# Chapter 3: Recursion, Binary Search and Backtracking

## Data Structures and Algorithms with C

Stack frames · Tail recursion · Binary search · Towers of Hanoi · Permutations · Satisfiability · n-queens · Knight's tour

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_03_Recursion_Binary_Search_and_Backtracking.html

# Learning Outcomes

### Thinking recursively

Express a problem as a base case + a shrinking subproblem; compare recursion with iteration.

### Tracing the stack

Draw the stack frame of each call (parameters, locals, return address).

### Binary search

Write iterative and recursive binary search; show the O(log n) bound.

### Recurrences

Set up and solve $h(n) = 2h(n-1) + 1$ for Hanoi; $O(n \cdot n!)$ for permutations.

### Backtracking

Tell brute force from backtracking; SAT, n-queens, subset generation.

### Heuristics

Explain why Warnsdorff's rule beats backtracking on the knight's tour.

# Recursive Algorithms

When the logic that solves the whole problem **also applies to its subproblems**, a recursive algorithm can be used.

### Direct recursion

A function calls itself: `fact` → `fact`.

### Indirect recursion

A function it calls calls it back: `A` → `B` → `A`.

### Base case

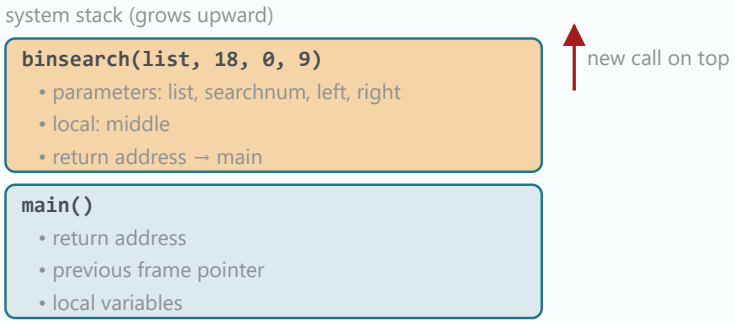
The smallest instance, solved without recursion. Without it the calls never end (the stack overflows).

### Recursive step

Reduces the problem to a **smaller** instance; every call must move toward the base case.

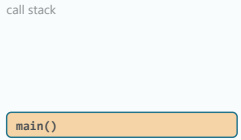
**INSTRUCTOR** *"'Doesn't this recursive solution eat a lot of memory?' Too early for that question. First learn to write the code; only once you can compute its complexity can you say so. You can't judge recursion before you can write it."*

# The Stack Frame




- A function call uses the **system stack**.
- Before control passes to the function, some bookkeeping is done: where to return, saving the parameters…
- This communication and control structure is called a **stack frame**.
- Initially the frame holds two values: the **return address** and a **pointer to the previous frame**.
- Every recursive call means a new frame → at depth d the extra space is O(d).

# Code Trace: `fact(3)` and the Call Stack




main calls `fact(3)` : a new frame will be pushed.

Step 1 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
4    return n * fact(n - 1);
5  }
6
7  int main(void) {
▶    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

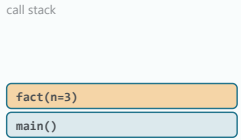
`n = –`

**CALL STACK**

`main()`

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




fact(3): n = 3 > 1, not the base case.

Step 2 / 9

```
1  long fact(int n) {
▶    if (n <= 1)
3      return 1;              /* base case */
4    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

`n = 3`

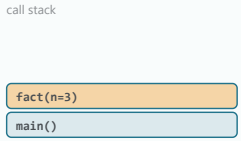
**CALL STACK**

`fact(3)`

`main()`

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




`3 * fact(2)` : the multiplication waits; fact(2) is called first.

Step 3 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
▶    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

`n = 3`

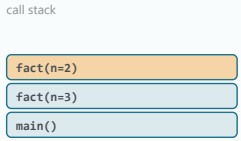
**CALL STACK**

`fact(3)`

`main()`

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




fact(2): n = 2 > 1, not the base case.

Step 4 / 9

```
1  long fact(int n) {
▶    if (n <= 1)
3      return 1;              /* base case */
4    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

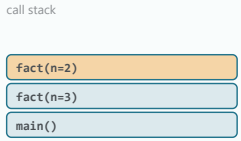
`n = 2`

**CALL STACK**

- **fact(2)**
- fact(3)
- main()

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




`2 * fact(1)` : the multiplication waits; fact(1) is called first.

Step 5 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
▶    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

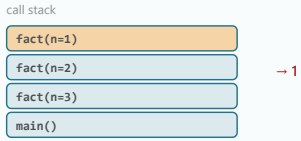
`n = 2`

**CALL STACK**

- **fact(2)**
- fact(3)
- main()

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




fact(1): **base case** → return 1. The stack is at its deepest (4 frames).

Step 6 / 9

```
1  long fact(int n) {
▶    if (n <= 1)
▶      return 1;              /* base case */
4    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

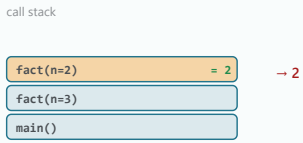
`n = 1`

**CALL STACK**

- `fact(1)`
- `fact(2)`
- `fact(3)`
- `main()`

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




fact(2) resumes where it left off: 2 · 1 = **2**, its frame is popped.

Step 7 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
▶    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

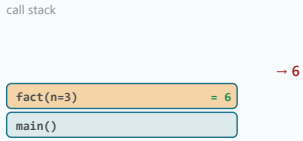
`n = 2` `fact(n-1) = 1`

**CALL STACK**

- **fact(2)**
- fact(3)
- main()

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




fact(3) resumes where it left off: 3 · 2 = **6**, its frame is popped.

Step 8 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
▶    return n * fact(n - 1);
5  }
6
7  int main(void) {
8    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

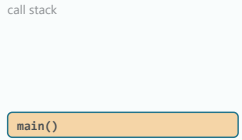
`n = 3` `fact(n-1) = 2`

**CALL STACK**

- `fact(3)`
- `main()`

**OUTPUT**

# Code Trace: `fact(3)` and the Call Stack




main prints 6. The variable n is a **separate** copy in every frame.

Step 9 / 9

```
1  long fact(int n) {
2    if (n <= 1)
3      return 1;              /* base case */
4    return n * fact(n - 1);
5  }
6
7  int main(void) {
▶    printf("%ld\n", fact(3));
9    return 0;
10 }
```

**VARIABLES**

`n = –`

**CALL STACK**

`main()`

**OUTPUT**

```
6
```

# Recursion or Iteration?

