MATH 333 Discrete Mathematics
Chapter 5 Algorithm Complexity

Jianan Lin

linj21@rpi.edu

May 24, 2022

Definition

An algorithm include three parts

1. Input data: some numbers, array, some structure such as graph, tree...

2. Output / Return value: the results we want.

3. Body: the main part of the algorithm which does what we want.

Example: Get the sum of an array $[1, 2, 3]$.

We can use either natural language, or math sentence to describe an algorithm.

Definition

Time Complexity: we use n to denote the input size and $f(n)$ be a function of n.

An algorithm is $O(f(n))$, if there exists some constant c, so that for any n, the running time $T \leq c \cdot f(n)$.

An algorithm is $\Omega(f(n))$, if there exists some constant c, so that for any n, the running time $T \geq c \cdot f(n)$.

An algorithm is $\Theta(f(n))$, if it is $O(f(n))$ and $\Omega(f(n))$.

Definition

Usually we only use $O(f(n))$ to describe the time complexity of an algorithm. There are three reasons. This is also allowed in homework and exam.

1. O is an English letter, which is easier to use compared with Ω and Θ.

2. Usually we only care about the upper bound of algorithm time complexity. If I say "I want an $O(n)$ algorithm", then $O(n)$ is enough. It does not matter whether we can find a faster algorithm.

3. Sometimes it is not easy to prove an algorithm is $\Theta(f(n))$ (because some problems are very hard). So we cannot ensure whether the upper bound is "tight". In this case we can only use $O(n)$.

Definition

Common time complexity. Here $\log(n) = \log_2(n)$ in time complexity.

Constant time: $O(1)$

Logarithmic time: $O(\log(n))$

Linear time: $O(n)$

Linearithmic time: $O(n\log(n))$

Polynomial time: $O(n^p)$

Exponential time: $O(2^n)$.

Definition

If an algorithm is $O(n)$, then of course it is also $O(n^2)$. But this is not allowed in homework and exam, because this does not provide enough information about the algorithm. You should write the tight upper bound.

If an algorithm is $O(\log(n))$, we can also say it is $O(\log_a(n))$ for other $a \neq 2$. In fact, they are totally the same because

$$\log_a(n) = \frac{\log(n)}{\log(a)}$$

Here, we know $\log(a)$ is a constant.

Definition

Usually, a good algorithm should satisfy:

1. Finish what we want the algorithm to do.

2. In polynomial time.

But this is not always that strict. For example, recall 3-SAT problem. Up to now, no one can find an algorithm satisfying the two conditions above. We have to satisfy only one condition.

An optimal algorithm is an efficient algorithm with the lowest time complexity. Also, there is no optimal algorithm in 3-SAT problem.

Definition

What will happen when the size n increases?

value of n	1	2	4	8	16	32	64
$O(1)$	1	1	1	1	1	1	1
$O(\log(n))$	0	1	2	3	4	5	6
$O(n)$	1	2	4	8	16	32	64
$O(n^2)$	1	4	16	64	256	1024	4096
$O(n^3)$	1	8	64	512	4096	32768	262144
$O(2^n)$	2	4	16	256	65536	4.3×10^9	1.8×10^{19}

We can see the importance of polynomial time!

Constant Algorithm

Given a binary positive integer, determine it is odd or even.

Algorithm: Consider the last digit of the integer. If it is 1, then odd, otherwise even.

Example: $32 = 100000$ is even. $15 = 1111$ is odd.

Also, determine whether an element is in a hash table is $O(1)$.

We can use this as a known theorem in homework and exam without proof.

Logarithmic Algorithm

Binary Search: Given an sorted array, determine whether a number belongs to it.

Algorithm: First choose the middle number. If it is larger than our target, then search the left part; if it is smaller than our target, then search the right part.

Example: find 2 from $[1, 2, 3, 4, 5, 6, 7, 8, 9]$.

Step 1: Middle number 5 is larger than 2, search it in $[1, 2, 3, 4]$.

Step 2: Middle number 2 (or 3) is equal to 2, we find it.

If we meet an empty array after some search, then we know this number does not belong to the original array.

Logarithmic Algorithm

How to calculate the time complexity of binary search?

Suppose n is the length of the array and $T(x)$ is the running time with data size x. Then we have

$$T(n) = T\left(\frac{n}{2}\right) + c$$

This is because when we search in a smaller array, the size becomes half of the previous one, and we only need to do a constant-time operation to determine, whether the target is larger than or smaller than the middle number. Therefore we have

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = ... = T(1) + c \cdot \log(n) = O(\log(n))$$

Logarithmic Algorithm

Calculate x^n: Exponential is not a basic operation. We have to use multiplication to get this result. The simplest algorithm to calculate $x, x^2, x^3, ..., x^n$ is not smart.

Idea: For convenience, we suppose $n = 2^k$.

If we want to have x^n, then we only need to calculate $x^{n/2}$ and let $x^n = x^{n/2} \cdot x^{n/2}$. In the same way, we calculate $x^{n/2}$ by calculating $x^{n/4}$......

Algorithm: First calculate x^2, then $x^4 = x^2 \cdot x^2$, until $x^n = x^{n/2} \cdot x^{n/2}$.

