

Algorithm Complexity

Tutorial 1

Exercise 1

Consider a modern computer capable of executing approximately 10^{15} basic instructions per second. An algorithm operating on an input of size n performs a number of instructions given by one of the following functions:

$$f(n) \in \{\log_2 n, n, n \log_2 n, n^2, n^3, 2^n\}.$$

- For each function $f(n)$ and each input size $n \in \{6, 25, 50, 100\}$, calculate the number of instructions performed by the algorithm and the corresponding execution time.
- Express the execution times using an appropriate time unit and complete a table summarizing the results.
- For each input size, arrange the algorithms from the fastest to the slowest according to their execution time.
- Arrange the six functions according to their asymptotic growth rate, from the slowest-growing to the fastest-growing function.
- Consider an algorithm with complexity 2^n and $n = 100$. Suppose that the computer used in the previous questions is replaced by a computer that is 10 000 times faster. Calculate the new execution time and compare it with the original execution time.
- Moore's law is commonly expressed as an approximate doubling of computing capacity every two years. Suppose that a computer in 2005 could execute 40×10^{12} instructions per second.
 - Estimate the computing capacity of this computer in 2025, assuming that the computing capacity doubles every two years.
 - Using the estimated 2025 computing capacity, calculate the execution time of an algorithm performing 2^{100} instructions.
 - Express the resulting execution time in years and discuss whether the improvement predicted by Moore's law makes the computation practically feasible.
- Based on the previous results, explain why improvements in computing capacity cannot compensate indefinitely for algorithms with rapidly growing complexity.

Additional Question

Consider two algorithms solving the same problem:

$$f_A(n) = n^4 \quad \text{and} \quad f_B(n) = 2^n.$$

For small input sizes, their execution times may appear comparable. Explain what happens as n increases and discuss which algorithm is more suitable for large-scale inputs.

Solution

The computer can execute approximately

$$10^{15}$$

basic instructions per second. Therefore, the execution time for an algorithm performing $f(n)$ instructions is

$$T(n) = \frac{f(n)}{10^{15}} \text{ seconds.}$$

1. Number of instructions and execution time

For the input sizes $n = 6, 25, 50, 100$, we obtain the following results.

$f(n)$	$n = 6$	$n = 25$	$n = 50$	$n = 100$
$\log_2 n$	2.585	4.644	5.644	6.644
n	6	25	50	100
$n \log_2 n$	15.510	116.096	282.193	664.386
n^2	36	625	2500	10000
n^3	216	15625	125000	1000000
2^n	64	33,554,432	1.1259×10^{15}	1.2677×10^{30}

Since the computer performs 10^{15} instructions per second,

$$T(n) = \frac{f(n)}{10^{15}}.$$

For example, for $f(n) = n^3$ and $n = 100$:

$$T(100) = \frac{100^3}{10^{15}} = \frac{10^6}{10^{15}} = 10^{-9} \text{ seconds.}$$

For $f(n) = 2^n$ and $n = 100$:

$$T(100) = \frac{2^{100}}{10^{15}} \approx 1.2677 \times 10^{15} \text{ seconds.}$$

Converting this value into years:

$$T(100) = \frac{1.2677 \times 10^{15}}{365.25 \times 24 \times 3600} \approx 4.02 \times 10^7 \text{ years.}$$

Thus, the execution times are:

$f(n)$	$n = 6$	$n = 25$	$n = 50$	$n = 100$
$\log_2 n$	$2.58 \times 10^{-15} s$	$4.64 \times 10^{-15} s$	$5.64 \times 10^{-15} s$	$6.64 \times 10^{-15} s$
n	$6 \times 10^{-15} s$	$2.5 \times 10^{-14} s$	$5 \times 10^{-14} s$	$10^{-13} s$
$n \log_2 n$	$1.55 \times 10^{-14} s$	$1.16 \times 10^{-13} s$	$2.82 \times 10^{-13} s$	$6.64 \times 10^{-13} s$
n^2	$3.6 \times 10^{-14} s$	$6.25 \times 10^{-13} s$	$2.5 \times 10^{-12} s$	$10^{-11} s$
n^3	$2.16 \times 10^{-13} s$	$1.5625 \times 10^{-11} s$	$1.25 \times 10^{-10} s$	$10^{-9} s$
2^n	$6.4 \times 10^{-14} s$	$3.355 \times 10^{-8} s$	$1.126 s$	$4.02 \times 10^7 \text{ years}$

