Chapter 1: Introduction and Algorithm Analysis

Data Structures and Algorithms with C

System life cycle | Algorithms and ADTs | Selection sort | Space and time | O, Ω, Θ | Magic square | Measuring time

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_01_Introduction_and_Algorithm_Analysis.html

Learning Outcomes

Process

Describe the five phases of the system life cycle and where data structure decisions are made.

Abstraction

Define algorithms and abstract data types (ADTs); build an interface in C with a header file.

Counting

Work out the space requirement and step count of a C function (`sum` , `rsum` , `add`).

Classifying

Express step counts with O, Ω and Θ; separate the best, average and worst cases.

Comparing

Explain how the common complexity classes pull apart as n grows.

Measuring

Time a program with `clock()` and recognize the pitfalls (clock ticks, output).

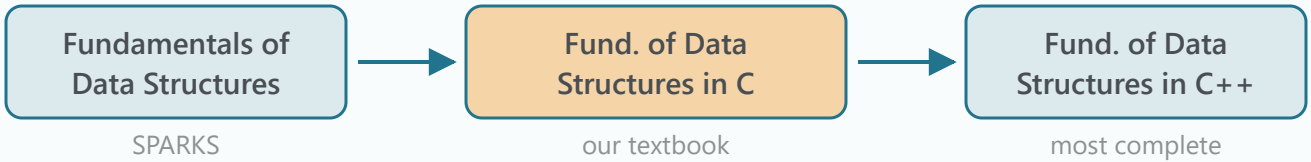
Textbooks and Resources

Desk reference

Kernighan and Ritchie, *The C Programming Language*. Written by C's designers; keep it within reach at all times.

Textbook

Horowitz, Sahni and Anderson-Freed, *Fundamentals of Data Structures in C*.



Three editions of one book: almost the same content, only the language of the code changes.

Further reading: Cormen et al. (CLRS), *Introduction to Algorithms*; Sedgewick, *Algorithms in C*.

Reading: Loving Your Work

Archaeologist

Lives today but thinks 5500 years back: “The pharaoh’s wife can’t have been buried there for that reason.”

Astronomer

Photographs for years to prove a black hole exists; thrilled when a star moves one pixel.

Football fan

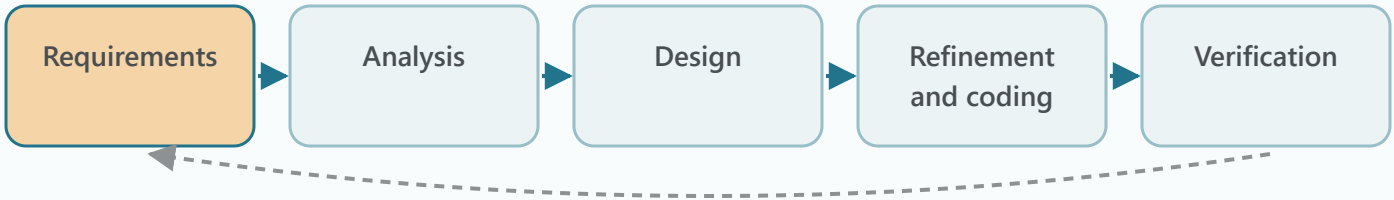
8 hours on the road, a 150 TL ticket, stones and sticks… Yet we call someone who travels as far for a conference a “fool”.

INSTRUCTOR *“Love your profession. People listen to someone who loves their work, never to someone who just goes through the motions. With code too, understand what happens as if you lived inside it.”*

“The way to a lifetime of happiness is to love your work.” — Chinese proverb

The System Life Cycle

Software goes through a process with the following phases:



Requirements: A project starts by specifying its purpose: which **inputs** are given, which **outputs** must be produced?

Horowitz–Sahni, Chapter 1

The System Life Cycle

Software goes through a process with the following phases:

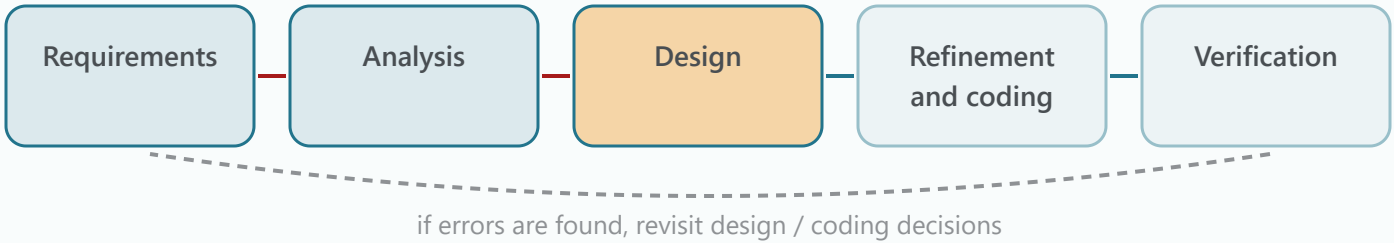


Analysis: The problem is broken into pieces. There are two approaches: **bottom-up** or **top-down**.

Horowitz–Sahni, Chapter 1

The System Life Cycle

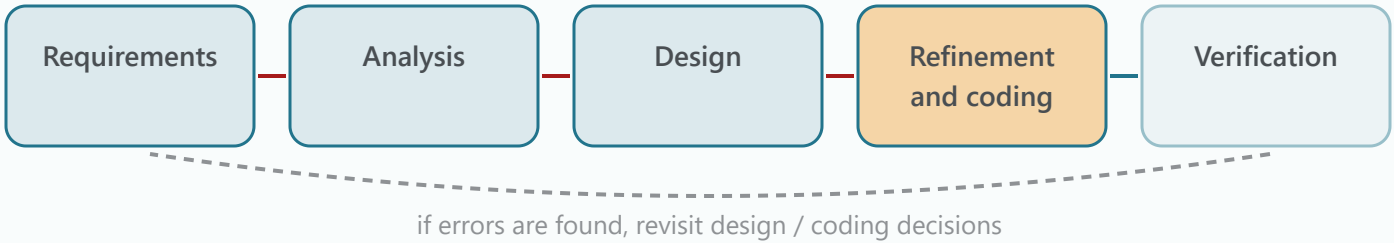
Software goes through a process with the following phases:



Design: Define the data objects (→ ADTs) and operations (→ algorithms); both are **language-independent**. More time spent on design means fewer errors.

The System Life Cycle

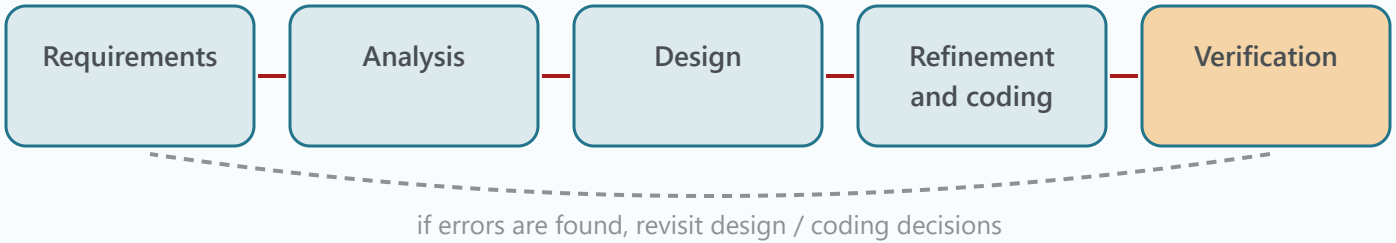
Software goes through a process with the following phases:



Refinement and coding: Algorithms are coded on the chosen data structures. The data representation determines how efficient the algorithm is.

The System Life Cycle

Software goes through a process with the following phases:



Verification: Correctness proofs, testing on varied data, and debugging. Each is a research area in its own right.

Analysis: Bottom-Up or Top-Down?

Bottom-up

+ Split the problem into pieces and focus on each one separately.
+ Modules are visible right away; the problem is solved module by module.
+ Preferred **when we are experienced in the domain**.

Top-down

+ Start with a master plan: “What is the goal?”, “What is the final product?”
+ Divide the problem into subproblems; this takes more time.
+ **Less** backtracking and fewer errors; preferred when we lack experience.