Time Complexity: $O(\log(n))$.

Linear Algorithm

Given a string in format: "{{}{}{}}", determine whether it is valid.

Example of invalid: "}{".

Algorithm: Maintain a stack. For the i-th char, if it is "{", then push it, else pop (if the element popped is not "{", then it's invalid). If at last the stack is empty, then valid.

A stack is an array. Push: $[1, 2, 3] \to [1, 2, 3, 4]$. Pop: $[1, 2, 3] \to [1, 2]$.

For the string "{{}{}{}}", the operations are: push, push, pop, push, pop, push, pop, pop. So it is valid.

Time Complexity: $O(n)$.

Sort Algorithm

Bubble Sort (Here we suppose the index is from 1 to n)

Algorithm 1 Bubble Sort(input = [1, 3, 5, 6, 2, 4])

```
n = input.length
for i ∈ [1, n − 1] do
    for j ∈ [i + 1, n] do
        if input[i] > input[j] then
            swap(input[i], input[j])
        end if
    end for
end for
return input
```

Time Complexity: $O(n^2)$

Sort Algorithm

Example of bubble sort: $[1, 3, 5, 6, 2, 4]$

$$
\begin{aligned}
[1, 3, 5, 6, 2, 4] &\to [1, 3, 5, 2, 6, 4] \\
&\to [1, 3, 5, 2, 6, 4] \\
&\to [1, 3, 5, 2, 4, 6] \\
&\to [1, 3, 2, 5, 4, 6] \\
&\to [1, 2, 3, 5, 4, 6] \\
&\to [1, 2, 3, 4, 5, 6]
\end{aligned}
$$

Sort Algorithm

Insert Sort (Here we suppose the index is from 1 to n)

Algorithm 2 Insert Sort(input = [1, 3, 5, 6, 2, 4])

```
n = input.length
for i ∈ [2, n] do
    for j ∈ [1, i − 1] do
        if input[i] < input[j] then
            insert input[i] in front of input[j]
            break
        end if
    end for
end for
return input
```

Time Complexity: $O(n^2)$

Sort Algorithm

Example of insert sort: $[1, 3, 5, 6, 2, 4]$

$$\begin{aligned}
[1, 3, 5, 6, 2, 4] &\rightarrow [1, 2, 3, 5, 6, 4] \\
&\rightarrow [1, 2, 3, 4, 5, 6]
\end{aligned}$$

Notice: There are several different implements for insert sort. So there are several versions for the code.

Sort Algorithm

Merge Sort: The first algorithm that breaks the $O(n^2)$ complexity.

Algorithm 3 Merge Sort(input = [1, 3, 5, 6, 2, 4])

```
n = input.length
if n == 1 then
    return input
else if n == 2 then
    return [min(input), max(input)]
else
    L1 = Merge Sort(input[1 : n/2])
    L2 = Merge Sort(input[n/2 + 1 : n])
    result = merge L1 and L2 in right order
end if
return result
```

Time Complexity.

$$\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + cn \\
&= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot \frac{1}{2}cn + cn \\
&= 8 \cdot T\left(\frac{n}{8}\right) + 4 \cdot \frac{1}{4}cn + 2cn \\
&= \ldots\ldots \\
&= nT(1) + cn\log(n) \\
&= O(n) + O(n\log(n)) \\
&= O(n\log(n))
\end{aligned}$$

Sort Algorithm

Example of merge sort: $[1,3,5,6,2,4]$

$$\begin{aligned}
[1,3,5,6,2,4] &\to [1,3,5] \oplus [6,2,4] \\
&\to ([1,3] \oplus [5]) \oplus ([6,2] \oplus [4]) \\
&\to ([1,3] \oplus [5]) \oplus ([2,6] \oplus [4]) \\
&\to [1,3,5] \oplus [2,4,6] \\
&\to [1,2,3,4,5,6]
\end{aligned}$$

Sort Algorithm

Quick Sort: Usually $O(n\log(n))$ but for worst case still $O(n^2)$.

Algorithm 4 Merge Sort(input = [1, 3, 5, 6, 2, 4])

n = input.length
if $n == 1$ **then**
    return input
else if $n == 2$ **then**
    return [min(input), max(input)]
else
    select a target number t (e.g., the first element)
    L = number in input if $x < t$
    M = number in input if $x = t$
    R = number in input if $x > t$
    result = Sort(L) + Sort(M) + Sort(R)
end if
return result

Sort Algorithm

Time Complexity. We consider a good case that: each target is the median number of the input, and there is no number equal to the target. So the time complexity is the same as merge sort.

$$
\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + cn \\
&= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot \frac{1}{2}cn + cn \\
&= 8 \cdot T\left(\frac{n}{8}\right) + 4 \cdot \frac{1}{4}cn + 2cn \\
&= ...... \\
&= nT(1) + cn\log(n) \\
&= O(n) + O(n\log(n)) \\
&= O(n\log(n))
\end{aligned}
$$