2. Ranking the algorithms

For every sufficiently large n , the functions can be ordered according to their growth rates as follows:

$$\log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n$$

Therefore, for large input sizes, the corresponding algorithms are expected to execute from fastest to slowest in this same order.

An interesting observation is that for $n = 6$, the difference between some algorithms is relatively small. However, as n increases, the differences become much more significant.

For example, for $n = 100$:

$$n^3 = 10^6,$$

whereas

$$2^{100} \approx 1.27 \times 10^{30}.$$

Thus, the exponential function grows enormously faster than the polynomial functions.

3. Increasing the computing capacity by 10,000 times

The original computer performs

$$10^{15}$$

instructions per second.

A computer that is 1,000 times faster performs

$$10^{15} \times 10^4 = 10^{19}$$

instructions per second.

For $f(n) = 2^n$ and $n = 100$:

$$T(100) = \frac{2^{100}}{10^{19}}.$$

Since

$$2^{100} \approx 1.2677 \times 10^{30},$$

we obtain

$$T(100) \approx 1.2677 \times 10^{11} \text{ seconds.}$$

Converting to years:

$$T(100) \approx \frac{1.2677 \times 10^{11}}{31,557,600} \approx 4.02 \times 10^3 \text{ years.}$$

Therefore,

$$T(100) \approx 4,020 \text{ years}$$

even with a computer that is 10,000 times faster.

This shows that a fixed improvement in hardware performance does not make an exponential computation practically feasible.

4. Moore's Law

According to the assumption in the exercise, computing capacity doubles every two years.

The initial computing capacity in 2005 is

$$40 \times 10^{12}$$

instructions per second.

The period from 2005 to 2025 is

$$2025 - 2005 = 20 \text{ years.}$$

Therefore, the number of doubling periods is

$$\frac{20}{2} = 10.$$

The improvement factor is consequently

$$2^{10} = 1024.$$

Hence, the estimated computing capacity in 2025 is

$$C_{2025} = 40 \times 10^{12} \times 2^{10}.$$

Thus,

$$C_{2025} = 40 \times 10^{12} \times 1024 = 4.096 \times 10^{16}$$

instructions per second.

Therefore,

$$C_{2025} \approx 4.096 \times 10^{16} \text{ instructions/second}$$

under the assumptions of the exercise.

5. Execution time for 2^{100} instructions in 2025

The algorithm performs

$$2^{100} \approx 1.2677 \times 10^{30}$$

instructions.

Using the estimated 2025 computing capacity:

$$T_{2025} = \frac{2^{100}}{4.096 \times 10^{16}}.$$

Therefore,

$$T_{2025} \approx 3.095 \times 10^{13} \text{ seconds.}$$

Converting into years:

$$T_{2025} = \frac{3.095 \times 10^{13}}{365.25 \times 24 \times 3600}.$$

Hence,

$$T_{2025} \approx 9.81 \times 10^5 \text{ years}$$

or approximately

$$981,000 \text{ years}.$$

Thus, even after twenty years of computing-capacity improvement according to the assumed Moore's law model, the computation involving 2^{100} instructions remains completely impractical.

6. Why hardware improvement cannot compensate indefinitely

Suppose computing capacity is multiplied by a constant factor k . The execution time becomes

$$T'(n) = \frac{f(n)}{kC}.$$

Therefore, increasing computing capacity by a fixed factor just divides the execution time by that same factor.