In the **design phase** we decide which fields a student record has, but we can **defer** whether to implement it with an array, a linked list or a tree. Deferring lets us pick a more efficient implementation later.

FOUNDATIONS

Coding and Verification

Correctness proofs

Mathematical techniques prove a program correct; proving a large project end to end is nearly impossible. Using proven algorithms reduces errors.

Testing

Needs working code and test data covering **all scenarios**. A program that runs without syntax errors is not necessarily correct.

Error removal

When proofs or tests reveal errors, debugging retraces design and coding decisions.

Running time

A correct but slow program is of little value: correctness and efficiency are separate questions.

INSTRUCTOR *"The longer you delay coding, the better the method you can work out."*

FOUNDATIONS

What Is an Algorithm?

An algorithm is a **finite** sequence of instructions that accomplishes a particular task. Every algorithm must satisfy these criteria:

Criterion	Meaning
Input	Zero or more quantities are supplied externally.
Output	At least one quantity is produced.
Definiteness	Each instruction is clear and unambiguous: no “add 6 or 7 to x”.
Finiteness	In every case, it terminates after a finite number of steps.
Effectiveness	Each instruction is basic enough to be carried out with pencil and paper.

A program that never terminates (e.g. an operating system) is not an algorithm but a **computational procedure**. We can describe an algorithm in natural language, as a flowchart or in C; we will use C.

Horowitz–Sahni, Chapter 1

Abstract Data Types

Data type = objects + operations. An **ADT** is a data type whose **specification** of objects and operations is **separated** from their representation and implementation.

Specification: WHAT?

Operation names, argument types, result type and what each operation does. This is all the user knows.

Implementation: HOW?

How the data is laid out in memory (struct, array…) and the code for the operations. Hidden from the user.

```
ADT Fraction
  objects: num/den, den ≠ 0
  operations:
    create(p, q)   → Fraction /* constructor */
    add(a, b)      → Fraction /* transformer */
    equal(a, b)    → Boolean  /* observer */
    print(a)                  /* reporter */
    destroy(a)
```

A **constructor** creates a new instance, a **transformer** builds a new value from existing ones, an **observer/reporter** returns information. The same ADT can be implemented with different data structures.

Horowitz–Sahni, Section 1.3

Abstraction in C: Header File + struct

```c
/* fraction.h : interface (WHAT?) */
#ifndef FRACTION_H
#define FRACTION_H
typedef struct fraction Fraction; /* opaque */

Fraction *fraction_create(int num, int den);
Fraction *fraction_add(const Fraction *a,
                       const Fraction *b);
int  fraction_equal(const Fraction *a,
                    const Fraction *b);
void fraction_print(const Fraction *f);
void fraction_free(Fraction *f);
#endif
```

```c
/* fraction.c : implementation (HOW?) */
#include <stdio.h>
#include <stdlib.h>
#include "fraction.h"

struct fraction { int num, den; };

Fraction *fraction_create(int num, int den) {
  Fraction *f;
  if (den == 0) return NULL;
  f = malloc(sizeof *f);
  if (f == NULL) return NULL;
  f->num = num;
  f->den = den;
  return f;
}
```

`typedef struct fraction Fraction;` declares an **incomplete (opaque) type**: a file that includes `fraction.h` cannot see the fields and only holds a `Fraction *`. The fields stay hidden in `fraction.c` → C's counterpart of an ADT.

The Rest of the Implementation, and Its Use

```
Fraction *fraction_add(const Fraction *a,
                       const Fraction *b) {
  return fraction_create(
      a->num * b->den
        + b->num * a->den,
      a->den * b->den);
}
int fraction_equal(const Fraction *a,
                   const Fraction *b) {
  return a->num * b->den
      == b->num * a->den;
}
void fraction_print(const Fraction *f) {
  printf("%d/%d\n", f->num, f->den);
}
void fraction_free(Fraction *f) { free(f); }
```

```
/* main.c : knows only the interface */
#include "fraction.h"
#include <stddef.h> /* NULL */
int main(void) {
  Fraction *a = fraction_create(1, 2);
  Fraction *b = fraction_create(1, 3);
  Fraction *s = NULL;
  if (a && b) s = fraction_add(a, b);
  if (s) fraction_print(s);  /* 5/6 */
  fraction_free(a); fraction_free(b);
  fraction_free(s);  /* free(NULL) is safe */
  return 0;
}
```

Contract

`fraction_create` returns `NULL` if the denominator is 0 or memory runs out; the caller checks.

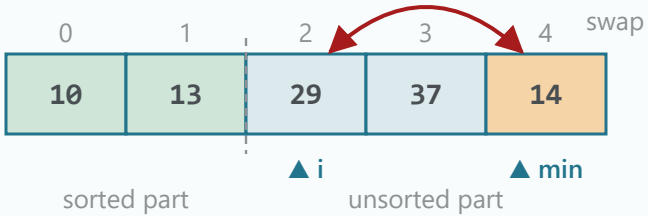
Ownership

Every result of `create` / `add` belongs to the caller and is released with `fraction_free`.

Selection Sort

We want to sort an array into ascending order. Each pass finds **the smallest of the remaining elements** and moves it to the front (position i):

```
for (i = 0; i < n; i++) {
    examine list[i] to list[n-1];
    let list[min] be the smallest integer;
    interchange list[i] and list[min];
}
```



INSTRUCTOR *"Aim for the level where you can find fault with published algorithms and offer better ones."*

Selection Sort in C

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int list[], int n);

int main(void) {
  int i, n, list[MAX_SIZE];
  printf("Enter the array size: ");
  if (scanf("%d", &n) != 1 || n < 1 || n > MAX_SIZE) {
    printf("Invalid value of n!\n");
    return 1;
  }
  for (i = 0; i < n; i++) list[i] = rand() % 100;
  sort(list, n);
  for (i = 0; i < n; i++) printf("%d ", list[i]);
  return 0;
}
```

```c
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i], list[min], temp);
  }
}
```

Comparisons: $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 \rightarrow \mathbf{O(n^2)}$. At most $n-1$ swaps.

Where Does a Common Faulty `sort()` Go Wrong?

```
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = 0; j < n; j++)
      if (list[j] > list[min]) {
        min = j;
        SWAP(list[i], list[min], temp);
      }
  }
}
```

Input	Version	Comp.	Swaps*
{29,10,14,37,13}	faulty	25	7
	selection	10	3
100 random	faulty	10000	2561
	selection	4950	94
500…1 (descending)	faulty	250000	124750
	selection	124750	250

The faulty version on the left **still sorts** the array (checked on all permutations for n ≤ 7), but it is **not** the selection sort of the pseudocode: the inner loop starts at `j = 0` and the swap sits inside the loop. The correct version (`j = i + 1` , swap after the inner loop) is the one used in this chapter.
*Swaps: `SWAP` s that exchange two distinct cells (`i ≠ min`).

SWAP: Macro or Function?

Macro

```
#define SWAP(x, y, t) \
        ((t) = (x), (x) = (y), (y) = (t))

int temp;
SWAP(list[i], list[min], temp);
```

+ No call overhead: the preprocessor substitutes the text.
+ The temporary `t` is supplied by the caller.

Function

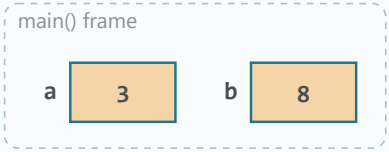
```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

swap(&list[i], &list[min]);
```

+ Type-checked; `temp` is local.
+ We must pass the **addresses** of the variables.

A macro may evaluate its arguments more than once: `SWAP(a[i++], a[j], t)` increments `i` twice! A function named `SWAP` would clash with the macro, so the function is called `swap`.

Code Trace: `swap(&a, &b)`



In `main`'s frame, **a = 3** and **b = 8**.

Step 1 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
▶    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

VARIABLES

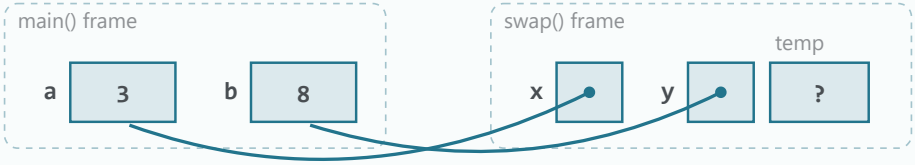
`a = 3` `b = 8`

CALL STACK

`main()`

OUTPUT

Code Trace: `swap(&a, &b)`



The call copies **addresses**, not values. x → a, y → b.

Step 2 / 6

