In [1]:

- Comments beginwith **//** and continue until the end of line.

- Blocks are indicated with matching braces: **{** and **}**. A compound statement (i.e., a collection of simple statements) can be represented as a block.The body of a procedure also forms a block. Statements are delimited by **;** .

- An identifier begins with a letter.The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with records. Here is an example

$$cnode = rec \text{ or } d\{datatype_1 data_1; \ldots datatype_n data_n; node \cdot l \in k;\}$$

- In the above mentioned example, link is a pointer to the record type node. Individual data items of a record can be accessed with $\rightarrow$ and period. For instance if p pointsto a record of type node, $p \rightarrow$data_1 stands for the value of the first field in the record. On the other hand, if q is a record of type node, q.data_1 will denote its first field.

- Assignment of values to variables is done using the assignment statement

$$c(variab \leq) := (\exp ression);$$

- There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<, \leq, =, \neq, \geq, \text{and} >$ are provided.

- Elements of multidimensional arrays are accessed using $[$ and $]$. For example, if A is a two dimensional array, the $(i, j)^{th}$ element of the array is denoted as - $A[i, j]$. Array indices start at zero.

- The following looping statements are employed: **for**, **while**, and **repeat-until**.

`c while (condition) do { . . . } `

`c for variable := valuel to value2 step step do { . . . } `

`c repeat . . . until (condition) `

In [ ]:

```
from random import randint

min_val = 0
max_val = 100

random_num = randint(min_val, max_val)
attempts = 0

for i in range(min_val, _max_val):
    if i == random_num:
        print("The number was: " + str(random_num))
        break
    attempts += 1
print("The number of guesses was: " + str(attempts))
```

In [ ]:

```
from random import randint

min_val = 0
max_val = 100
random_num = randint(min_val, max_val)

attempts = 0
while True:
    guess = round((min_val + max_val) / 2)
    if guess == randnum:
        print("The number was: " + str(random_num))
        break
    elif guess > randnum:
        print("Too big: " + str(guess))
        max_val = guess
    elif guess < randnum:
        print("Too small: " + str(guess))
        min_val = guess
    attempts += 1
print("The number of guesses was: " + str(attempts))
```

In [1]:

```
def fact(n):
    product = 1
    for i in range(n):
        product = product * (i+1)
    return product

print(fact(5))
```

```
120
```

In [2]:

```
def fact2(n):
    if n == 0:
        return 1
    else:
        return n * fact2(n-1)

print(fact2(5))
```

```
120
```

In [3]:

```
%timeit fact(50)
```

```
5.56 µs ± 190 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [4]:

```
%timeit fact2(50)
```

```
10.2 µs ± 178 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

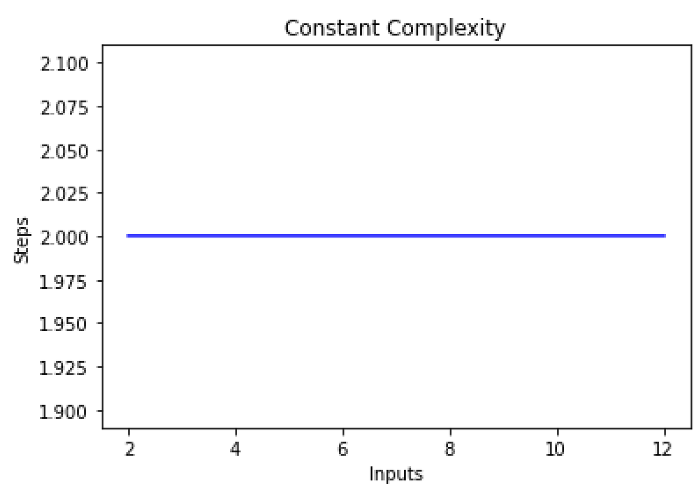
In [5]:

```
import matplotlib.pyplot as plt
import numpy as np

x = [2, 4, 6, 8, 10, 12]

y = [2, 2, 2, 2, 2, 2]

plt.plot(x, y, 'b')
plt.xlabel('Inputs')
plt.ylabel('Steps')
plt.title('Constant Complexity')
plt.show()
```



Best case — being represented as Big Omega – Ω(n) Big-Omega, written as Ω, is an Asymptotic Notation for the best case, or a floor growth rate for a given function. It gives us an asymptotic lower bound for the growth rate of the runtime of an algorithm. Average case — being represented as Big Theta – Θ(n) Theta, written as Θ, is an Asymptotic Notation to denote the asymptotically tight bound on the growth rate of the runtime of an algorithm. Worst case — being represented as Big O Notation – O(n) Big-O, written as O, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. It gives us an asymptotic upper bound for the growth rate of the runtime of an algorithm.

function reverseArray(arr) { let newArr = [] for (let i = arr.length - 1; i >= 0; i--) { newArr.push(arr[i]) } return newArr } const reversedArray1 = reverseArray([1, 2, 3]) // [3, 2, 1] const reversedArray2 = reverseArray([1, 2, 3, 4, 5, 6]) // [6, 5, 4, 3, 2, 1]

function logTime(arr) { let numberOfLoops = 0 for (let i = 1; i < arr.length; i *= 2) { numberOfLoops++ } return numberOfLoops } let loopsA = logTime([1]) // 0 loops let loopsB = logTime([1, 2]) // 1 loop let loopsC = logTime([1, 2, 3, 4]) // 2 loops let loopsD = logTime([1, 2, 3, 4, 5, 6, 7, 8]) // 3 loops let loopsE = logTime(Array(16)) // 4 loops

function linearithmic(n) { for (let i = 0; i < n; i++) { for (let j = 1; j < n; j = j * 2) { console.log("Hello") } } }

Exponential

```
function fibonacci(num) {
  // Base cases
  if (num === 0) return 0
  else if (num === 1) return 1
  // Recursive part
  return fibonacci(num - 1) + fibonacci(num - 2)
}
fibonacci(1) // 1
fibonacci(2) // 1
fibonacci(3) // 2
fibonacci(4) // 3
fibonacci(5) // 5
```

The Traveling Salesman Problem The usual example of an algorithm with factorial growth is the Travelling Salesman Problem: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? The brute force method would be to check every possible configuration between each city, which would be a factorial, and quickly get crazy!

Say we have 3 cities: A, B and C.

How many permutations are there? Permutations is a maths term meaning how many ways can we order a set of items.

A -> B -> C A -> C -> B B -> A -> C B -> C -> A C -> A -> B C -> B -> A With 3 cities, we have 3! permutations. That’s 1 x 2 x 3 = 6 permutations.

Why? If we have three possible starting points, then for each starting point, we have two possible routes to the final destination.

What if our salesman needs to visit 4 cities? A, B, C and D.

We’d have 4! = 4 x 3 x 2 x 1 = 24 permutations.

Why? If we have four possible starting points, then for each starting point, we have three possible routes, and for each of those points we have two possible routes, and the final stop - So:

4 *3* 2 * 1

function factorial(n) { let num = n if (n === 0) return 1 for (let i = 0; i < n; i++) { num = n * factorial(n - 1) } return num }