For an exponential algorithm,

$$f(n) = 2^n,$$

the number of instructions doubles whenever n increases by just one:

$$2^{n+1} = 2 \cdot 2^n.$$

Consequently, increasing the input size produces an extremely rapid increase in execution time.

For example,

$$2^{100} \approx 1.27 \times 10^{30},$$

while

$$2^{200} \approx 1.61 \times 10^{60}.$$

The latter is approximately

$$2^{100}$$

times larger than the former.

Thus, hardware improvements alone cannot overcome the rapid growth associated with exponential algorithms.

7. Discussion: n^4 versus 2^n

Consider

$$f_A(n) = n^4$$

and

$$f_B(n) = 2^n.$$

For small values of n , the two functions can have relatively close values. For example,

$$6^4 = 1,296 \text{ while } 2^6 = 64.$$

$$\text{and } 10^4 = 10,000 \text{ while } 2^{10} = 1,024.$$

However, at $n = 20$:

$20^4 = 160,000$,

whereas

$2^{20} = 1,048,576$.

At $n = 50$:

$50^4 = 6,250,000$,

whereas

$2^{50} \approx 1.126 \times 10^{15}$.

The difference becomes enormous as n increases.

Therefore,

$$n^4 < 2^n \quad \text{for sufficiently large } n$$

and the polynomial-time algorithm becomes much more suitable for large input sizes.

Conclusion

The exercise illustrates a fundamental principle of algorithmic complexity: increasing computing capacity can reduce execution time, but it does not change the growth rate of an algorithm. An exponential algorithm such as 2^n can remain impractical even on extremely powerful computers, whereas polynomial algorithms such as n , $n \log n$, n^2 , n^3 , and n^4 generally scale much better as the input size increases.

Moore's law can provide substantial improvements in hardware performance, but these improvements are insufficient to overcome the exponential growth associated with computationally expensive algorithms.

Exercise 2

The Fibonacci sequence is defined by:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Consider the following approaches for computing the n -th Fibonacci number:

1. A direct recursive approach based on the mathematical definition.
2. A dynamic programming approach using memoization (top-down).
3. A dynamic programming approach using tabulation (bottom-up).
1. Write the pseudocode for the direct recursive algorithm.
2. Write a pseudocode algorithm using the top-down dynamic programming approach with memoization.
3. Write a pseudocode algorithm using the bottom-up dynamic programming approach with tabulation.
4. For $n = 6$, illustrate the recursive calls generated by the direct recursive algorithm. Identify the subproblems that are solved more than once.
5. For each of the approaches, determine:
 - (a) the time complexity;
 - (b) the space complexity.
6. Explain how memoization and tabulation avoid the repeated computation observed in the direct recursive approach.
7. Calculate F_{10} and F_{20} using the bottom-up algorithm.

Additional Questions

- The Fibonacci number F_n grows exponentially with n , although the dynamic programming algorithm computes it in linear time. Explain why the growth of the output value does not imply that the algorithm has exponential time complexity.
- The bottom-up algorithm stores all Fibonacci values from F_0 to F_n . Propose a modified version that computes F_n using just a constant amount of additional memory. Give its time and space complexities.

Solution

The Fibonacci sequence is defined by

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

The objective is to compare a direct recursive solution with two dynamic programming approaches: memoization and tabulation.

1. Direct recursive algorithm

The direct recursive implementation follows directly from the mathematical definition.

```

1 if n = 0 or n = 1 then
2   return n;
3 end
4 return FIBRECURSIVE(n - 1) + FIBRECURSIVE(n - 2);

```

Algorithm 1: FIBRECURSIVE(n)

For example:

$$F_5 = F_4 + F_3.$$

The algorithm recursively computes both F_4 and F_3 , and these computations themselves generate several identical subproblems.