```
▶ void swap(int *x, int *y) {
2   int temp = *x;
3   *x = *y;
4   *y = temp;
5 }
6
7 int main(void) {
8   int a = 3, b = 8;
▶   swap(&a, &b);
10  printf("%d %d\n", a, b);
11 }
```

VARIABLES

`x = &a` `y = &b`

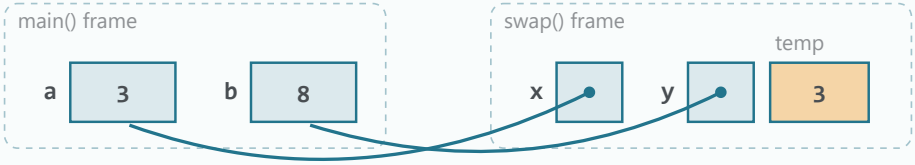
CALL STACK

`swap(&a, &b)`

`main()`

OUTPUT

Code Trace: `swap(&a, &b)`



`temp = *x` : read the value x points to (3).

Step 3 / 6

```
1  void swap(int *x, int *y) {
▶    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

VARIABLES

`x = &a` `y = &b` `temp = 3`

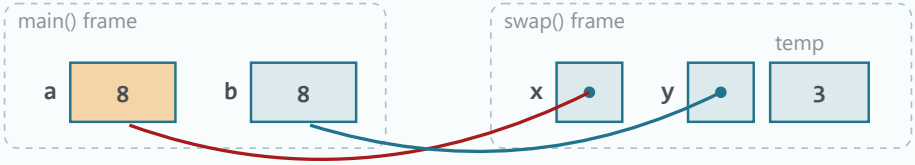
CALL STACK

`swap(&a, &b)`

`main()`

OUTPUT

Code Trace: `swap(&a, &b)`



`*x = *y` : the 8 that y points to is written where x points, i.e. into **a**.

Step 4 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
▶    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

VARIABLES

`x = &a` `y = &b` `temp = 3`

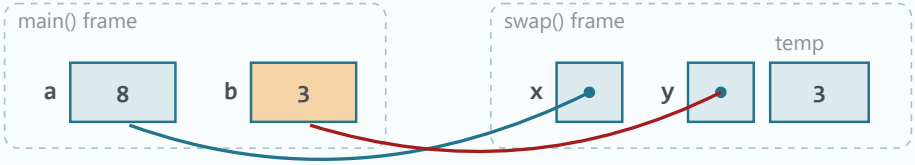
CALL STACK

`swap(&a, &b)`

`main()`

OUTPUT

Code Trace: `swap(&a, &b)`



`*y = temp` : 3 goes into b. The swap happened **in main's variables**.

Step 5 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
▶    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

VARIABLES

`x = &a` `y = &b` `temp = 3`

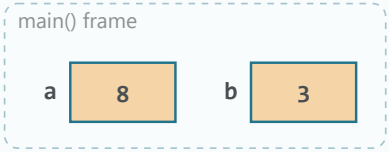
CALL STACK

`swap(&a, &b)`

`main()`

OUTPUT

Code Trace: `swap(&a, &b)`



swap's frame is gone (x, y, temp), but its effect remains.

Step 6 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
▶    printf("%d %d\n", a, b);
11 }
```

VARIABLES

a = 8 b = 3

CALL STACK

main()

OUTPUT

```
8 3
```

How Do We Judge a Program?

1. Does it meet the specification?
2. Does it work correctly?
3. Is it documented (with comments) so others can use it?
4. Does it use functions effectively to produce logical units?
5. Is the code readable?
6. **Does it use primary and secondary memory efficiently?**
7. **Is its running time acceptable?**

1 · Performance analysis

Machine-**independent**: estimates of time and space (complexity).

2 · Performance measurement

Machine-**dependent**: the actual running time is measured.

The last two questions are the subject of **performance evaluation**; the first five matter at least as much.

Space Complexity

The amount of memory a program needs to run to completion: **$S(P) = c + S_P(I)$**

Fixed part c

Independent of input/output size: the code itself, simple variables, constants.

Variable part $S_P(I)$

Depends on characteristics of input I (usually its size). This is **what we analyze**.

```
float abc(float a, float b, float c) {
  return a + b + b * c
         + (a + b - c) / (a + b) + 4.00;
}
/* fixed space only: S_abc(I) = 0 */
```

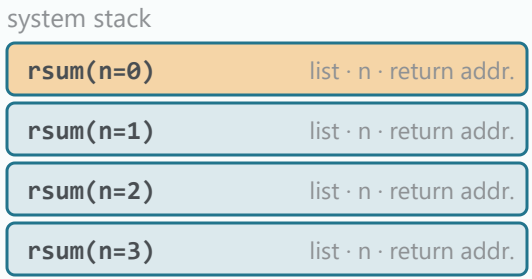
INSTRUCTOR *"Write code as if every resource were scarce. Plenty of memory is no excuse for wasting it."*

`sum` and `rsum`: Variable Space

```
float sum(float list[], int n) {
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    tempsum += list[i];
  return tempsum;
}

float rsum(float list[], int n) {
  if (n)
    return rsum(list, n - 1) + list[n - 1];
  return 0;
}
```

- `sum` receives the **address of the first element** → $S_{sum}(n) = 0$. If the elements were copied by value, $S_{sum}(n)$ would be n.
- `rsum(list, n)` is called n+1 times in all; each call pushes a stack frame → $S_{rsum}(n) \propto n$.



One frame per call:

on a 16-bit machine 2 + 2 + 2 = 6 bytes
n = 1000 → at least 6000 bytes
on 64-bit a frame is typically ≥ 24 bytes

PERFORMANCE ANALYSIS

Time Complexity

T(P) = compile time + run time. A program is compiled once; we care about the **run time** t_P.

$$t_P(n) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n)$$

Constants c

The time of each operation depends on the compiler and the machine; hard to know in advance.

Step count

Instead we count **program steps**: a meaningful, machine-independent unit.

INSTRUCTOR *"Why does performance matter so much? Suppose you are programming a fighter jet's bombing system. If yours releases in 3 s and the enemy's in 2 s, the battle is lost before it starts."*

PERFORMANCE ANALYSIS

Step Count Table: `sum()`

Statement	s/e	Frequency	Total
`float sum(float list[], int n) {`	0	0	0
`  float tempsum = 0;`	1	1	1
`  int i;`	0	0	0
`  for (i = 0; i < n; i++)`	1	n + 1	n + 1
`  tempsum += list[i];`	1	n	n
`  return tempsum;`	1	1	1
Total			**2n + 3**

s/e (steps per execution): how many steps one execution of the statement costs. The `for` test runs n+1 times: n times “continue”, once “done”. Declarations count as zero steps.

Code Trace: `sum(list, 3)` with `count`



Global `count = 0`. It goes up by one at every **program step**.

Step 1 / 10

```
▶ float sum(float list[], int n) {
2   float tempsum = 0; count++;
3   int i;
4   for (i = 0; i < n; i++) {
5     count++;              /* loop test */
6     tempsum += list[i]; count++;
7   }
8   count++;  /* last loop test */
9   count++;  return tempsum;
10 }
```

VARIABLES

`list = [3, 1, 4]` `n = 3` `count = 0`

Code Trace: `sum(list, 3)` with `count`



The assignment is one step → count = 1.

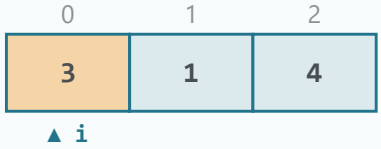
Step 2 / 10

```
1  float sum(float list[], int n) {
▶    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `tempsum = 0` `count = 1`

Code Trace: `sum(list, 3)` with `count`



Loop test **0 < 3** is true → one step (count = 2).

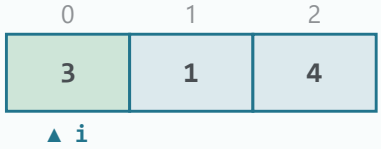
Step 3 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 0` `tempsum = 0` `count = 2`

Code Trace: `sum(list, 3)` with `count`



tempsum += list[0] → **3**; count = 3.