Sort Algorithm

Example of quick sort: $[1, 3, 5, 6, 2, 4]$. We select the average of the first and last number of input as the target

$$
\begin{aligned}
[1, 3, 5, 6, 2, 4] &\to [1, 2] \oplus [3, 5, 6, 4] \\
&\to [1, 2] \oplus ([3] \oplus [5, 6, 4]) \\
&\to [1, 2] \oplus ([3] \oplus ([4] \oplus [5, 6])) \\
&\to [1, 2] \oplus ([3] \oplus [4, 5, 6]) \\
&\to [1, 2] \oplus [3, 4, 5, 6] \\
&\to [1, 2, 3, 4, 5, 6]
\end{aligned}
$$

Matrix Multiplication Algorithm

For convenience, we consider two $n \times n$ matrices A and B. Suppose $C = A \cdot B$ and obviously C is $n \times n$. So we know

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

We can design a simple algorithm according to this formula. And the time complexity is $O(n^3)$ because there are n^2 elements in a matrix and it takes $O(n)$ time to calculate an element in C.

However, can we have a faster algorithm in matrix multiplication?

The answer is yes!

Matrix Multiplication Algorithm

Strassen Algorithm: $O(n^{2.8})$. But this is only useful when $n > 300$.

Step 1: Decompose the matrix to get the following small matrix in $O(1)$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

So we know

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

Here we will use 8 multiplication, which will not improve. But we can consider some method to use only 7 multiplication.

Matrix Multiplication Algorithm

Step 2: We have the following variables in $O(n^2)$.

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

Matrix Multiplication Algorithm

Step 3: We have the following variables with 7 multiplications.

$$P_1 = A_{11} \cdot S_1$$
$$P_2 = S_2 \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6$$
$$P_6 = S_7 \cdot S_8$$
$$P_7 = S_9 \cdot S_{10}$$

Matrix Multiplication Algorithm

Step 4: We get C by calculate C_{ij} in $O(n^2)$.

$$\begin{aligned}
C_{11} &= P_5 + P_4 + P_6 - P_2 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_1 + P_5 - P_3 - P_7
\end{aligned}$$

Therefore we can write the time complexity formula:

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + c \cdot n^2$$

Then we solve this and get the time complexity.

Matrix Multiplication Algorithm

$$
\begin{aligned}
T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + c \cdot n^2 \\
&= 7^2 \cdot T\left(\frac{n}{4}\right) + \left(1 + \frac{7}{4}\right) cn^2 \\
&= 7^3 \cdot T\left(\frac{n}{2^3}\right) + \left(1 + \frac{7}{4} + \frac{7^2}{4^2}\right) cn^2 \\
&= \ldots\ldots \\
&= 7^{\log(n)} \cdot T(1) + \frac{(7/4)^{\log(n)} - 1}{7/4 - 1} cn^2 \\
&= O\left(7^{\log(n)}\right) = O\left(2^{7\log(n)}\right) \\
&= O\left(n^{\log(7)}\right) \qquad \text{Recall that } \log(a^x) = x\log(a) \\
&\approx O\left(n^{2.8}\right)
\end{aligned}
$$

Summary

Suppose

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$$

We have

$$T(n) = \begin{cases} O\left(n^d\right) & a < b^d \\ O\left(n^d \log(n)\right) & a = b^d \\ O\left(n^{\log_a(b)}\right) & a > b^d \end{cases}$$

Do not need to remember. We only need to know how to calculate easy time complexity such as binary search $O(\log(n))$ and merge sort $O(n \log(n))$.

That means, always suppose $a = 1, 2$, $b = 2$ and $d = 0, 1, 2...$

P and NP

P problem: We can solve it in polynomial time, i.e. there exists polynomial algorithm.

NP problem: We can test the result in polynomial time.

Obviously a P problem must belong to NP problem. If we can find a result in polynomial time, then we can also test it in polynomial time.

Here, NP is "polynomial in nondeterministic Turing machine", not "non-polynomial".

In the same way, P is "polynomial in deterministic Turing machine".

P and NP

P = NP ? This is a famous question. If P = NP, then "If a problem can be tested in polynomial time, then it can be solved in polynomial time".

NP Complete: The hardest problem set in NP.

If we find polynomial algorithm for a NPC problem, then we prove P = NP. If we want to prove P != NP, then we must prove a NPC problem does not have polynomial algorithm.

But most of the scientists believe P != NP. This is very hard (even impossible) to prove. After all, to find something is always easier than to prove something does not exist!

NP hard: problem at least as hard as NPC. Some NP hard problem does not have polynomial algorithm. NP hard does not belong to NP.

P and NP

The first NPC problem SAT: Can we find variables to let the sentence be true?

$$(p \vee q \vee ...) \wedge (\neg p \vee \neg q \vee ...) \wedge (p \vee \neg q \vee ...) \wedge ... \wedge (\neg p \vee q \vee ...)$$

Stephan Cook proved that this is NPC, i.e. the hardest problem (not unique) in NP.

If we can transfer a problem A to problem B in polynomial time, then we say A is at least as hard as B. So we can transfer any NP problem to SAT in polynomial time.

Scientists have found many NPC problems by transferring this problem to a known NPC problem in polynomial time.