```
/* recursive */
long fact(int n) {
  return n <= 1 ? 1 : n * fact(n - 1);
}

/* iterative */
long fact_iter(int n) {
  long f = 1;
  while (n > 1)
    f *= n--;
  return f;
}
```

| | Recursive | Iterative |
|---|---|---|
| Readability | Close to the definition | Needs state variables |
| Extra space | O(depth) stack | O(1) |
| Call overhead | A frame per step | None |
| Risk | Stack overflow when deep | Infinite loop |
| Natural for | Trees, divide and conquer, backtracking | Linear scans, counters |

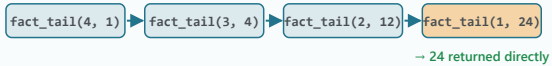
Any recursive algorithm can be turned into a loop by simulating the call stack with an **explicit stack** (and vice versa). The choice trades clarity against space and speed.

Horowitz–Sahni 1.2.2

# Tail Recursion

If the recursive call is the **last thing** a function does (nothing is left to do after it returns), it is called tail recursion.

```c
long fact_tail(int n, long acc) {
  if (n <= 1) return acc;
  return fact_tail(n - 1, n * acc);  /* last: call */
}
/* first call: fact_tail(n, 1) */
```




### fact is not tail-recursive

`n * fact(n-1)` : a multiplication waits after the return, so the frame must be kept.

### Accumulator parameter

`acc` carries the partial result; nothing is left to do on return.

### The compiler

A tail call can become a **jump** (O(1) stack). The C standard **does not guarantee** this; gcc/clang usually do it when optimizing.

# Binary Search

We search a **sorted** array for a value: return its **index** if found, **−1** otherwise. Each step looks at the middle element and discards half of the range.




Test array

```
#define COMPARE(x, y) ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)
```

**INSTRUCTOR** *"Return i, not list[i]. Why? So that the error value −1 can't be confused with an array element that happens to be −1."*

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




`binsearch(list, 18, 0, 9)` : the range is the whole array.

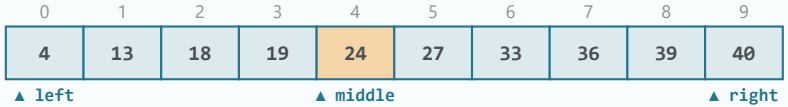
Step 1 / 7

```
▶ int binsearch(int list[], int searchnum,
▶               int left, int right) {
3   while (left <= right) {
4     int middle = (left + right) / 2;
5     switch (COMPARE(list[middle], searchnum)) {
6       case -1: left = middle + 1; break;
7       case 0: return middle;
8       case 1: right = middle - 1;
9     }
10  }
11  return -1;
12 }
```

VARIABLES

`searchnum = 18` `left = 0` `right = 9`

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




0 ≤ 9: round 1. middle = (0 + 9) / 2 = **4**, list[4] = 24.

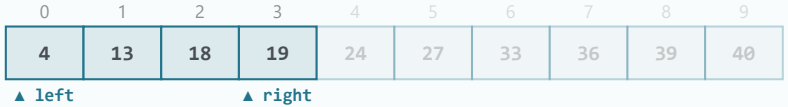
Step 2 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
   while (left <= right) {
     int middle = (left + right) / 2;
5    switch (COMPARE(list[middle], searchnum)) {
6      case -1: left = middle + 1; break;
7      case 0: return middle;
8      case 1: right = middle - 1;
9    }
10 }
11 return -1;
12 }
```

VARIABLES

`searchnum = 18` `left = 0` `right = 9` `middle = 4`

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




24 > 18 → COMPARE = 1: discard the right half, **right = 3**.

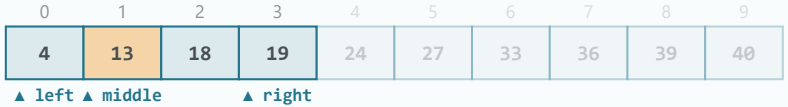
Step 3 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
▶        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

`searchnum = 18` `left = 0` `right = 3` `middle = 4`

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




0 ≤ 3: round 2. middle = (0 + 3) / 2 = **1**, list[1] = 13.

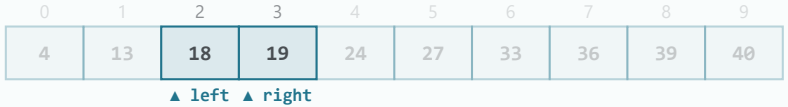
Step 4 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
▶      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

searchnum = 18 | left = 0 | right = 3 | middle = 1

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




13 < 18 → COMPARE = −1: discard the left half, **left = 2**.

Step 5 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
▶        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

`searchnum = 18` `left = 2` `right = 3` `middle = 1`

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




2 ≤ 3: round 3. middle = (2 + 3) / 2 = **2**, list[2] = 18.

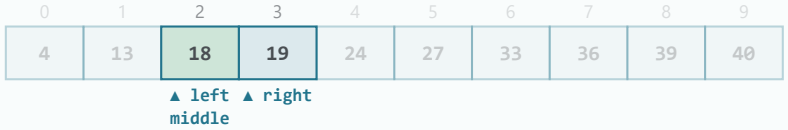
Step 6 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
▶      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

`searchnum = 18` `left = 2` `right = 3` `middle = 2`

# Code Trace: Iterative `binsearch(list, 18, 0, 9)`




COMPARE = 0 → **return 2**. Found in 3 rounds.

Step 7 / 7

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
▶        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

**VARIABLES**

`searchnum = 18` `left = 2`

`right = 3` `middle = 2`

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




`binsearch(list, 25, 0, 9)` : the range is the whole array.

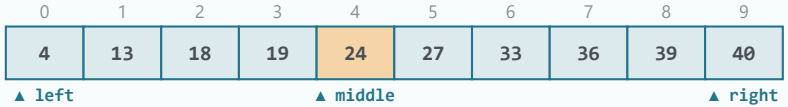
Step 1 / 8

```
▶ int binsearch(int list[], int searchnum,
▶               int left, int right) {
3   while (left <= right) {
4     int middle = (left + right) / 2;
5     switch (COMPARE(list[middle], searchnum)) {
6       case -1: left = middle + 1; break;
7       case 0: return middle;
8       case 1: right = middle - 1;
9     }
10  }
11  return -1;
12 }
```

VARIABLES

`searchnum = 25` `left = 0` `right = 9`

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




0 ≤ 9: round 1. middle = (0 + 9) / 2 = **4**, list[4] = 24.

Step 2 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
▶      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

searchnum = 25 | left = 0 | right = 9 | middle = 4

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




24 < 25 → COMPARE = −1: discard the left half, **left = 5**.