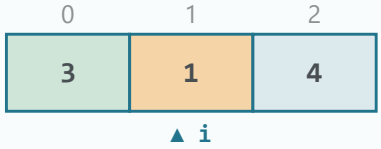
Step 4 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 0` `tempsum = 3` `count = 3`

Code Trace: `sum(list, 3)` with `count`



Loop test **1 < 3** is true → one step (count = 4).

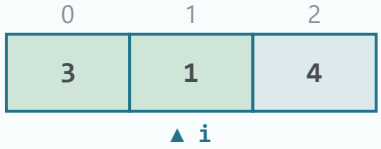
Step 5 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 1` `tempsum = 3` `count = 4`

Code Trace: `sum(list, 3)` with `count`



tempsum += list[1] → **4**; count = 5.

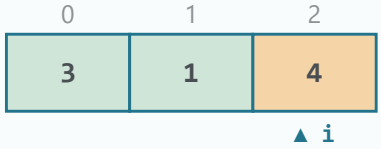
Step 6 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 1` `tempsum = 4` `count = 5`

Code Trace: `sum(list, 3)` with `count`



Loop test **2 < 3** is true → one step (count = 6).

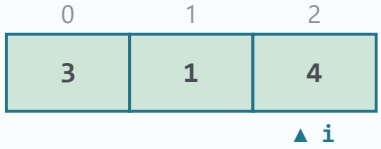
Step 7 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 2` `tempsum = 4` `count = 6`

Code Trace: `sum(list, 3)` with `count`



tempsum += list[2] → **8**; count = 7.

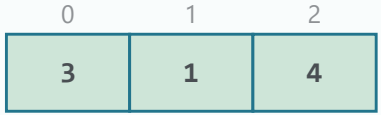
Step 8 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 2` `tempsum = 8` `count = 7`

Code Trace: `sum(list, 3)` with `count`



The final test **3 < 3** is false; it is a step too → count = 8.

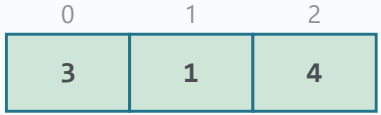
Step 9 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
▶    count++;  /* last loop test */
9    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `i = 3` `tempsum = 8` `count = 8`

Code Trace: `sum(list, 3)` with `count`



return: count = **9 = 2·3 + 3**, matching 2n + 3 from the table.

Step 10 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* loop test */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* last loop test */
▶    count++;  return tempsum;
10 }
```

VARIABLES

`n = 3` `tempsum = 8` `count = 9`

PERFORMANCE ANALYSIS

Step Count Table: `rsum()`

Statement	s/e	Frequency	Total
`float rsum(float list[], int n) {`	0	0	0
`if (n)`	1	n + 1	n + 1
`return rsum(list, n-1) + list[n-1];`	1	n	n
`return 0;`	1	1	1
Total			**2n + 2**

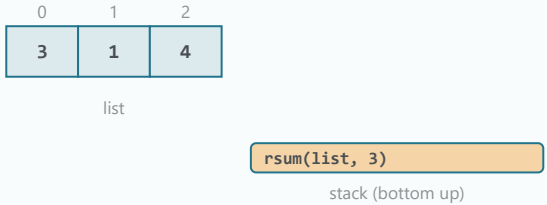
Step counts can mislead

rsum (2n+2) seems to take “fewer steps” than sum (2n+3), yet each recursive call (frame setup, return) costs more than one loop iteration.

Both are O(n)

What matters is how the step count grows **as n grows**: both are linear.

Code Trace: `rsum(list, 3)` and the Call Stack



rsum(list, 3): if test (count = 1); n = 3 ≠ 0.

Step 1 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if test */
▶   if (n) {
4     count++;            /* add + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

VARIABLES

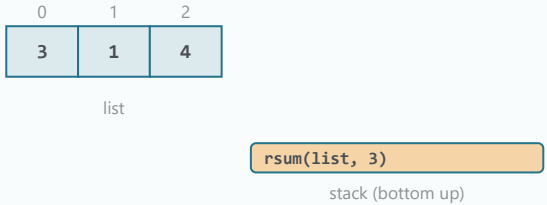
`n = 3` `count = 1`

CALL STACK

`rsum(list, 3)`

`main()`

Code Trace: `rsum(list, 3)` and the Call Stack



count = 2; **rsum(list, 2)** must be computed first → new frame.

Step 2 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
▶     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

VARIABLES

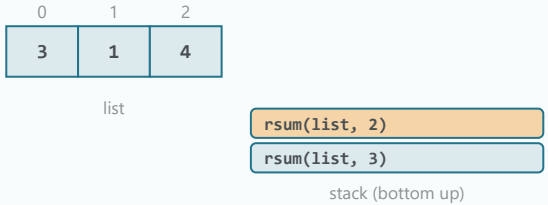
`n = 3` `count = 2`

CALL STACK

`rsum(list, 3)`

`main()`

Code Trace: `rsum(list, 3)` and the Call Stack



rsum(list, 2): if test (count = 3); n = 2 ≠ 0.

Step 3 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if test */
▶   if (n) {
4     count++;            /* add + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

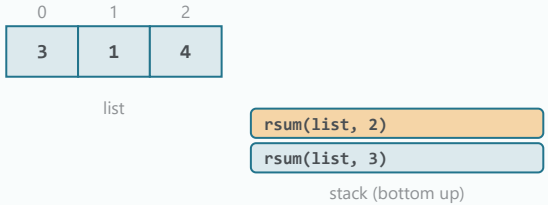
VARIABLES

`n = 2` `count = 3`

CALL STACK

- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



count = 4; **rsum(list, 1)** must be computed first → new frame.

Step 4 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
▶     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

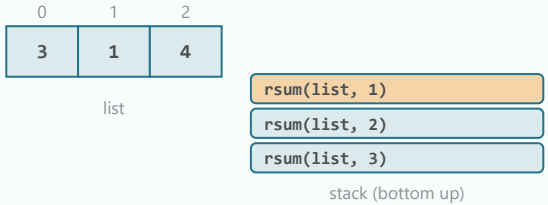
VARIABLES

`n = 2` `count = 4`

CALL STACK

- **`rsum(list, 2)`**
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



rsum(list, 1): if test (count = 5); n = 1 ≠ 0.

Step 5 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if test */
▶   if (n) {
4     count++;            /* add + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

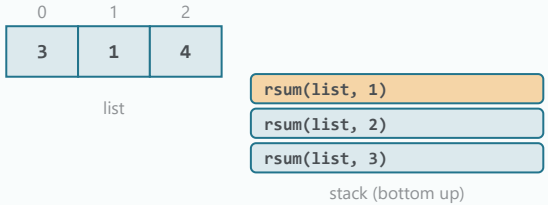
VARIABLES

`n = 1` `count = 5`

CALL STACK

- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



count = 6; **rsum(list, 0)** must be computed first → new frame.

Step 6 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
▶     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

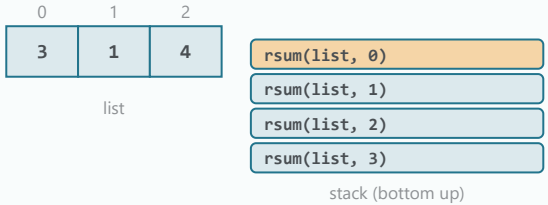
VARIABLES

`n = 1` `count = 6`

CALL STACK

- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



rsum(list, 0): if test (count = 7); n = 0 → **base case**.

Step 7 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if test */
▶   if (n) {
4     count++;            /* add + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

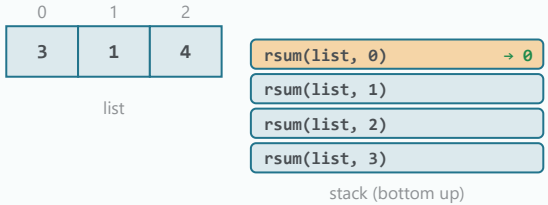
VARIABLES

`n = 0` `count = 7`

CALL STACK

- `rsum(list, 0)`
- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



count = 8; **return 0**. The stack is at its deepest: 4 frames.

Step 8 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
4     count++;              /* add + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
▶   count++;
▶   return 0;
9 }
```

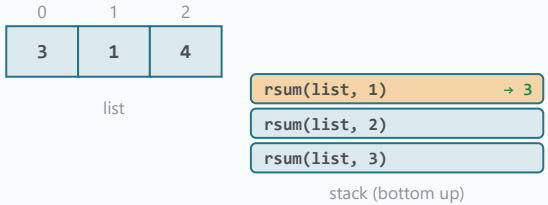
VARIABLES

`n = 0` `count = 8`

CALL STACK

- **`rsum(list, 0)`**
- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



Add list[0] = 3 to the returned value → rsum(list, 1) = **3**.

