first search) Algorithm 9 below shows the pseudo-code of iterating over a graph by the well-known *deep-first search* algorithm. Assume the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ has n nodes and m edges, then the computational complexity, of repeatedly calling Algorithm 9 for all $v \in \mathcal{V} \setminus \{u \in \mathcal{V} \mid u \text{ has been visited}\}$ until all nodes have been visited, is $\mathcal{O}(n + m)$.

I have revised the expression here, which is unclear in the original manuscript. I apologize for any misleading it has made.

Algorithm 9 Deep-first search

Input: a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and a node $v \in \mathcal{V}$

Output: $\varnothing$

mark v as visited ▷ This can be done in $\mathcal{O}(1)$ time.
do something with v in $\mathcal{O}(1)$ time
for all $u \in \mathcal{N}(v)$ **do**
  call Algorithm 9 with inputs $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and u, if u has not been visited yet
end for

Explanation: In Algorithm 9, every node in $\mathcal{V}$ will be visited only once. Besides, when visiting a specific node v, the $\mathcal{O}(1)$ operations will be executed once, and the whole neighborhood of v (i.e. $\mathcal{N}(v)$) will be checked once (which has a time complexity of $\mathcal{O}(\deg(v))$). Hence, the total computational complexity is $\sum_{v \in \mathcal{V}} (\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(\deg(v))) \subseteq \mathcal{O}(2n + 2m) = \mathcal{O}(n + m)$. ■

Remark 1.13 Notice that we are unable to merge $\mathcal{O}(1) + \mathcal{O}(\deg(v)) = \mathcal{O}(\deg(v))$, because $\deg(v)$ can sometimes be 0. □

Next, we will deal with some complicated computational complexity analysis problems, arising in numerical computation and optimization problems, with the help of existing results (see this Wikipedia page).

Example 1.14 Given $\boldsymbol{A} \in \mathbb{R}^{m \times m}$ which is invertible and $\boldsymbol{B} \in \mathbb{R}^{m \times n}$, the computational complexity of computing $\boldsymbol{A}^{-1}\boldsymbol{B}$ is $\mathcal{O}(m^3 + m^2 n)$.

Explanation: The computation consists of two steps.

1. **Computing $\boldsymbol{A}^{-1}$:** Referring to the Wikipedia page above, the computational complexity of computing $\boldsymbol{A}^{-1}$ is $\mathcal{O}(m^3)$.
2. **Computing $\boldsymbol{A}^{-1}\boldsymbol{B}$:** With $\boldsymbol{A}^{-1}$ computed, the computation of $\boldsymbol{A}^{-1}\boldsymbol{B}$ is just a matter of matrix multiplication, whose computational complexity is $\mathcal{O}(m^2 n)$.

To sum up, the total computational complexity of computing $\boldsymbol{A}^{-1}\boldsymbol{B}$ is $\mathcal{O}(m^3 + m^2 n)$. ■

Example 1.15 Given three matrices $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times p}$ and $\boldsymbol{C} \in \mathbb{R}^{p\times q}$. The computational complexity of evaluating $\boldsymbol{ABC}$ can be either $\mathcal{O}(mp(n+q))$ or $\mathcal{O}(nq(m+p))$.

Explanation: There are two ways of computing $\boldsymbol{ABC}$.

1. **First compute $\boldsymbol{AB}$ as $\boldsymbol{D}$, and then compute $\boldsymbol{DC}$:** In this way, evaluating $\boldsymbol{D} = \boldsymbol{AB}$ needs $\mathcal{O}(mnp)$ time, evaluating $\boldsymbol{DC}$ needs $\mathcal{O}(mpq)$ time. Hence, the total time complexity is $\mathcal{O}(mp(n+q))$.
2. **First compute $\boldsymbol{BC}$ as $\boldsymbol{D}$, and then compute $\boldsymbol{AD}$:** In this way, evaluating $\boldsymbol{D} = \boldsymbol{BC}$ needs $\mathcal{O}(npq)$ time, evaluating $\boldsymbol{AD}$ needs $\mathcal{O}(mnq)$ time. Hence, the total time complexity is $\mathcal{O}(nq(m+p))$.

Thence, our claim holds. ■

Remark 1.16 In practical, we will always choose the time complexity of computing $\boldsymbol{ABC}$ to be $\min(\mathcal{O}(mp(n+q)), \mathcal{O}(nq(m+p)))$, because we can always force multiplication priority by adding parentheses in our implementations (e.g., $(\boldsymbol{AB})\boldsymbol{C}$ or $\boldsymbol{A}(\boldsymbol{BC})$). □

Exercise 1.17 (Orthogonal Procrustes Problem) Suppose we have two matrices $\boldsymbol{P} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{Q} \in \mathbb{R}^{n\times d}$. The optimization problem

$$\min_{\tilde{\boldsymbol{T}} \in \mathbb{R}^{m\times d}} \quad \left\| \boldsymbol{P}\tilde{\boldsymbol{T}} - \boldsymbol{Q} \right\|_F^2 \qquad \text{s.t.} \quad \tilde{\boldsymbol{T}}\tilde{\boldsymbol{T}}^\top = \boldsymbol{I}_m,$$

has an analytical solution $\tilde{\boldsymbol{T}} = \boldsymbol{U}\boldsymbol{I}_{m,d}\boldsymbol{V}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{V} \in \mathbb{R}^{d\times d}$ are left and right eigenvectors of $\boldsymbol{P}^\top\boldsymbol{Q}$, computed by singular value decomposition, respectively.

What is the computational complexity of solving the above optimization problem? (Hint: the computational complexity of SVD for an m by n matrix is $\mathcal{O}(mn^2 + m^2 n)$, as shown in the aforementioned Wikipedia page.)

References

[Cormen09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to algorithms," *MIT press*,

[Jordi] Jordi Cortadella, "Complexity Analysis of Algorithms", `https://www.cs.upc.edu/~jordicf/Teaching/FME/Informatica/pdf/Complexity.pdf`

[Vera] Vera Sacrist´an , "Complexity of recursive algorithms", `https://dccg.upc.edu/people/vera/wp-content/uploads/2013/06/recurrenciesEN.pdf`