Step 3 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
▶        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

`searchnum = 25` `left = 5` `right = 9` `middle = 4`

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




5 ≤ 9: round 2. middle = (5 + 9) / 2 = **7**, list[7] = 36.

Step 4 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
▶      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

searchnum = **25** | left = **5** | right = **9** | middle = **7**

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




36 > 25 → COMPARE = 1: discard the right half, **right = 6**.

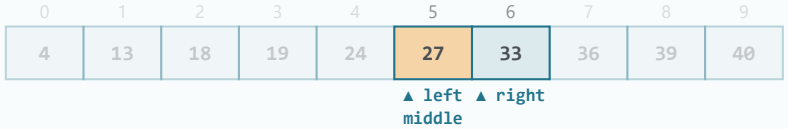
Step 5 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
▶        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

`searchnum = 25` `left = 5` `right = 6` `middle = 7`

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




5 ≤ 6: round 3. middle = (5 + 6) / 2 = **5**, list[5] = 27.

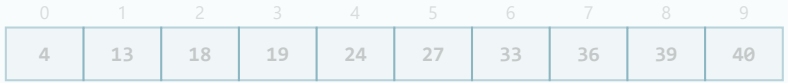
Step 6 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
▶      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

**VARIABLES**

`searchnum = 25` `left = 5` `right = 6` `middle = 5`

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




left = 5 > right = 4: empty range

27 > 25 → COMPARE = 1: discard the right half, **right = 4**.

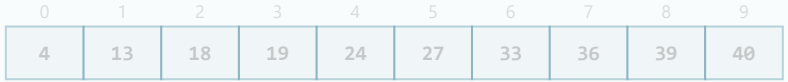
Step 7 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
3    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
▶        case 1: right = middle - 1;
9      }
10   }
11   return -1;
12 }
```

VARIABLES

searchnum = 25 | left = 5 | right = 4 | middle = 5

# Code Trace: `binsearch(list, 25, 0, 9)` → −1




left = 5 > right = 4: empty range

left > right: the loop ends → **return −1** (3 rounds).

Step 8 / 8

```
1  int binsearch(int list[], int searchnum,
2                int left, int right) {
▶    while (left <= right) {
4      int middle = (left + right) / 2;
5      switch (COMPARE(list[middle], searchnum)) {
6        case -1: left = middle + 1; break;
7        case 0: return middle;
8        case 1: right = middle - 1;
9      }
10   }
▶    return -1;
12 }
```

VARIABLES

`searchnum = 25` `left = 5` `right = 4`

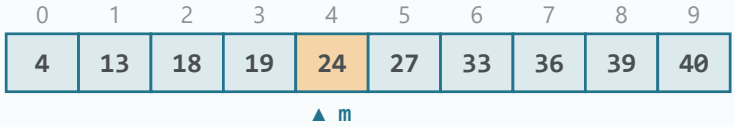
# Recursive Binary Search

```
int binsearch(int list[], int x, int l, int r) {
  int m;
  if (l > r) return -1;          /* empty range */
  m = (l + r) / 2;
  switch (COMPARE(list[m], x)) {
    case -1: return binsearch(list, x, m + 1, r);
    case 0:  return m;
    case 1:  return binsearch(list, x, l, m - 1);
  }
  return -1;                     /* unreachable */
}
```

- Instead of looping, it **calls itself** on a smaller range.
- Base cases: found ( `case 0` ) or the range is empty ( `l > r` ). For brevity x = searchnum, l/r = left/right, m = middle.
- Both recursive calls are **tail calls**: they return the result unchanged.

A common mistake: writing `while` instead of `if` in the recursive version. Every branch exits with `return` , so the loop runs once and the result is correct, but it is misleading: it should be `if` .

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




`l=0 r=9 m=4`

New frame: l = 0, r = 9 → m = **4** (list[4] = 24).

Step 1 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
▶    if (l > r) return -1;          /* empty range */
▶    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

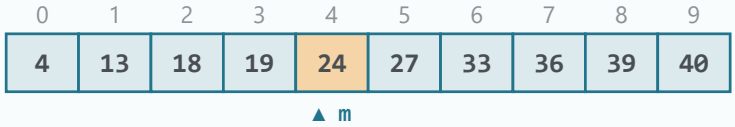
**VARIABLES**

`l = 0` `r = 9` `m = 4`

**CALL STACK**

- `binsearch(0, 9)`
- `main()`

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




`l=0 r=9 m=4`

24 > 18 → case 1: call binsearch(list, 18, 0, 3).

Step 2 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
3    if (l > r) return -1;          /* empty range */
4    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
▶      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 0` `r = 9` `m = 4`

**CALL STACK**

- `binsearch(0, 9)`
- `main()`

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




`l=0 r=3 m=1`

`l=0 r=9 m=4`

New frame: l = 0, r = 3 → m = **1** (list[1] = 13).

Step 3 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
▶    if (l > r) return -1;          /* empty range */
▶    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 0` `r = 3` `m = 1`

**CALL STACK**

- **binsearch(0, 3)**
- binsearch(0, 9)
- main()

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




l=0 r=3 m=1

l=0 r=9 m=4

13 < 18 → case −1: call binsearch(list, 18, 2, 3).

Step 4 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
3    if (l > r) return -1;          /* empty range */
4    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
▶      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 0` `r = 3` `m = 1`

**CALL STACK**

- **binsearch(0, 3)**
- binsearch(0, 9)
- main()

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




l=2 r=3 m=2

l=0 r=3 m=1

l=0 r=9 m=4

New frame: l = 2, r = 3 → m = **2** (list[2] = 18).

Step 5 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
▶    if (l > r) return -1;          /* empty range */
▶    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 2` `r = 3` `m = 2`

**CALL STACK**

- **binsearch(2, 3)**
- binsearch(0, 3)
- binsearch(0, 9)
- main()

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




l=2 r=3 m=2 →2

l=0 r=3 m=1

l=0 r=9 m=4

list[2] = 18: **case 0** → **return 2**.

Step 6 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
3    if (l > r) return -1;          /* empty range */
4    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
▶      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 2` `r = 3` `m = 2`

**CALL STACK**

- **binsearch(2, 3)**
- binsearch(0, 3)
- binsearch(0, 9)
- main()

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




l=0 r=3 m=1 →2

l=0 r=9 m=4

The returned 2 is passed up unchanged (the case −1 frame is popped).

Step 7 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
3    if (l > r) return -1;          /* empty range */
4    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
▶      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
8      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 0` `r = 3` `m = 1`

**CALL STACK**

- **binsearch(0, 3)**
- binsearch(0, 9)
- main()

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




`l=0 r=9 m=4 →2`

The returned 2 is passed up unchanged (the case 1 frame is popped).

Step 8 / 9