Step 9 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
4     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

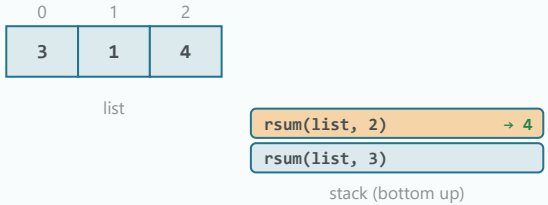
VARIABLES

`n = 1` `count = 8`

CALL STACK

- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



Add list[1] = 1 to the returned value → rsum(list, 2) = **4**.

Step 10 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
4     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

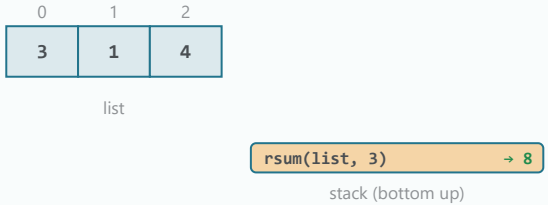
VARIABLES

`n = 2` `count = 8`

CALL STACK

- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Code Trace: `rsum(list, 3)` and the Call Stack



Add list[2] = 4 to the returned value → rsum(list, 3) = **8**. count = **8 = 2·3 + 2**.

Step 11 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if test */
3   if (n) {
4     count++;              /* add + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

VARIABLES

`n = 3` `count = 8`

CALL STACK

`rsum(list, 3)`

`main()`

PERFORMANCE ANALYSIS

Step Count: Matrix Addition `add()`

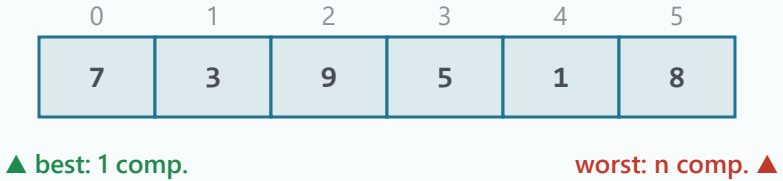
Statement	Frequency
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE], int c[][MAX_SIZE], int rows, int cols) {`	0
`int i, j;`	0
`for (i = 0; i < rows; i++)`	rows + 1
`for (j = 0; j < cols; j++)`	rows · (cols + 1)
`c[i][j] = a[i][j] + b[i][j];`	rows · cols
Total	**2·rows·cols + 2·rows + 1**

The result is **O(rows·cols)**; for a square matrix (rows = cols = n) it is **O(n^2)**. Looping over the smaller dimension outside shrinks the 2·rows term: with rows ≫ cols, swapping the loops gives 2·rows·cols + 2·cols + 1. Matrix multiplication is **O(n^3)**.

Writing "O(rows, cols)" is wrong; the product is correct: O(rows·cols).

Best, Average and Worst Case

For the same n, the step count may also depend on **the input itself**. In linear search:



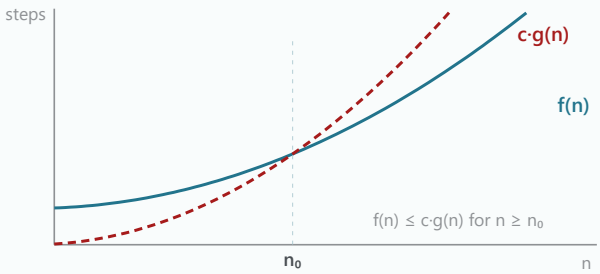
Case	When?	Comparisons
Best	Key is in the first cell	1 → Θ(1)
Average	Key present, every position equally likely	(n + 1) / 2 → Θ(n)
Worst	Key in the last cell or absent	n → Θ(n)

We usually report the **worst case**: it is a guaranteed upper bound for every input. The average case requires knowing the probability distribution of the inputs.

CLRS Section 2.2

Big-O Notation

Definition: f(n) = O(g(n)) if there exist constants **c > 0** and **n_0** such that f(n) ≤ c·g(n) for all n ≥ n_0: "the algorithm has complexity of order g(n)".



- $T_{sum}(n) = 2n + 3 = O(n)$
- $T_{rsum}(n) = 2n + 2 = O(n)$
- $T_{add} = 2 \cdot rows \cdot cols + 2 \cdot rows + 1 = O(rows \cdot cols) \rightarrow O(n^2)$

INSTRUCTOR *"Time matters more than memory!"*

O is an **upper bound**; here "=" means "is a member of" and reads one way only.

Example: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (7n \le n^2 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (7n^2 \le n^3 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

Bound the lower-order term 7n by n^2: $7n \le n^2$ whenever $n \ge 7$.

Step 1 / 4

1 · Upper bound

The running time never exceeds $4n^3$ steps ($n \ge 7$).

2 · Cubic growth

If sorting 100 numbers takes 2 s, sorting 300 takes at most $2 \cdot 3^3$ = **54 s** (not 6 s!).

Example: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2$ ($7n \le n^2$ for $n \ge 7$)

$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$

$3n^3 + 6n^2 + 7n \le 3n^3 + n^3$ ($7n^2 \le n^3$ for $n \ge 7$)

$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3)$ ($c = 4$, $n_0 = 7$)

$6n^2 + n^2 = 7n^2$.

1 · Upper bound

The running time never exceeds $4n^3$ steps ($n \ge 7$).

2 · Cubic growth

If sorting 100 numbers takes 2 s, sorting 300 takes at most $2 \cdot 3^3$ = **54 s** (not 6 s!).

Example: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (7n \le n^2 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (7n^2 \le n^3 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

Same trick once more: $7n^2 \le n^3$ for $n \ge 7$.

1 · Upper bound

The running time never exceeds $4n^3$ steps ($n \ge 7$).

2 · Cubic growth

If sorting 100 numbers takes 2 s, sorting 300 takes at most $2 \cdot 3^3$ = **54 s** (not 6 s!).

Example: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (7n \le n^2 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (7n^2 \le n^3 \text{ for } n \ge 7)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

We found $c = 4$ and $n_0 = 7$: the definition holds.

Step 4 / 4

1 · Upper bound

The running time never exceeds $4n^3$ steps ($n \ge 7$).

2 · Cubic growth

If sorting 100 numbers takes 2 s, sorting 300 takes at most $2 \cdot 3^3$ = **54 s** (not 6 s!).

Lower Bound Ω and Tight Bound Θ

Notation	Definition (for all $n \geq n_0$)	Meaning
$f(n) = O(g(n))$	$f(n) \leq c \cdot g(n)$	g is an **upper** bound on f
$f(n) = \Omega(g(n))$	$f(n) \geq c \cdot g(n)$	g is a **lower** bound on f
$f(n) = \Theta(g(n))$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	both O and Ω: a **tight** bound

- $3n + 2 = \Theta(n)$: $3n \leq 3n + 2 \leq 4n$ for $n \geq 2$.
- $3n + 2 = O(n^2)$ is also true but says little; we write the smallest g(n) we can.
- Selection sort makes $n(n-1)/2$ comparisons on every input → $\Theta(n^2)$.

