\[ f(n) = O(g(n)) \]

A graphical view of big-O notation
method

The method involves determining the number of operations executed in each loop.

- Loop: number of loop iterations x number of operations inside loop
- Inequality side expression is determined separately
- Assumption: count as 1 operation (operation count of basic operation + arithmetic/logical operation counts)
- Detailed counting: estimate number of basic operation counts

In general, the number of operations executed in a single loop is

\[
\text{operations per loop} = \text{loop iterations} \times \text{operations per iteration}
\]

For example, if there are 10 loop iterations and each iteration executes 5 operations, then

\[
\text{operations per loop} = 10 \times 5 = 50
\]

Thus, the total number of operations executed in the loop is

\[
\sum_{i=1}^{10} 5 = 50
\]

For a program with multiple loops, the total number of operations is the sum of the operations in each loop.

Symbolic complexity

Suppose we have a computer that can execute 1000 instructions per second. This computer can solve a problem in a certain amount of time.

\[
\text{time to solve} = \frac{\text{number of operations}}{1000}\text{instructions/second}
\]

In general, the time to solve a problem depends on the number of operations required.

For example, if a program requires 10,000 operations, then it will take

\[
\frac{10000}{1000} = 10\text{ seconds}
\]

Therefore, the time to solve a problem is determined by the number of operations required and the computer's speed.
When differentiating, estimate running time for recursive programs.

1. Implement loops (while, for).

2. Once you get the basic form of this, you can quickly zero in on what your estimate of the time and space obtained the same thing.

For asymptotic running time, we do not need to count please.

4. What's the cost of each.[2, etc.

How do we solve the recurrence equation?

\[
T(n) = \begin{cases} 

c_0 & \text{if } n = 1 \\
T(n/2) + T(n/4) + \frac{\log n}{\log 2} & \text{otherwise}
\end{cases}
\]

Example: section sort.

\[
O = \begin{cases} 

c_0 & \text{if } n = \log n \\
T(n) + \frac{\log n}{\log 2} & \text{otherwise}
\end{cases}
\]

Matrix multiplication.
<table>
<thead>
<tr>
<th>uitive search</th>
<th>(n) = O(\log(n))</th>
<th>(T(n) = 2T(n/2) + \Theta(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>(n) = (O(n \log n))</td>
<td>(T(n) = \Theta(n \log n))</td>
</tr>
<tr>
<td>Linear search</td>
<td>(n) = (\Theta(n))</td>
<td>(T(n) = \Theta(n))</td>
</tr>
</tbody>
</table>

**Example**

**Closed-form expressions**

\[
\begin{align*}
(c(n) &= 0) \\
\ \text{if } n &= 2^n \\
\end{align*}
\]

**Recurrence relation**

\[
T(1) = \Theta(1) \\
T(n) = \Theta(n \log n) \\
\]

**Simplify by dropping lower-order terms**

\[
T(n) = \Theta(n \log n) \\
\]

**Remarks**

**Practice:**

- Prove via induction that \(T(n) = \Theta(n \log n)\).
- For CS 211, learn and practice recurrence relations.

**Information:**

- No general techniques known for solving recurrences (like Master Theorem).
can guarantee that worst-case behavior will not show up.

Often appears to exceed worst-case behavior. Pick pivot carefully so it

which in theory is usually predicted in practice.

On the average (not worst-case), quick-sort runs in $O(n \log n)$ time.

It can be shown that \( O(n) = \frac{(n)}{(1)} \)

\( (n-1) \) \( (n) \) \( (n-2) \) \( (n-3) \)

So actual recurrence relation is

the other has \( (n-1) \) elements

Worst-case for quicksort: one of the partitioned array is empty, and

Remember: Big-O is worst-case complexity.

\begin{align*}
\text{partition} \quad \text{sort the two partitioned arrays} \\
\end{align*}

\begin{align*}
\text{--} \\
(0) + \quad (c(n)) \quad (n) \\
1 \quad (c(n)) \quad (n-1) \\
(0) + \quad (c(n)) \quad (n) \\
1 \quad (c(n)) \quad (n-1) \\
\end{align*}

\begin{align*}
\text{make recursive calls} \\
\text{more steps into this linear recursive phase} \\
\text{find p = partition}(a', i' + 1 , a'[j]) \\
\text{public static void quicksort(comparable a, int i, int j) } \\
\text{and why is quicksort tricky} \end{align*}
fib(5) = fib(4) + fib(3) + 2
fib(4) = fib(3) + fib(2) + 2
fib(3) = fib(2) + fib(1)
fib(2) = fib(1)

$c(n) = c(n-1) + c(n-2) + 2$
$c(2) = 1$ $c(1) = 1$

Iterative Fibonacci Code:

dad = 1
granddad = 1
current = 1;

for (i = 3; i <= n; i++) {
    granddad = dad;
    dad = current;
    current = dad + granddad;
}

printf("answer is ", current);

Number of times loop is executed is bounded by n.
Each iteration does some constant amount of work.
=> Time complexity of algorithm = \(\mathcal{O}(n)\).

Each iteration does some constant amount of work.
Number of times loop is executed is bounded by n.

int main() {
  int n = 6;
  int dad = 1;
  int granddad = 1;
  int current = 1;

  for (int i = 3; i <= n; i++) {
    int temp = granddad;
    granddad = dad;
    dad = current;
    current = temp + dad;
  }

  printf("answer is ", current);
  return 0;
}