```
1  int binsearch(int list[], int x, int l, int r) {
2    int m;
3    if (l > r) return -1;          /* empty range */
4    m = (l + r) / 2;
5    switch (COMPARE(list[m], x)) {
6      case -1: return binsearch(list, x, m + 1, r);
7      case 0:  return m;
▶      case 1:  return binsearch(list, x, l, m - 1);
9    }
10   return -1;                     /* unreachable */
11 }
```

**VARIABLES**

`l = 0` `r = 9` `m = 4`

**CALL STACK**

- `binsearch(0, 9)`
- `main()`

# Code Trace: Recursive `binsearch(list, 18, 0, 9)`




**2** reaches main: “it returns 2 to case 1, which returns it to main.” Depth 3 → O(log n) stack.

Step 9 / 9

```
▶ int binsearch(int list[], int x, int l, int r) {
2   int m;
3   if (l > r) return -1;           /* empty range */
4   m = (l + r) / 2;
5   switch (COMPARE(list[m], x)) {
6     case -1: return binsearch(list, x, m + 1, r);
7     case 0:  return m;
8     case 1:  return binsearch(list, x, l, m - 1);
9   }
10  return -1;                      /* unreachable */
11 }
```

**VARIABLES**

`result = 2`

**CALL STACK**

`main()`

# Complexity of Binary Search

Each round **at least halves** the range: n → n/2 → n/4 → … → 1.

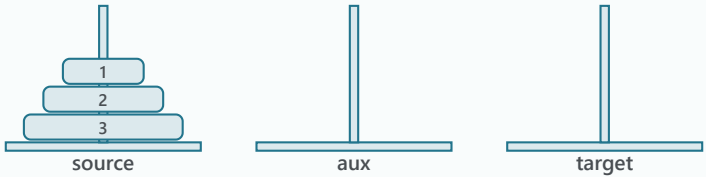
$$T(n) = T(n/2) + c \Rightarrow T(n) = O(\log n)$$

- At most $\lfloor \log_2 n \rfloor + 1$ rounds.
- Extra space for the iterative version: **O(1)**.
- Recursive version: one frame per level → **O(log n)**.
- Precondition: the array must be **sorted** and indexable in O(1) (useless on a linked list).

| n | max rounds | linear search |
|---|---|---|
| 10 | 4 | 10 |
| 1,000 | 10 | 1,000 |
| 1,000,000 | 20 | 1,000,000 |
| 1,000,000,000 | 30 | 1,000,000,000 |

In the sample array (n = 10) at most 4 rounds: 3 were enough for 18.

# The Towers of Hanoi Problem




The source tower holds n disks, each larger than the one above it. Goal: move them all to the **target** tower, using the **auxiliary** tower.

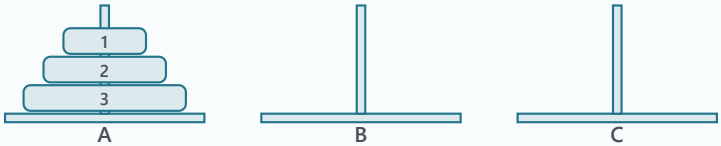
1. Move **one** disk at a time.
2. Never place a disk on top of a **smaller** disk.

**Recursive solution:** to move the largest disk to the target, there must be a moment when the n−1 disks above it are on the auxiliary tower: n−1 disks source→aux, the largest disk source→target, n−1 disks aux→target. **h(n) = h(n−1) + 1 + h(n−1)**.

**INSTRUCTOR** *"Some problems can only be solved recursively."*

In theory any recursion can become a loop with an explicit stack; the point is that recursion is the only **natural** way to express such problems.

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 3: first move the top 2 disks A → B (via C).

Step 1 / 11

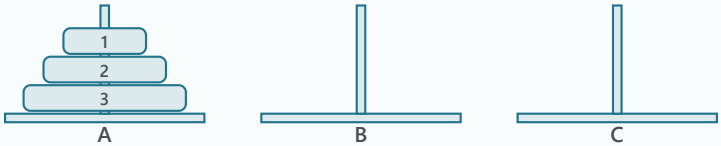
```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
▶    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
▶      hanoi(n - 1, from, to, via);
7      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

CALL STACK

- `hanoi(3,A,B,C)`
- `main()`

OUTPUT

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 2: first move the top disk A → C (via B).

Step 2 / 11

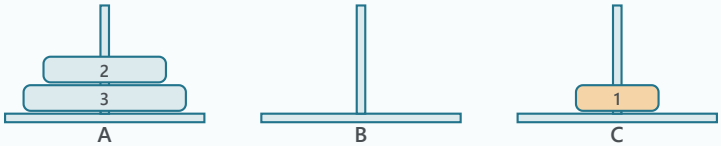
```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
▶    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
▶      hanoi(n - 1, from, to, via);
7      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

CALL STACK

- `hanoi(2,A,C,B)`
- `hanoi(3,A,B,C)`
- `main()`

OUTPUT

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 1: base case. Disk 1: **A** → **C**.

Step 3 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
▶    if (n == 1)
▶      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
7      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

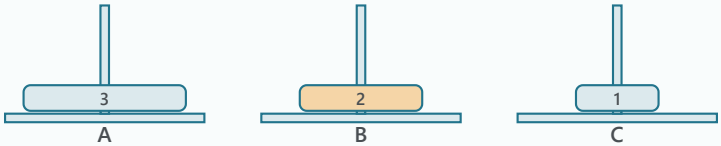
CALL STACK

- **hanoi(1,A,B,C)**
- hanoi(2,A,C,B)
- hanoi(3,A,B,C)
- main()

OUTPUT

```
A -> C
```

# Code Trace: `hanoi(3, "A", "B", "C")`




Back again; the largest disk 2: **A** → **B**. Then 1 disk C → B.

Step 4 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
3    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
▶      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

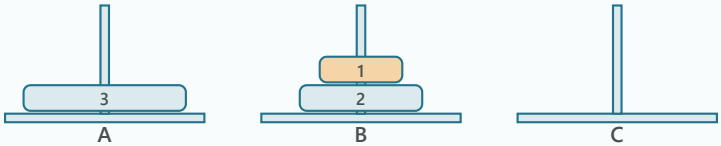
CALL STACK

- **hanoi(2,A,C,B)**
- hanoi(3,A,B,C)
- main()

OUTPUT

```
A -> C
A -> B
```

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 1: base case. Disk 1: **C** → **B**.

Step 5 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
   if (n == 1)
     printf("%s -> %s\n", from, to);
5  else {
6    hanoi(n - 1, from, to, via);
7    printf("%s -> %s\n", from, to);
8    hanoi(n - 1, via, from, to);
9  }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

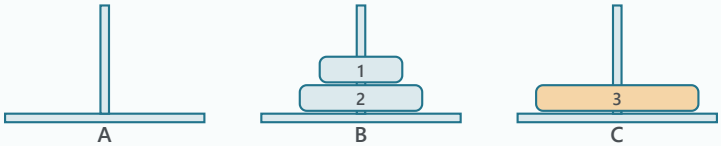
CALL STACK

- **hanoi(1,C,A,B)**
- hanoi(2,A,C,B)
- hanoi(3,A,B,C)
- main()

OUTPUT

```
A -> C
A -> B
C -> B
```

# Code Trace: `hanoi(3, "A", "B", "C")`




Back again; the largest disk 3: **A** → **C**. Then 2 disks B → C.

Step 6 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
3    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
▶      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

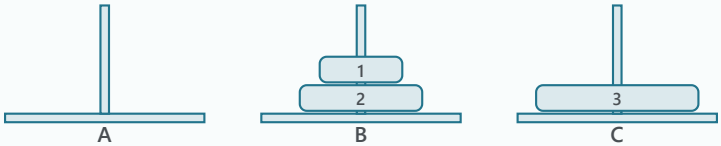
CALL STACK

- **hanoi(3,A,B,C)**
- main()

OUTPUT

```
A -> C
A -> B
C -> B
A -> C
```

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 2: first move the top disk B → A (via C).

Step 7 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
   if (n == 1)
4    printf("%s -> %s\n", from, to);
5  else {
     hanoi(n - 1, from, to, via);
7    printf("%s -> %s\n", from, to);
8    hanoi(n - 1, via, from, to);
9  }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

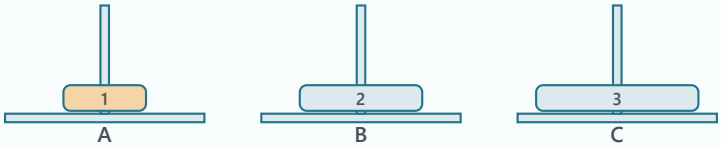
**CALL STACK**

- **`hanoi(2,B,A,C)`**
- `hanoi(3,A,B,C)`
- `main()`

**OUTPUT**

```
A -> C
A -> B
C -> B
A -> C
```

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 1: base case. Disk 1: **B** → **A**.

Step 8 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
   if (n == 1)
     printf("%s -> %s\n", from, to);
5  else {
6    hanoi(n - 1, from, to, via);
7    printf("%s -> %s\n", from, to);
8    hanoi(n - 1, via, from, to);
9  }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

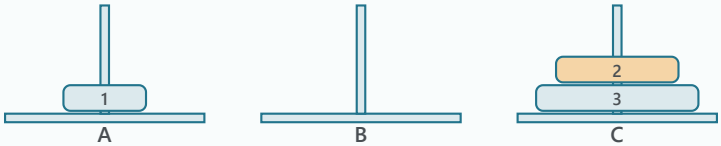
CALL STACK

- **hanoi(1,B,C,A)**
- hanoi(2,B,A,C)
- hanoi(3,A,B,C)
- main()

OUTPUT

```
A -> C
A -> B
C -> B
A -> C
B -> A
```

# Code Trace: `hanoi(3, "A", "B", "C")`




Back again; the largest disk 2: **B** → **C**. Then 1 disk A → C.



```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
3    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
▶      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

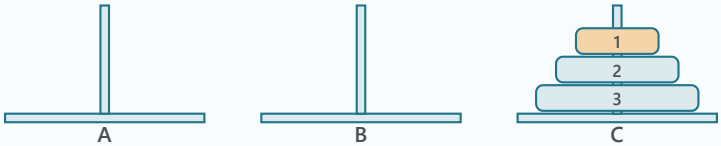
CALL STACK

- **hanoi(2,B,A,C)**
- hanoi(3,A,B,C)
- main()

OUTPUT

```
A -> C      A -> C
A -> B      B -> A
C -> B      B -> C
```

# Code Trace: `hanoi(3, "A", "B", "C")`




n = 1: base case. Disk 1: **A** → **C**.

Step 10 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
▶    if (n == 1)
▶      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
7      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
11 /* main: hanoi(3, "A", "B", "C"); */
```

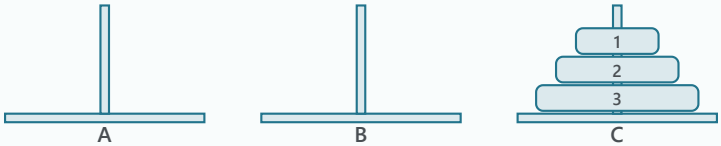
**CALL STACK**

- **`hanoi(1,A,B,C)`**
- `hanoi(2,B,A,C)`
- `hanoi(3,A,B,C)`
- `main()`

**OUTPUT**

```
A -> C     A -> C     A -> C
A -> B     B -> A
C -> B     B -> C
```

# Code Trace: `hanoi(3, "A", "B", "C")`




Done: 7 = $2^3$ − 1 moves, 7 calls. The stack was at most n = 3 frames deep.

Step 11 / 11

```
1  void hanoi(int n, char *from, char *via,
2             char *to) {
3    if (n == 1)
4      printf("%s -> %s\n", from, to);
5    else {
6      hanoi(n - 1, from, to, via);
7      printf("%s -> %s\n", from, to);
8      hanoi(n - 1, via, from, to);
9    }
10 }
▶ /* main: hanoi(3, "A", "B", "C"); */
```

**CALL STACK**

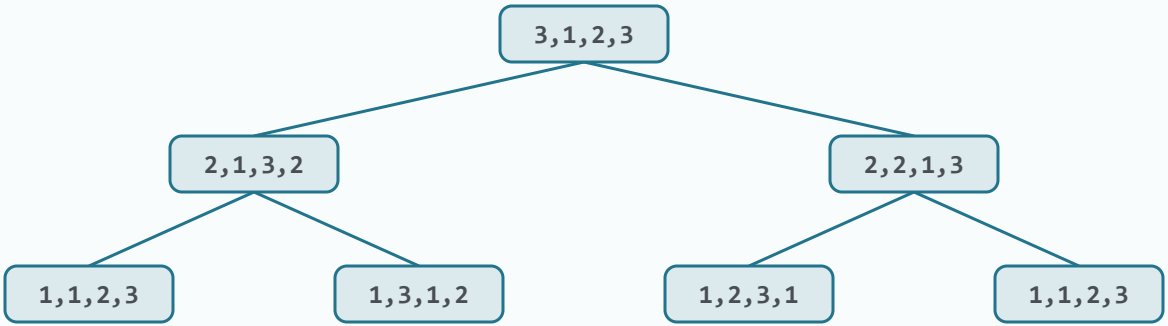
`main()`

**OUTPUT**