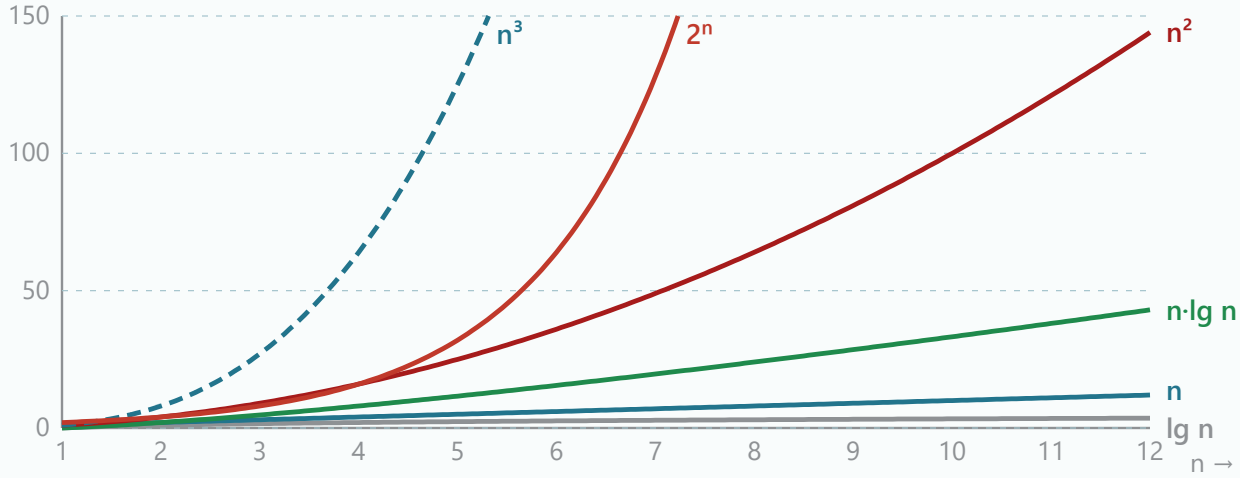
INSTRUCTOR *"An exam answer solved the problem in O(n), then added an empty loop running m times, because the question said 'solve it in O(n+m)'. No need. If O(n+m) is asked and you can do better, all the better!"*

Practical Complexities

Complexity	Name	n=1	n=2	n=4	n=8	n=16	n=32
1	Constant	1	1	1	1	1	1
lg n	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
n·lg n	Linearithmic	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1,024
n^3	Cubic	1	8	64	512	4,096	32,768
2^n	Exponential	2	4	16	256	65,536	$4.29 \cdot 10^9$
n!	Factorial	1	2	24	40,320	$2.09 \cdot 10^{13}$	$2.63 \cdot 10^{35}$

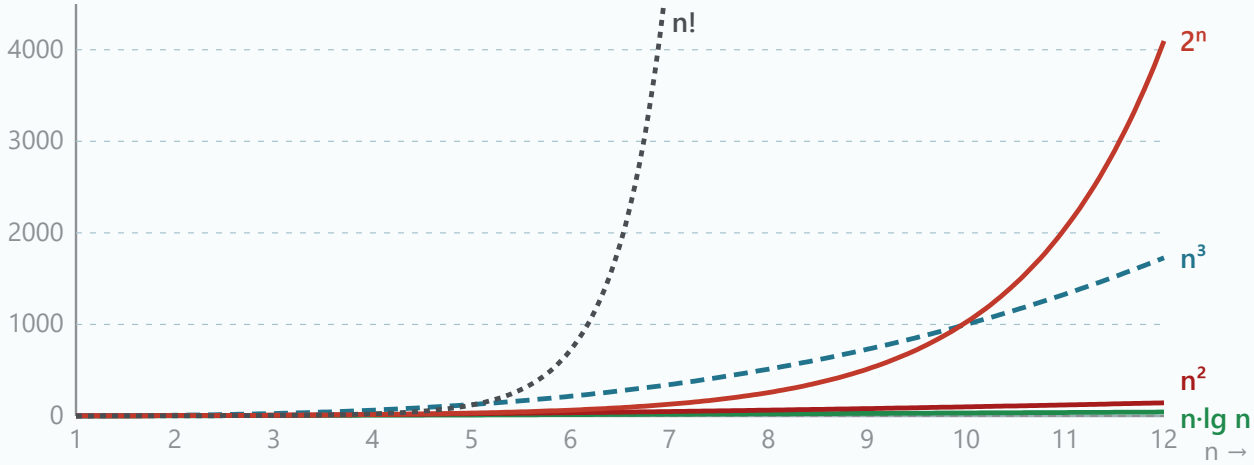
lg n = $\log_2 n$. Small values of n are misleading: at n = 4, $2^n = n^2$; at n = 8, $2^n = 256 > n^2 = 64$. Beyond some point the exponential and factorial terms crush everything else.

As n Grows: Growth Chart



Vertical scale 0–150: lg n, n and n·lg n separate; n^2 climbs fast, while n^3 and 2^n are already off the chart.

As n Grows: Growth Chart



Scale 0–4500: $n^3 = 1728$ ($n = 12$), $2^n = 4096$ ($n = 12$); $n!$ leaves the chart at $n = 7$ with 5040. The linear functions seem glued to the axis.

Running Times on a Machine Executing 10^9 Instructions per Second

n	n	lg n	n·lg n	n^2	n^3	n^4	n^{10}	2^n
10	0.01 µs	0.00332 µs	0.0332 µs	0.1 µs	1 µs	10.0 µs	10.0 s	1.02 µs
20	0.02 µs	0.00432 µs	0.0864 µs	0.4 µs	8 µs	160 µs	2.84 h	1.05 ms
30	0.03 µs	0.00491 µs	0.147 µs	0.9 µs	27.0 µs	810 µs	6.83 days	1.07 s
40	0.04 µs	0.00532 µs	0.213 µs	1.6 µs	64.0 µs	2.56 ms	4.05 mo	18.3 min
50	0.05 µs	0.00564 µs	0.282 µs	2.5 µs	125 µs	6.25 ms	3.1 yr	13.0 days
100	0.1 µs	0.00664 µs	0.664 µs	10.0 µs	1 ms	100 ms	31.7 cent.	$4.02 \cdot 10^{11}$ cent.
1000	1 µs	0.00997 µs	9.97 µs	1 ms	1 s	16.7 min	$3.17 \cdot 10^{11}$ cent.	$3.40 \cdot 10^{282}$ cent.
10000	10.0 µs	0.0133 µs	133 µs	100 ms	16.7 min	3.86 mo	$3.17 \cdot 10^{21}$ cent.	$6.33 \cdot 10^{2991}$ cent.

Each step takes 1 ns; 1 year = 365 days, 1 month = 30 days; cent. = centuries. For n = 10000, 2^n is about 10^{2992} centuries.

A First Look at Amortized Analysis

Some operations are **occasionally expensive** but usually cheap. Amortized analysis divides the **total** cost of a sequence of n operations by n. Example: an array whose capacity **doubles** when full (`realloc`); we trace it and count its copies in Chapter 2.

Over n insertions the copies are 1 + 2 + 4 + … < 2n → total < 3n. A single insertion may take O(n), but the **amortized cost per insertion is O(1)**. (Growing the capacity by a constant amount would give O(n).)

CLRS 3rd ed., Chapter 17 (Amortized Analysis)

The Magic Square Problem

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Every row, column and diagonal: 65

Place the numbers 1 to n^2 in an n×n square so that every row, every column and both main diagonals have the same sum: **$n(n^2+1)/2$** (65 for 5×5).

Coxeter's rule (n odd; also known as the Siamese or de la Loubère method): put 1 in the middle of the first row. Always move diagonally **up and to the left**; if that square is taken, move **down** from the current square instead. Moves off the square wrap around to the opposite edge. It finishes in n^2 steps → **$O(n^2)$**.

Brute force: $(n^2)!$ arrangements. $25! \approx 1.55 \cdot 10^{25}$ possibilities; at 10^9 tries per second that is $\approx 1.55 \cdot 10^{16}$ s ≈ **4.9 million centuries**.

“Computers Are Fast, It’ll Solve It Instantly!”?

> *A common objection: a new 10 GHz computer will solve this right away; no need to bother with algorithms.*

A machine 10× faster

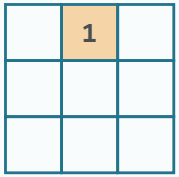
4.9 million centuries → about **490 thousand centuries**. Still hopeless.

The right algorithm

Coxeter’s rule: 25 steps, under a microsecond. **Algorithms come before hardware.**

Calculation: $25! / (10^{10} \cdot 3.15 \cdot 10^{9}) \approx 4.9 \cdot 10^{5}$ centuries.

Code Trace: `magic(3, sq)`



Middle of the first row: **i = 0, j = 1** → write 1.