2. Top-down dynamic programming: memoization

Memoization stores the result of each already computed subproblem. When the same subproblem is encountered another time, its stored value is returned instead of computing it another time.

```

1 if F[n] ≠ null then
2   return F[n];
3 end
4 if n = 0 or n = 1 then
5   F[n] ← n;
6 end
7 else
8   F[n] ← FIBTOPDOWN(n - 1, F) + FIBTOPDOWN(n - 2, F);
9 end
10 return F[n];

```

Algorithm 2: FIBTOPDOWN(n, F)

Initially, all elements of $F[0..n]$ are set to **null**.

For example, after computing F_5 , the array contains:

i	0	1	2	3	4	5
$F[i]$	0	1	1	2	3	5

Thus, if F_2 is needed another time, the algorithm directly returns the stored value 1.

3. Bottom-up dynamic programming: tabulation

The bottom-up approach starts with the two smallest subproblems and progressively computes larger Fibonacci numbers.

```

1 if n = 0 or n = 1 then
2   return n;
3 end
4 Create an array F[0..n]; F[0] ← 0; F[1] ← 1;
5 for i ← 2 to n do
6   F[i] ← F[i - 1] + F[i - 2];
7 end
8 return F[n];

```

Algorithm 3: FIBBOTTOMUP(n)

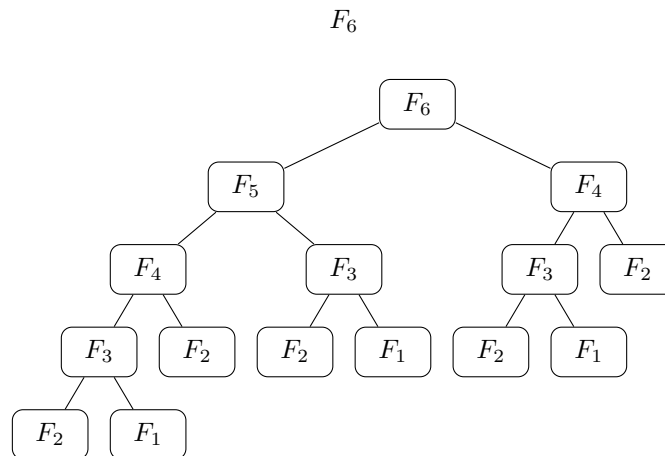
For $n = 6$, the algorithm computes:

Therefore,

$$F_6 = 8.$$

4. Repeated subproblems for $n = 6$

The direct recursive algorithm generates the following recursive structure:



Expanding the tree completely gives:

$$\begin{aligned} F_6 &= F_5 + F_4 \\ &= (F_4 + F_3) + F_4 \\ &= ((F_3 + F_2) + (F_2 + F_1)) + (F_3 + F_2). \end{aligned}$$

Therefore, several subproblems are evaluated multiple times.

For F_6 , the number of occurrences of the different subproblems is:

Subproblem	Number of occurrences
F_6	1
F_5	1
F_4	2
F_3	3
F_2	5
F_1	8
F_0	5

For example, F_4 is calculated twice, F_3 three times, and F_2 five times.

This repeated computation is the main source of inefficiency in the direct recursive algorithm.

5. Time and space complexity

Direct recursive approach

For $n > 1$, the algorithm performs two recursive calls:

$$T(n) = T(n-1) + T(n-2) + O(1).$$

This recurrence has exponential growth. More precisely,

$$T(n) = O(\lambda^n),$$

where

$$\lambda = \frac{1 + \sqrt{5}}{2} \approx 1.618.$$

It is commonly simplified to

$$T(n) = O(2^n).$$

The maximum depth of the recursive call stack is n , giving

$$S(n) = O(n).$$

Top-down dynamic programming

With memoization, each Fibonacci value from F_0 to F_n is computed at most once.