```
A -> C      A -> C      A -> C
A -> B      B -> A
C -> B      B -> C
```

# Homework: Stack Frames of `hanoi(3, "1", "2", "3")`

Each box is the frame of one call: `n, from, via, to`. The left child is the first recursive call, the right child the second. At any moment the stack holds the path from the root to the current node.




▶ **Order of moves (inorder traversal)**

# Solving the Recurrence h(n) = 2·h(n−1) + 1

$$
\begin{aligned}
h(1) &= 1 \\
h(n) &= 2 \cdot h(n-1) + 1 \\
&= 4 \cdot h(n-2) + 2 + 1 \\
&= 8 \cdot h(n-3) + 4 + 2 + 1 \\
&= \dots \\
&= 2^{n-1} \cdot h(1) + 2^{n-2} + \dots + 1 \\
&= 2^n - 1
\end{aligned}
$$

| n | moves |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |
| 10 | 1,023 |
| 20 | 1,048,575 |
| 64 | 18,446,744,073,709,551,615 |

64 disks: $2^{64} - 1 \approx 1.8 \cdot 10^{19}$ moves. At one move per second ≈ **585 billion years**. The algorithm uses $O(2^n)$ time and $O(n)$ stack, and nothing does better: $2^n - 1$ moves are necessary and sufficient.

# The Permutation Problem [O(n!)]

We want to print **every arrangement** of a set of n ≥ 1 elements: there are n! of them. {a, b, c} → abc, acb, bac, bca, cab, cba.

### The recursive idea

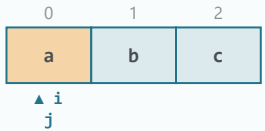
The permutations of {a, b, c, d} are:

- a followed by the permutations of {b, c, d}
- b followed by the permutations of {a, c, d}
- c followed by the permutations of {a, b, d}
- d followed by the permutations of {a, b, c}

### Implementation by swapping

Bring each element in turn to position i with a **swap**, permute the remaining n−i−1 positions recursively, then **swap back** to restore the array (backtracking!).

# Code Trace: `perm("abc", 0, 3)`




i = 0, j = 0: swap with itself → **abc**; call perm(i = 1).

Step 1 / 11

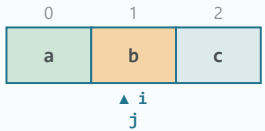
```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

# Code Trace: `perm("abc", 0, 3)`




i = 1, j = 1: swap with itself → **abc**; call perm(i = 2).

Step 2 / 11

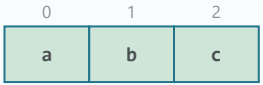
```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

# Code Trace: `perm("abc", 0, 3)`




i = n: an arrangement is complete → print **abc**.

Step 3 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

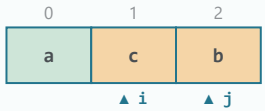
CALL STACK

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
```

# Code Trace: `perm("abc", 0, 3)`




i = 1, j = 2: list[1] ↔ list[2] → **acb**; call perm(i = 2).

Step 4 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

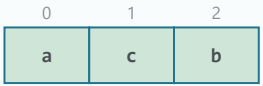
CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
```

# Code Trace: `perm("abc", 0, 3)`




i = n: an arrangement is complete → print **acb**.

Step 5 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

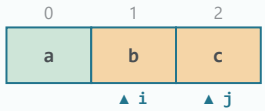
CALL STACK

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
```

# Code Trace: `perm("abc", 0, 3)`




Swap back: list[1] ↔ list[2] → **abc** (restored).

Step 6 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
▶        SWAP(list[i], list[j], temp);
11     }
12 }
```

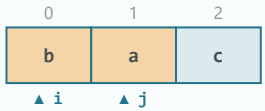
CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

```
abc
acb
```

# Code Trace: `perm("abc", 0, 3)`




i = 0, j = 1: list[0] ↔ list[1] → **bac**; call perm(i = 1).

Step 7 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

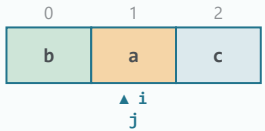
CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

```
abc
acb
```

# Code Trace: `perm("abc", 0, 3)`




i = 1, j = 1: swap with itself → **bac**; call perm(i = 2).

Step 8 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

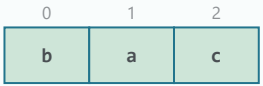
CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
```

# Code Trace: `perm("abc", 0, 3)`




i = n: an arrangement is complete → print **bac**.

Step 9 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

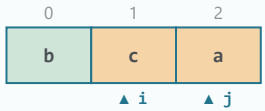
CALL STACK

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
bac
```

# Code Trace: `perm("abc", 0, 3)`




i = 1, j = 2: list[1] ↔ list[2] → **bca**; call perm(i = 2).

Step 10 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

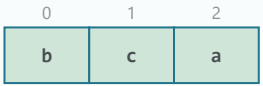
CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
bac
```

# Code Trace: `perm("abc", 0, 3)`




i = n: an arrangement is complete → print **bca**.

Step 11 / 11

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

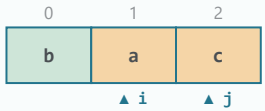
**CALL STACK**

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

**OUTPUT**

```
abc          bca
acb
bac
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




Swap back: list[1] ↔ list[2] → **bac** (restored).

Step 1 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
▶        SWAP(list[i], list[j], temp);
11     }
12 }
```

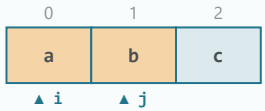
CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

```
abc          bca
acb
bac
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




Swap back: list[0] ↔ list[1] → **abc** (restored).

Step 2 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
▶        SWAP(list[i], list[j], temp);
11     }
12 }
```

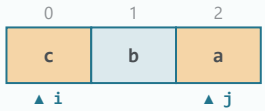
CALL STACK

- **perm(i=0)**
- main()

OUTPUT

```
abc          bca
acb
bac
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




i = 0, j = 2: list[0] ↔ list[2] → **cba**; call perm(i = 1).

Step 3 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

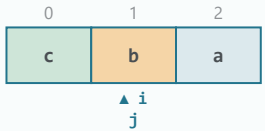
CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

```
abc
acb
bac
bca
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




i = 1, j = 1: swap with itself → **cba**; call perm(i = 2).

Step 4 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

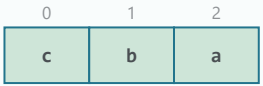
CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc         bca
acb
bac
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




i = n: an arrangement is complete → print **cba**.

Step 5 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

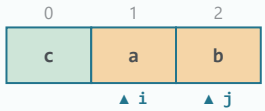
CALL STACK

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
bac
bca
cba
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




i = 1, j = 2: list[1] ↔ list[2] → **cab**; call perm(i = 2).

Step 6 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
▶      for (j = i; j < n; j++) {
▶        SWAP(list[i], list[j], temp);
▶        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

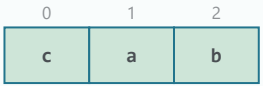
CALL STACK

- **perm(i=2)**
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc
acb
bac
bca
cba
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




i = n: an arrangement is complete → print **cab**.

Step 7 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
▶    if (i == n) {                    /* one arrangement */
▶      for (j = 0; j < n; j++) printf("%c", list[j]);
▶      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
10       SWAP(list[i], list[j], temp);
11     }
12 }
```

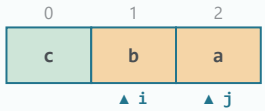
CALL STACK

- **perm(i=3)**
- perm(i=2)
- perm(i=1)
- perm(i=0)
- main()

OUTPUT

```
abc        bca
acb        cba
bac        cab
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




Swap back: list[1] ↔ list[2] → **cba** (restored).

Step 8 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
▶        SWAP(list[i], list[j], temp);
11     }
12 }
```

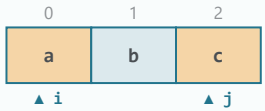
CALL STACK

- **perm(i=1)**
- perm(i=0)
- main()

OUTPUT

```
abc         bca
acb         cba
bac         cab
```

# Code Trace: `perm("abc", 0, 3)` (Continued)




Swap back: list[0] ↔ list[2] → **abc** (restored).

Step 9 / 9

```
1  void perm(char *list, int i, int n) {
2    int j, temp;
3    if (i == n) {                    /* one arrangement */
4      for (j = 0; j < n; j++) printf("%c", list[j]);
5      printf("\n");
6    } else
7      for (j = i; j < n; j++) {
8        SWAP(list[i], list[j], temp);
9        perm(list, i + 1, n);
▶        SWAP(list[i], list[j], temp);
11     }
12 }
```

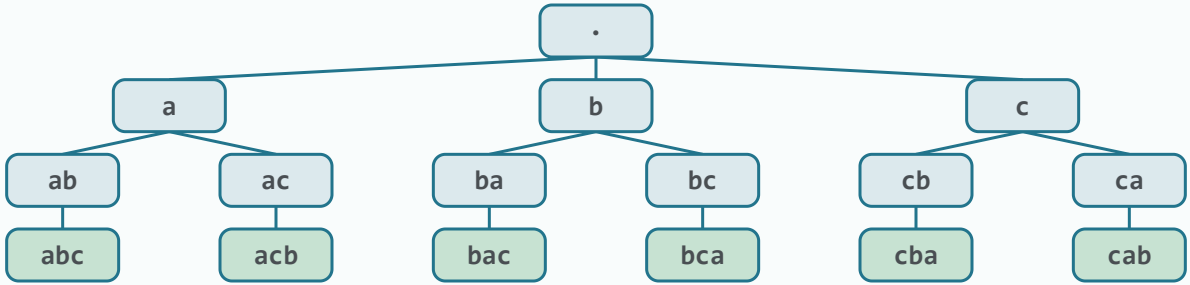
**CALL STACK**

- **perm(i=0)**
- main()

**OUTPUT**

```
abc        bca
acb        cba
bac        cab
```

# Recursion Tree and Complexity




Root: no fixed prefix · each level fixes one position · leaves: the 6 arrangements in output order

### Number of calls

1 + n + n(n−1) + … + n! calls; n! leaves.

### Time

Each leaf prints n characters → **O(n·n!)**. The common shorthand O(n!) misses this factor of n.

### Why swap back?

Sibling branches must see the same starting array; the second SWAP repairs it.

# Brute Force and Backtracking

### Brute force

Generate every candidate and test each one. It relies on raw computing power; there may be $2^n$, n! … candidates.

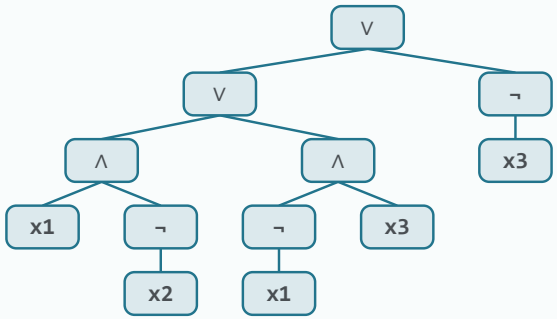
### Backtracking

Build the solution **step by step**; as soon as a partial solution breaks a rule, **prune** that branch, undo the last choice and try the next.

```
void backtrack(partial_solution) {
  if (is_complete) { print; return; }
  for (each candidate choice)
    if (satisfies_rules) { choose; backtrack(...); undo_choice; }
}
```

**INSTRUCTOR** *"Keep looking for a better solution: when solving problems with data structures, never be fully sure of your choice. Keep writing, keep discussing; tell yourself, 'There is a better method, we just haven't found it.'"*

# The Satisfiability (SAT) Problem




(x1 ∧ ¬x2) ∨ (¬x1 ∧ x3) ∨ ¬x3

x1, x2, … are Boolean variables taking true/false; ∧, ∨, ¬ are the operators. **Question:** is there an assignment that makes the formula **true**?

- 3 variables → $2^3$ = 8 possible assignments: {F,F,F}, {F,F,T}, …
- n variables → $2^n$ cases; brute force is **$O(2^n)$**. Even split over m processors it is $O(2^n/m)$: still exponential.
- SAT is NP-complete; **no** polynomial-time algorithm is known. 2-SAT is solvable in polynomial time, 3-SAT is NP-complete.

**INSTRUCTOR** *"I wish you had taken a course on the theory of computation too; that is where problems get classified."*

# SAT: Evaluation by Postorder Traversal

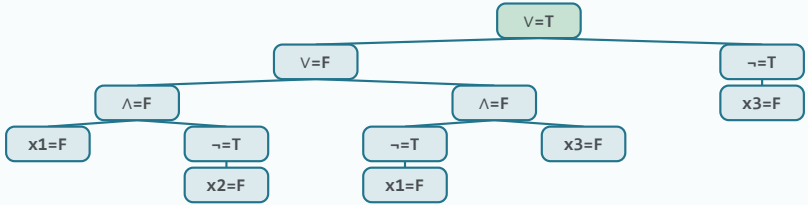
```
typedef enum { L_NOT, L_AND, L_OR, L_TRUE, L_FALSE } logical;
typedef struct node *tree_ptr;
struct node {
  tree_ptr left_child, right_child;
  logical data;
  short value;
};