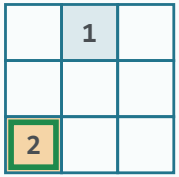
Step 1 / 10

```
1  void magic(int n, int sq[][MAX]) {
▶    int i = 0, j = (n - 1) / 2, s, r, c;
▶    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
5      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
8      else { i = r; j = c; }
9      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`n = 3` `i = 0` `j = 1` `sq[0][1] = 1`

Code Trace: `magic(3, sq)`



s = 2: the up-left square (2, 0) is free (row wrapped → bottom row) → move there.

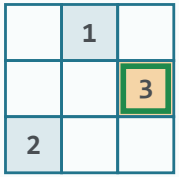
Step 2 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
▶      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 2` `i = 2` `j = 0` `r = 2` `c = 0`

Code Trace: `magic(3, sq)`



s = 3: the up-left square (1, 2) is free (column wrapped → rightmost column) → move there.

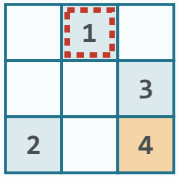
Step 3 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
       r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
       c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
       else { i = r; j = c; }
       sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 3` `i = 1` `j = 2` `r = 1` `c = 2`

Code Trace: `magic(3, sq)`



s = 4: the up-left square (0, 1) is **taken** (it holds 1) → one down: (2, 2).

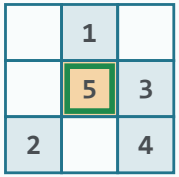
Step 4 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
▶      if (sq[r][c]) i = (i + 1) % n; /* down */
8      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 4` `i = 2` `j = 2` `r = 0` `c = 1`

Code Trace: `magic(3, sq)`



s = 5: the up-left square (1, 1) is free → move there.

Step 5 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
       r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
       c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
       else { i = r; j = c; }
       sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 5` `i = 1` `j = 1` `r = 1` `c = 1`

Code Trace: `magic(3, sq)`

6	1	
	5	3
2		4

s = 6: the up-left square (0, 0) is free → move there.

Step 6 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
▶      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

s = 6 | i = 0 | j = 0 | r = 0 | c = 0

Code Trace: `magic(3, sq)`

6	1	
7	5	3
2		4

s = 7: the up-left square (2, 2) is **taken** (it holds 4) → one down: (1, 0).

Step 7 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
▶      if (sq[r][c]) i = (i + 1) % n; /* down */
8      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 7` `i = 1` `j = 0` `r = 2` `c = 2`

Code Trace: `magic(3, sq)`

6	1	8
7	5	3
2		4

s = 8: the up-left square (0, 2) is free (column wrapped → rightmost column) → move there.

Step 8 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
▶      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 8` `i = 0` `j = 2` `r = 0` `c = 2`

Code Trace: `magic(3, sq)`

6	1	8
7	5	3
2	9	4

s = 9: the up-left square (2, 1) is free (row wrapped → bottom row) → move there.

Step 9 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
▶      else { i = r; j = c; }
▶      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 9` `i = 2` `j = 1` `r = 2` `c = 1`

Code Trace: `magic(3, sq)`

6	1	8
7	5	3
2	9	4

s = 10 > 9: the loop ends. Every row, column and diagonal sums to **15** = 3·(9+1)/2.

Step 10 / 10

```
1  void magic(int n, int sq[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    sq[i][j] = 1;        /* sq must be all zeros */
▶    for (s = 2; s <= n * n; s++) {
5      r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
7      if (sq[r][c]) i = (i + 1) % n; /* down */
8      else { i = r; j = c; }
9      sq[i][j] = s;
10   }
11 }
```

VARIABLES

`s = 10` `i = 2` `j = 1`

Performance Measurement: Two Methods

	1 · Processor time (clock)	2 · Calendar time (time)
Start	`start = clock();`	`start = time(NULL);`
Stop	`stop = clock();`	`stop = time(NULL);`
Type	`clock_t`	`time_t`
Result (s)	`(double)(stop - start) / CLOCKS_PER_SEC`	`difftime(stop, start)`

clock()

Number of processor clock ticks used by the program. Resolution is limited by the system's tick interval.

time()

Wall-clock time with one-second resolution. Fine for long jobs, too coarse for short ones.

`CLK_TCK` is an old, nonstandard name; standard C uses **`CLOCKS_PER_SEC`**. I/O devices (screen, disk, printer) are slow: **keep `printf` calls outside the timed region.**

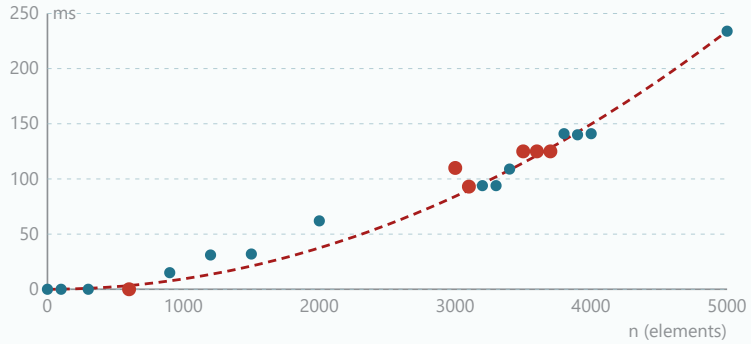
Exercise 1: Timing Selection Sort

```c
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 5001
#define ITERATIONS 20
void sort(int list[], int n);   /* selection sort */

int main(void) {
  int i, j, list[MAX_SIZE];
  int sizelist[] = { 0, 100, 300, 600, 900, 1200, 1500, 2000,
    3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800,
    3900, 4000, 5000 };
  clock_t start, stop;  double ms;
  for (i = 0; i < ITERATIONS; i++) {
    for (j = 0; j < sizelist[i]; j++)
      list[j] = sizelist[i] - j;   /* descending */
    start = clock();
    sort(list, sizelist[i]);
    stop = clock();
    ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
    printf("%4d elements: %6.1f ms\n", sizelist[i], ms);
  }
  return 0;
}
```

- 20 values of n; for each, an array in **descending** order.
- Only the `sort` call lies between the two `clock()` calls.
- Writing `(int)(1000*(stop-start)/CLK_TCK)` drops the fraction through integer division; **double** arithmetic with `1000.0` is more accurate.

Results and Inconsistencies



Dots: sample measurements · dashed: $c \cdot n^2$ curve

- Everything up to 600 elements takes **0 ms**?
- 3000 elements 110 ms, but 3100 only 93 ms?
- 3500, 3600 and 3700 all 125 ms?

Why? If the sort finishes between two clock ticks, `stop - start = 0`. On a system with a ~15.6 ms tick, results are rounded to multiples of 15–16 ms.

Fix: Repeat each measurement many times and take the average.

Exercise 2: Sequential Search (with a Sentinel)

```
int seqsearch(int list[], int searchnum, int n) {
  int i;
  list[n] = searchnum;          /* sentinel */
  for (i = 0; list[i] != searchnum; i++)
    ;
  return (i < n) ? i : -1;
}
```

```
start = clock();
for (j = 0; j < runtimes[i]; j++)
  pos = seqsearch(list, -1, sizelist[i]);
stop = clock();
total = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
avg = 1000.0 * total / runtimes[i];  /* in µs */
list[sizelist[i]] = sizelist[i];  /* undo sentinel */
```

- **Sentinel**: the key is stored in `list[n]`, so the loop always stops and needs no `i < n` test on each pass.
- The array needs room for **n + 1** elements (MAX_SIZE = 5001).
- We search for −1, which is never found → the **worst case** is measured.
- For each n the search is repeated `runtimes[i]` times (500000 … 5000).

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



The array holds 3 elements, but **room for list[3] is reserved**. We search for 5.

Step 1 / 7

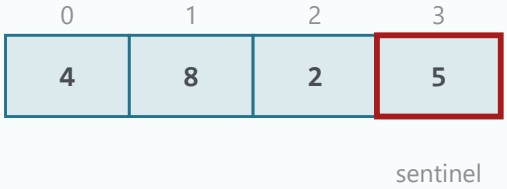
```
▶ int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;           /* sentinel */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, ?]`

`searchnum = 5` `n = 3`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



Sentinel: **list[3] = 5**. The loop now stops here at the latest.

Step 2 / 7

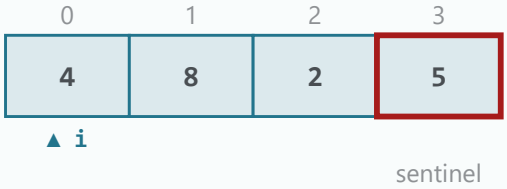
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
▶   list[n] = searchnum;          /* sentinel */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



list[0] = 4 ≠ 5 → i++. (One test: no bounds check.)

Step 3 / 7

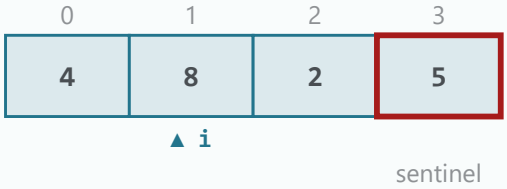
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* sentinel */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 0`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



list[1] = 8 ≠ 5 → i++. (One test: no bounds check.)

Step 4 / 7

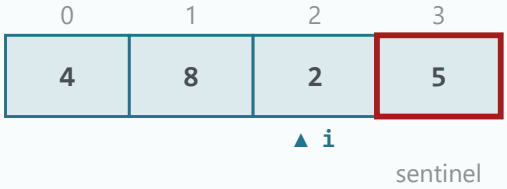
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;           /* sentinel */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 1`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



list[2] = 2 ≠ 5 → i++. (One test: no bounds check.)

Step 5 / 7

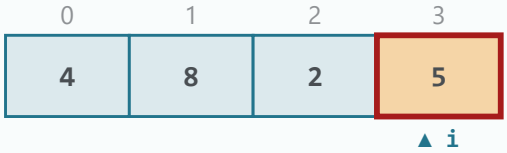
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;           /* sentinel */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 2`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



sentinel

list[3] = 5 → the loop stops, but this is the **sentinel**.

Step 6 / 7

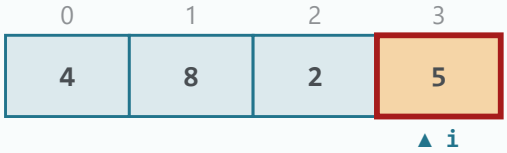
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;           /* sentinel */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 3`

Code Trace: `seqsearch({4, 8, 2}, 5, 3)`



i = 3 is not < n → **return −1**: not found. n + 1 = 4 comparisons.

Step 7 / 7

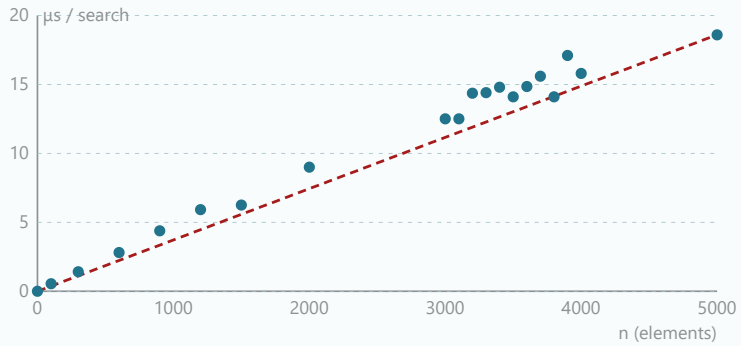
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;           /* sentinel */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
▶   return (i < n) ? i : -1;
7 }
```

VARIABLES

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 3`

Repeated Measurement: Average Search Time



Dots: measured averages · dashed: linear trend

- The averages are no longer 0 and grow **linearly** with n, as O(n) predicts.
- Fluctuation beyond 3000: with fewer repetitions (runtimes) the measurement gets coarse again.
- The test machine ticks 1000 times per second: 1 tick = 1 ms.

Repeat short jobs **many times** and divide the total time by the repetition count.

Homework (Recursion)

1. Write a **recursive** function that computes n!.
2. The Fibonacci numbers are $f_0 = 0$, $f_1 = 1$, $f_i = f_{i-1} + f_{i-2}$ ($i > 1$). Write a recursive function that computes f_i.
3. Given $C(n, m) = C(n-1, m) + C(n-1, m-1)$, write a recursive function that computes $C(n, m) = n! / (m! \cdot (n-m)!)$.

▶ **Hints**

Solutions and Common Mistakes

```
long fact(long n) {
  if (n <= 1) return 1;
  return n * fact(n - 1);
}

long fib(long n) {
  if (n <= 1) return n;   /* f0 = 0, f1 = 1 */
  return fib(n - 1) + fib(n - 2);
}

long binom(int n, int m) {
  if (m == 0 || m == n) return 1;
  return binom(n - 1, m) + binom(n - 1, m - 1);
}
```

fib: + not *

A common mistake is writing `fib(n-1) * fib(n-2)`; the product is always 0 or 1. By definition it must be a **sum**.

binom: parameters

A common mistake is writing `long binom()` with no parameters; `n` and `m` must be parameters.

Cost

fact is O(n); fib and binom as written are **exponential**: the same subproblems are solved again and again.

SUMMARY

Complexity Summary

Function	Time	Extra space	Note
`sort` (selection sort)	$\Theta(n^2)$	$O(1)$	$n(n-1)/2$ comparisons, $\leq n-1$ swaps
`sum`	$\Theta(n)$	$O(1)$	$2n + 3$ steps
`rsum`	$\Theta(n)$	$O(n)$	$2n + 2$ steps, $n+1$ frames
`add` (matrix addition)	$\Theta(\text{rows}\cdot\text{cols})$	$O(1)$	$\Theta(n^2)$ for a square matrix
`seqsearch`	$O(n)$	$O(1)$	best $\Theta(1)$, worst $n + 1$
`magic` (Coxeter)	$\Theta(n^2)$	$O(1)$	brute force: $(n^2)!$
Dynamic array insertion	$O(1)$ amortized	$O(n)$	a single insertion is $O(n)$ worst case
`fib` (recursive)	exponential	$O(n)$	stack depth n

SUMMARY

Common Mistakes

1. Thinking “it compiled and ran, so it’s correct”: test data must cover every scenario.
2. Confusing step counts with constants: 2n + 3 and 100n are in the same class, O(n).
3. Finding $O(n^2)$ and calling it “the best bound”: write the smallest g(n) you can; use Θ for a tight bound.
4. Timing a short job with a single `clock()` pair: the result is 0 or a multiple of the tick.
5. Leaving `printf` in the timed region: I/O time swamps the algorithm.
6. Passing an argument with side effects to a macro: `SWAP(a[i++], …)`.

SUMMARY

Exercises

1. How many times does `x++` execute in `for (i = 0; i < n; i++) for (j = 0; j < i; j++) x++;`? O(?)
2. Show that $10n^2 + 100n + 1000 = O(n^2)$ by giving c and n_0.
3. How many comparisons does selection sort make on an already sorted array? Does it have a best case?
4. An O(n log n) algorithm takes 10 ms for n = 1000. About how long for $n = 10^6$?
5. Why is calling `rsum` with $n = 10^6$ dangerous?
6. Write the first row of a 7×7 magic square using Coxeter's rule (what must the sum be?).

▶ **Show answers / hints**

SUMMARY

Chapter Summary

Process

Requirements → analysis → design → coding → verification. Deferring the implementation leads to better choices.

Abstraction

ADT = specification ⊥ implementation. In C: a header file + an opaque `struct`.

Counting

Space: $c + S_P(I)$. Time: program steps (sum $2n+3$, rsum $2n+2$).

Classifying

O upper, Ω lower, Θ tight bound. The worst case is a guarantee; amortized analysis looks at the whole sequence.

Growth

Small n misleads; 2^n and $n!$ quickly become hopeless. Algorithms come before hardware.

Measurement

`clock()` + repetition + averaging; keep I/O out of the timing.

SUMMARY

References

- Horowitz, Sahni, Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 1.
- Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 3rd ed.: Chapters 2–3 (analysis, asymptotics), Chapter 17 (amortized analysis).
- Kernighan, Ritchie, *The C Programming Language*, 2nd ed.

Course page

Chapter 2: Arrays and Structures

The End

Chapter 1: Introduction and Algorithm Analysis

Next chapter: Chapter 2: Arrays and Structures · Course page