There are $n + 1$ subproblems and each computation performs a constant number of operations. Therefore,

$$T(n) = O(n).$$

The memoization array contains $n + 1$ values, and the recursive call stack can have depth n :

$$S(n) = O(n).$$

Bottom-up dynamic programming

The algorithm performs one iteration for each value from 2 to n :

$$[T(n) = O(n)].$$

The array contains $n + 1$ elements:

$$S(n) = O(n).$$

Consequently, the approaches can be summarized as:

Approach	Time Complexity	Space Complexity
Direct recursion	$O(2^n)$	$O(n)$
Memoization	$O(n)$	$O(n)$
Tabulation	$O(n)$	$O(n)$

6. How dynamic programming eliminates repeated computation

The direct recursive algorithm does not remember the results of previously solved subproblems. Consequently, the same Fibonacci numbers are calculated many times.

For example, in the computation of F_6 , the value F_4 is calculated twice:

$$F_6 = F_5 + F_4$$

and

$$F_5 = F_4 + F_3.$$

Therefore, F_4 appears in two different branches of the recursive computation.

Dynamic programming eliminates this redundancy by storing previously calculated results.

In the **top-down approach**, a result is stored when its recursive computation finishes. A later access for the same value retrieves the stored result.

In the **bottom-up approach**, each value is calculated exactly once, starting from F_0 and F_1 and progressing toward F_n .

Thus, the exponential computation is reduced into a linear-time computation:

$$O(2^n) \longrightarrow O(n).$$

7. Calculation of F_{10} and F_{20}

Using the bottom-up approach:

n	0	1	2	3	4	5	6	7	8	9	10
F_n	0	1	1	2	3	5	8	13	21	34	55

Therefore,

$$\boxed{F_{10} = 55}.$$

Continuing the same process:

n	10	11	12	13	14	15	16	17	18	19	20
F_n	55	89	144	233	377	610	987	1597	2584	4181	6765

Hence,

$$\boxed{F_{20} = 6765}.$$

8. Exponential growth of the Fibonacci numbers versus algorithm complexity

The Fibonacci numbers themselves grow exponentially:

$$F_n = \frac{\lambda^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}},$$

and therefore

$$F_n = \Theta(\lambda^n).$$

However, the size or value of the output does not determine the number of operations performed by an algorithm.

The bottom-up algorithm uses the recurrence

$$F_i = F_{i-1} + F_{i-2}$$

and performs this calculation once for each

$$i = 2, 3, \dots, n.$$

Consequently, it performs just $n - 1$ iterations, giving

$$\boxed{T(n) = O(n)}.$$

Thus, although the numerical value of F_n grows exponentially, the number of arithmetic operations performed by the dynamic programming algorithm grows linearly with n .

This distinction is important in complexity analysis:

$$\boxed{\text{Growth of the output value} \neq \text{growth of the running time.}}$$

9. Additional question: constant-space implementation

The bottom-up algorithm does not actually need to store the complete Fibonacci values. To calculate the next value, just the two preceding values are needed.

A modified algorithm can therefore use two variables:

```

1 if  $n = 0$  then
2   | return 0;
3 end
4  $a \leftarrow 0$ ;  $b \leftarrow 1$ ;
5 for  $i \leftarrow 2$  to  $n$  do
6   |  $c \leftarrow a + b$ ;  $a \leftarrow b$ ;  $b \leftarrow c$ ;
7 end
8 return  $b$ ;
```

Algorithm 4: FIBOPTIMIZED(n)

For example, for $n = 6$:

i	a	b	c
—	0	1	—
2	1	1	1
3	1	2	2
4	2	3	3
5	3	5	5
6	5	8	8

Therefore,

$$[F_6 = 8]$$

The algorithm still performs one iteration for each value of i :

$$[T(n) = O(n)]$$

However, just a constant number of variables are stored:

$$[S(n) = O(1)]$$

Therefore, the approaches can finally be compared as follows:

Approach	Time	Space
Direct recursion	$O(2^n)$	$O(n)$
Memoization	$O(n)$	$O(n)$
Tabulation	$O(n)$	$O(n)$
Optimized tabulation	$O(n)$	$O(1)$

The comparison illustrates the main benefit of dynamic programming: **storing or reusing solutions to previously solved subproblems can reduce an exponential-time recursive algorithm into a linear-time algorithm.**