Algorithm: A clearly specified set of instructions the computer will follow to solve a problem.

Given an algorithm, we want to determine the amount of memory it uses, and how much time it requires to solve a problem.

---

Maximum contiguous subsequence sum

Given an array with integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$.

4, -3, 5, -2, -1, 2, 6, -2

How many possible subsequences are there?

Number of additions: $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$

---

The naive algorithm

```python
maxSum = 0
for i in range(len(a)):
    for j in range(i, len(a)):
        sum = 0
        for k in range(i, j+1):
            sum += a[k]
        if sum > maxSum:
            maxSum = sum
```

---

Etymology of “Algorithm”

Algorism = process of doing arithmetic using Arabic numerals.

A misperception: algiros [painful] + arithmos [number].

True origin: Abū 'Abdallāh Muhammad ibn Mūsā al-Khwārizmī was a 9th-century Persian mathematician, astronomer, and geographer, who wrote Kitab al-jabr wa’l-muqabala (Rules of restoring and equating), which evolved into today’s high school mathematics text.
maxSum = 0
for i in range(len(a)):
    sum = 0
    for j in range(i, len(a)):
        sum += a[j]
        if sum > maxSum:
            maxSum = sum

Number of additions: $\sum_{i=0}^{n-1} (n - i)$

### Divide and Conquer (Divide et impera)

- Split the problem into subproblems.
- Solve the subproblems recursively.
- Combine the solutions to the subproblems.

### Experimental analysis of algorithms

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `time.perf_counter()` to get a measure of the actual running time

<table>
<thead>
<tr>
<th>n</th>
<th>Naive</th>
<th>Faster</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>760</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>1,000</td>
<td>652,285</td>
<td>2,411</td>
<td>236</td>
</tr>
<tr>
<td>10,000</td>
<td>–</td>
<td>218,210</td>
<td>2,378</td>
</tr>
<tr>
<td>100,000</td>
<td>– 23,033,000</td>
<td>25,037</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>–</td>
<td>–</td>
<td>260,375</td>
</tr>
</tbody>
</table>
Limitations:

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used. Programming language, programming style, fine tuning should not be measured.

Theoretical analysis counts primitive operations.

Primitive operations are:

- Assigning a value to a variable
- Calling a method
- Arithmetic operations (e.g., adding two numbers)
- Indexing into an array
- Following a reference
- Returning from a method

While we do not know the exact cost of a primitive operation (it depends on the processor speed, processor architecture, programming language, compiler, etc.), we know that all primitive operations take constant time.

There is a fixed, finite number of primitive operations.

Let $a$ be the time taken by the fastest primitive operation, let $b$ be the time taken by the slowest primitive operation.

If our algorithm uses $k$ primitive operations, then its running time $T(n)$ is bounded by

$$ak \leq T(n) \leq bk$$

We are more interested in the growth rate of the running time than in the exact formula. A quadratic algorithm will always be faster than a cubic algorithm if the input size is sufficiently large.

The growth rate determines the scaling behavior of an algorithm: If we increase the problem size by a factor 10, how much does the running time increase?

Or, put differently: If we buy a computer that is ten times faster, how much larger problems can we solve?
Since we only want to know the growth rate of an algorithm, we can simplify the analysis using Big-Oh notation.

**Definition of Big-Oh:**
Let $f(n)$, $g(n)$ be functions from $\{1, 2, 3, 4, \ldots\}$ to $\mathbb{R}$. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer $n_0 \geq 1$ such that
\[ f(n) \leq cg(n) \quad \text{for } n \geq n_0. \]

- $4n + 1$ is $O(n)$
- $2n^2 + 3n + 5$ is not $O(n)$
- $2n^2 + 3n + 5$ is $O(n^2)$

**Asymptotic analysis**
The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
To perform the asymptotic analysis
- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big-Oh notation
Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

A word of caution:
What is better, $10^{100}n$, $n^{100}$, or $2^n$?

We want to express the running time in the simplest possible Big-Oh notation.

- $4n \log n + 3n - 12$ is $O(n \log n + 3n)$ is correct, but we should say that it is $O(n \log n)$.

Any polynomial
\[ f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \cdots + a_dn^d \]
with $a_d > 0$ is just $O(n^d)$.

- $5n^2 + 3n \log n + 2n + 5$ is $O(n^2)$
- $20n^3 + 10n \log n + 5$ is $O(n^3)$
- $3 \log n + 2$ is $O(\log n)$
- $2^n + 2$ is $O(2^n)$
- $2n + 100 \log n$ is $O(n)$

**Keeping your perspective**
- Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency
- When choosing an ADT implementation, consider how frequently particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient
- If the problem size is always small, you can probably ignore an algorithm’s efficiency
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements