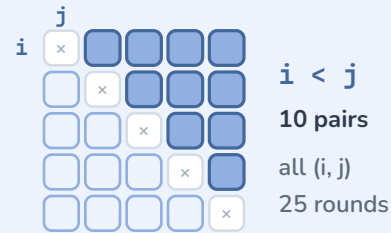


Complexity and limits

Chapter 1.2 · Overall level: First programs



Two programs can give the same answer on a small test and behave completely differently on a large one. *If the input becomes twice as large, does your program need twice the work, four times the work, or hardly any more?*

This chapter teaches you to answer that question with pencil and paper, before writing any code, by counting how many basic steps a plan performs and how that count grows with the size of the input.

1.2.1 Counting the work of a loop

A sensor records n readings. Consider two tasks on the same list. The first is to find the largest reading: look at each reading once and remember the biggest so far. The second is to check whether any two readings are equal by comparing them two at a time.

For the first task, a single loop visits each position once, so it performs n rounds. For the second, the most direct plan is a loop inside a loop: for every position i , look at every position j .

```
for i from 1 to n:                ← one scan: n rounds
    look at a[i]

for i from 1 to n:                ← all ordered pairs: n × n rounds
    for j from 1 to n:
        compare a[i] with a[j]
```

Count the rounds on a list of 5 readings. The scan does 5. The double loop does 5 rounds of the inner loop for each of the 5 outer rounds: 25. With 10 readings the scan does 10 and the double loop 100. The scan grew by a factor of 2, the double loop by a factor of 4.

1.2.2 The triangular loop

The double loop wastes effort. It compares each reading with itself, when $i = j$, and it compares every pair twice, once as (i, j) and again as (j, i) . To meet each unordered pair exactly once, start the inner loop just after i :

```
for i from 1 to n:
  for j from i + 1 to n:           ← only pairs with i < j
    compare a[i] with a[j]
```

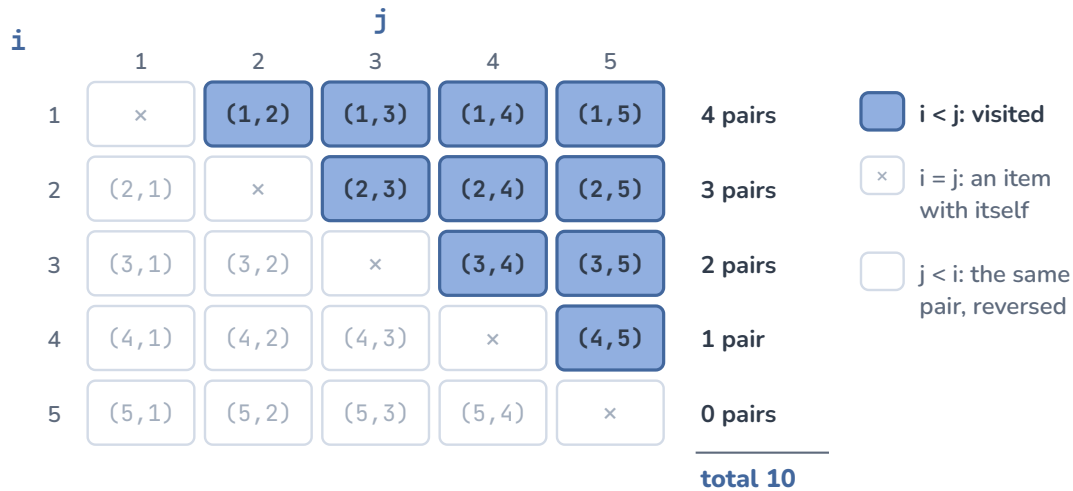


Figure 1.2.1 — The triangular loop visits every unordered pair exactly once. Row i contributes $n - i$ pairs.

Now the inner loop runs a different number of times in each round: 4 times when $i = 1$, then 3, 2, 1 and finally 0. The total is $4 + 3 + 2 + 1 + 0 = 10$, not 5×4 . For 10 readings it is $9 + 8 + \dots + 1 = 45$.

A formula for the number of pairs

There are $n \times n$ ordered pairs (i, j) . Removing the n pairs with $i = j$ leaves $n(n - 1)$. Each unordered pair appears among those exactly twice, as (i, j) and as (j, i) , so the number of pairs with $i < j$ is half of that:

$$\text{pairs}(n) = n(n - 1) / 2$$

$$\text{pairs}(5) = 5 \times 4 / 2 = 10$$

$$\text{pairs}(10) = 10 \times 9 / 2 = 45$$

The division is always exact, because one of two consecutive whole numbers $n - 1$ and n is even. So integer division gives the right answer, provided you multiply first and divide second.

n	one scan	all ordered pairs $n \times n$	pairs with $i < j$
5	5	25	10
10	10	100	45
1,000	1,000	1,000,000	499,500
200,000	200,000	40,000,000,000	19,999,900,000

Halving the work helped, but look at the last row. Whether the inner loop starts at 1 or at $i + 1$, the pair count grows like $n \times n$: two hundred thousand readings mean about twenty billion comparisons. A better *idea*, not a tidier loop, is needed for that size. Chapter 1.4 checks for equal readings with a single scan.

Independent parts add up

Suppose the readings come in separate batches and only readings in the same batch are compared. Then each batch does its own triangular loop, and the totals add. Two batches of 2 readings need $1 + 1 = 2$ comparisons; one batch of 4 needs 6. Squaring the overall total would count comparisons between batches that never happen.

1.2.3 Naming growth: O-notation

Exact counts depend on small details: whether a loop starts at 0 or 1, or whether one comparison is counted as one step or two. What stays the same is the **shape** of the growth. O-notation keeps only that shape:

- Keep the fastest-growing term and drop the rest: $n(n - 1)/2 = n^2/2 - n/2$ becomes $n^2/2$.
- Drop constant factors: $n^2/2$ becomes n^2 . So the triangular loop is $O(n^2)$, and so is the full double loop.
- A scan doing $3n + 2$ steps is $O(n)$. A fixed number of steps, whatever n is, is $O(1)$.

Read $O(f)$ as “grows at most in proportion to f ”. It is a statement about growth, not about seconds: two $O(n)$ programs may differ in speed by a large constant factor, yet both double their work when n doubles.

1.2.4 What doubling the input does

The quickest way to feel the difference between growth rates is to double n repeatedly and watch the counts.

n	$\log_2 n$	n	n^2
8	3	8	64
16	4	16	256
32	5	32	1,024
64	6	64	4,096
per doubling	+1	$\times 2$	$\times 4$

Counts of basic steps, not times.

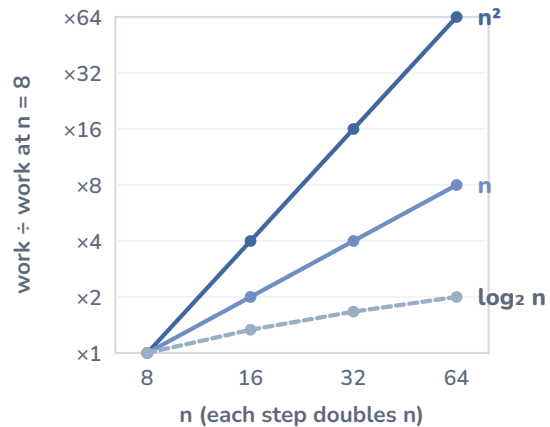


Figure 1.2.2 — Growth rates predict how work changes as the input grows, not a fixed execution time.

Each time n doubles, $O(n)$ work doubles, $O(n^2)$ work becomes four times larger, and $O(\log n)$ work grows by only one step. Ten doublings multiply n by about a thousand; they multiply n^2 by about a million, and add just 10 to $\log_2 n$.

This is why the growth rate matters more than the constant factor for large inputs. A program doing $50n$ steps is slower than one doing $n^2/2$ steps when $n = 20$ (1,000 against 200), equal at $n = 100$, and ten times faster at $n = 1,000$ (50,000 against 500,000).

1.2.5 Repeated halving and logarithms

Some loops shrink their work instead of stepping through it. This loop halves x with integer division until nothing is left:

```
x ← n
steps ← 0
while x > 0:
    steps ← steps + 1
    x ← x div 2
```

← whole part of $x / 2$

For $n = 13$ the loop sees $x = 13, 6, 3, 1$ and then 0, so it runs 4 times. The table shows how slowly that count grows:

n	1	2	13	16	100	1,000	10^6	10^{12}
rounds	1	2	4	5	7	10	20	40

Why so few? After k rounds, x is at most $n / 2^k$, because each round at least halves it. Once 2^k exceeds n , that bound is below 1, so x is 0 and the loop has stopped. The number of rounds is therefore about $\log_2 n$, the power to which 2 must be raised to reach n . Exactly, it is the number of binary digits of n (13 is 1101 in binary: 4 digits). Such a loop is $O(\log n)$; even for $n = 10^{12}$ it runs only 40 times.

It is tempting to compute the count with a floating-point `log2` instead of running the loop, but rounding can fool you. For $n = 2^{49} - 1$, Python's `math.log2` returns exactly 49.0, which suggests 50 digits, while the true count is 49. A loop of at most a few dozen integer steps is both exact and cheap.

1.2.6 Time and extra memory

Memory grows with the input too. We usually count **auxiliary space**: the extra memory a method uses beyond the input itself.

Method	Time	Auxiliary space
Add up n readings as they are read	$O(n)$	$O(1)$: just the total
Store all readings, then process them	$O(n)$	$O(n)$: the list
Store the result of every pair comparison	$O(n^2)$	$O(n^2)$: one entry per pair
Halving loop on one number n	$O(\log n)$	$O(1)$

$O(1)$ auxiliary space does *not* mean the input is small or the time is constant. Adding up 200,000 readings still reads 200,000 tokens; it simply keeps one running total instead of the whole list. In the other direction, a pair table for 200,000 readings would need 4×10^{10} entries, far more memory than any judge provides, even before counting time.

1.2.7 From constraints to a plan

Every problem statement lists **constraints**, such as “ $1 \leq n \leq 200,000$ ”. They tell you the largest input your program must handle, and they are the most useful hint you get. Use them in three steps.

1. **Propose a method** and write its step count as a formula in n .
2. **Put in the largest allowed values.** For the pair loop with $n = 200,000$ that is about 2×10^{10} ; for a scan, 2×10^5 ; for a method that sorts first (Chapter 1.5), about $n \log_2 n \approx 3.6 \times 10^6$.
3. **Compare.** Counts that differ by a factor of thousands decide the matter on their own. Counts within a small factor of each other are decided by details: the language, the amount of work per step, and how the input is read.

Constraints also hint at the intended method. If n is at most 20, the setter is expecting something that grows very fast; if n can be 10^{12} , even a single scan is too slow and the answer must come from a formula or a halving loop.

Why there is no universal speed rule

You may hear rules such as “a computer does so many operations per second”. Treat them with suspicion. The real speed of one “step” depends on the machine, the language and the step itself: an interpreted Python loop is often many times slower than the same loop in C++, and a step that touches memory scattered far apart costs more than one that walks through a list in order. O-notation deliberately ignores these constants. It tells you how the work will *change*, which is the part you can trust. When two plans are close, run your program on a test of the maximum size and see.

Numbers that outgrow their type

Step counts are often large numbers in their own right. With $n \leq 10^6$, $n \times n$ can reach 10^{12} , far beyond the 32-bit `int` limit of about 2.1×10^9 . In C++ and Java the *operands* must already be wide: if n is an `int`, the product $n * n$ overflows before it is stored, even into a `long long`. Declare n as `Long long` (C++) or `Long` (Java). JavaScript numbers are exact up to $2^{53} \approx 9 \times 10^{15}$, so 10^{12} is safe there, and Python integers have no limit.

1.2.8 Code in four languages

Each program solves the first practice problem, **Square budget**. A routine performs exactly $n \times n$ operations and a budget allows b . The input is one line “ $n\ b$ ” with $1 \leq n \leq 10^6$ and $0 \leq b \leq 10^{12}$; the output is YES if the routine fits the budget and NO otherwise.

Sample input	Sample output
12 140	NO

Here $12 \times 12 = 144$ is more than 140, so the answer is NO. With a budget of exactly 144 it would be YES: using the whole budget still fits.

Pseudocode

Compare with a formula: **O(1)** time and memory

```
read n and b
if n * n ≤ b:
    print YES
else:
    print NO
```

The programs do *not* run the routine or count its operations one by one: with $n = 10^6$ that loop would itself take 10^{12} rounds. The count is already given by a formula, so one multiplication and one comparison decide the answer. Recognising when a count can be computed instead of simulated is one of the most useful habits in this library.

Python 3

```
import sys

def main():
    data = sys.stdin.read().split()
    n = int(data[0])
    b = int(data[1])
    print("YES" if n * n <= b else "NO")

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    long long n, b;
    cin >> n >> b;
    cout << (n * n <= b ? "YES" : "NO") << '\n';
    return 0;
}
```

Java 17

```
import java.io.*;
import java.util.*;

public class Main {
    static BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    static StringTokenizer tok = new StringTokenizer("");

    static String next() throws IOException {
        while (!tok.hasMoreTokens()) tok = new StringTokenizer(in.readLine());
        return tok.nextToken();
    }

    public static void main(String[] args) throws IOException {
        long n = Long.parseLong(next());
        long b = Long.parseLong(next());
        System.out.println(n * n <= b ? "YES" : "NO");
    }
}
```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(data[0]);
const b = Number(data[1]);
console.log(n * n <= b ? "YES" : "NO");
```

In C++ and Java, n is read into a 64-bit variable so that $n * n$ is computed in 64 bits. In JavaScript, $n * n$ is at most 10^{12} , well below 2^{53} , so the comparison is exact.

1.2.9 Common mistakes

- **Multiplying loop bounds when the inner bound depends on i .** The triangular loop does $0 + 1 + \dots + (n - 1) = n(n - 1)/2$ rounds, not $n \times (n - 1)$. Add the rounds row by row, as in Figure 1.2.1.
- **Confusing $O(1)$ extra space with $O(1)$ input size.** A one-variable accumulator uses constant extra memory, but it still reads all n tokens, so its time is $O(n)$.
- **Trusting a universal operations-per-second rule.** Constants, hardware and language change real speed. Compare growth rates and test at the maximum size instead.
- **Overflow in step counts.** $n \times n$ and $n(n - 1)/2$ exceed 32 bits long before n reaches a million. Use 64-bit operands, and multiply before dividing by 2.
- **Simulating a count that has a formula.** Counting 10^{12} operations one at a time is itself 10^{12} steps.
- **Using a floating-point logarithm for an exact count.** Run the integer halving loop instead.

1.2.10 What growth rates do not tell you

O -notation describes behaviour for large inputs. For small inputs the constants dominate, and a simple $O(n^2)$ method can beat a clever $O(n \log n)$ one; with $n \leq 50$ there is rarely a reason to be clever. It also describes one input size at a time, usually the worst input of that size, so a method that is fast on typical data may still be slow on a carefully built test. And it hides which resource runs out first: a method can be fast enough and still need too much memory.

Finally, counting rounds works for loops whose bounds you can read off directly. Methods that call themselves, or whose cost is spread unevenly over many operations, need more careful counting; Chapter 2.4 starts on recursion, and later chapters state the costs of their data structures.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● If n doubles, roughly how does n^2 change?
2. ● Why is $O(n)$ not a promise that a program fits a time limit?
3. ● How many rounds does the halving loop of Section 1.2.5 make for $n = 100$? List the values of x .
4. ● How many times does the inner statement run for $n = 4$?
for i from 1 to n : for j from 1 to i : do one step
5. ●● A problem allows $n \leq 100,000$. Plan A compares all pairs; plan B sorts first and then scans. Estimate both step counts at the maximum and choose.
6. ●● Three batches have sizes 3, 3 and 6, and only items in the same batch are compared. How many comparisons are made? What wrong total do you get by applying the pair formula to all 12 items?
7. ● Plan C takes $100n$ steps and plan D takes n^2 steps. For which n is plan C the faster one?

Answers and explanations

Read these after attempting the exercises.

1. It becomes about **four times** as large: $(2n)^2 = 4n^2$.
2. O-notation ignores constant factors. The hardware, the language and the implementation decide how long each step takes, so the same growth rate can mean very different running times.
3. **7** rounds: 100, 50, 25, 12, 6, 3, 1, and then 0. 100 is 1100100 in binary, which has 7 digits.
4. $1 + 2 + 3 + 4 = 10$. The inner bound depends on i , so add row by row; it is $n(n + 1)/2$, still $O(n^2)$.
5. Plan A: about $100,000 \times 99,999 / 2 \approx 5 \times 10^9$. Plan B: about $100,000 \times 17 \approx 1.7 \times 10^6$ for sorting plus 10^5 for the scan. They differ by a factor of thousands, so choose **plan B**.
6. $3 + 3 + 15 = 21$. The formula on all 12 items gives $12 \times 11 / 2 = 66$, which wrongly counts pairs from different batches.
7. For every n **greater than 100**: $n^2 > 100n$ exactly when $n > 100$. At $n = 100$ they tie, and below that plan D does fewer steps. The constant 100 only moves the crossing point.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Square budget

Apply

A routine performs exactly $n \times n$ operations. A machine budget allows b operations. Print YES if the routine fits, otherwise NO.

Pair iterations

Connect

A loop visits every pair of positions i, j satisfying $1 \leq i < j \leq n$. Print how many iterations it performs.

Halving log

Reason

Start with $x = n$. While $x > 0$, record one operation and replace x by $\text{floor}(x/2)$. Print the number of recorded operations.

Batch pair workload

Stretch

There are t independent batches of sizes n_i . In each batch a routine compares every unordered pair of different items once. Print the total comparisons across all batches. Items in different batches are never compared.