void post_order_eval(tree_ptr t) {
  if (t) {
    post_order_eval(t->left_child);
    post_order_eval(t->right_child);
    switch (t->data) {
      case L_NOT: t->value = !t->right_child->value; break;
      case L_AND: t->value = t->left_child->value
                          && t->right_child->value; break;
      case L_OR:  t->value = t->left_child->value
                          || t->right_child->value; break;
      case L_TRUE:  t->value = 1; break;
      case L_FALSE: t->value = 0; break;
    }
  }
}
```

- **Postorder**: both children first, then the node itself.
- A ¬ node keeps its single operand in the **right** child.
- Leaves: for each assignment the variables' values are stored as `L_TRUE` / `L_FALSE`.

Names like `enum { not, and, or, true, false }` clash with the C99 `<stdbool.h>` macros and C23 keywords, and `typedef struct node {…};` is a typedef that names nothing. Here we use prefixed names that do not clash.

# Code Trace: Trying All $2^3$ Assignments




Assignment 000: ¬x3 = T → root **true: print**.

Step 1 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {    /* 2^3 assignments */
▶    x1 = (m >> 2) & 1;
▶    x2 = (m >> 1) & 1;
▶    x3 = m & 1;
▶    set_leaves(root, x1, x2, x3);
▶    post_order_eval(root);
▶    if (root->value)
▶      printf("%d%d%d\n", x1, x2, x3);
10 }
```

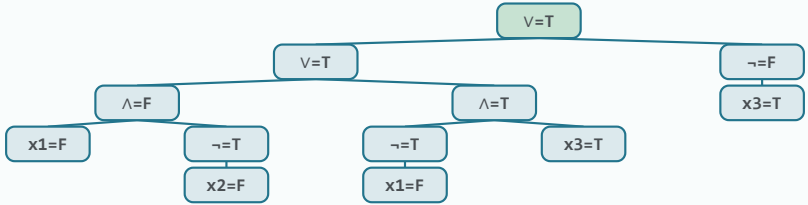
**VARIABLES**

`m = 0` `x1 = 0` `x2 = 0` `x3 = 0` `root = T`

**OUTPUT**

```
000
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 001: ¬x1 ∧ x3 = T → root **true: print**.

Step 2 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {    /* 2^3 assignments */
     x1 = (m >> 2) & 1;
     x2 = (m >> 1) & 1;
     x3 = m & 1;
     set_leaves(root, x1, x2, x3);
     post_order_eval(root);
     if (root->value)
       printf("%d%d%d\n", x1, x2, x3);
10 }
```

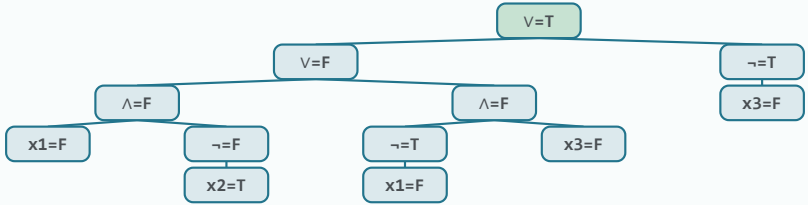
**VARIABLES**

`m = 1` `x1 = 0` `x2 = 0` `x3 = 1` `root = T`

**OUTPUT**

```
000
001
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 010: ¬x3 = T → root **true: print**.

Step 3 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {     /* 2^3 assignments */
▶    x1 = (m >> 2) & 1;
▶    x2 = (m >> 1) & 1;
▶    x3 = m & 1;
▶    set_leaves(root, x1, x2, x3);
▶    post_order_eval(root);
▶    if (root->value)
▶      printf("%d%d%d\n", x1, x2, x3);
10 }
```

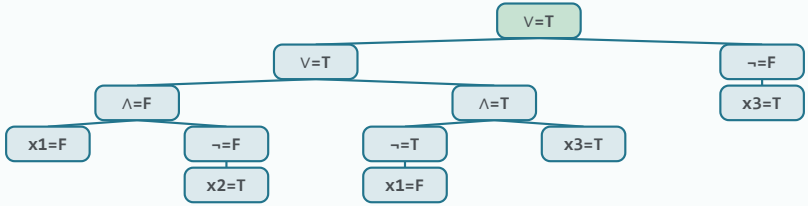
**VARIABLES**

`m = 2` `x1 = 0` `x2 = 1` `x3 = 0` `root = T`

**OUTPUT**

```
000
001
010
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 011: ¬x1 ∧ x3 = T → root **true: print**.

Step 4 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {    /* 2^3 assignments */
     x1 = (m >> 2) & 1;
     x2 = (m >> 1) & 1;
     x3 = m & 1;
     set_leaves(root, x1, x2, x3);
     post_order_eval(root);
     if (root->value)
       printf("%d%d%d\n", x1, x2, x3);
10 }
```

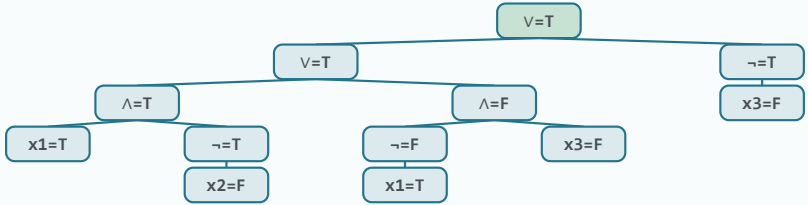
**VARIABLES**

`m = 3` `x1 = 0` `x2 = 1` `x3 = 1` `root = T`

**OUTPUT**

```
000        011
001
010
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 100: ¬x3 = T → root **true: print**.

Step 5 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {    /* 2^3 assignments */
     x1 = (m >> 2) & 1;
     x2 = (m >> 1) & 1;
     x3 = m & 1;
     set_leaves(root, x1, x2, x3);
     post_order_eval(root);
     if (root->value)
       printf("%d%d%d\n", x1, x2, x3);
10 }
```

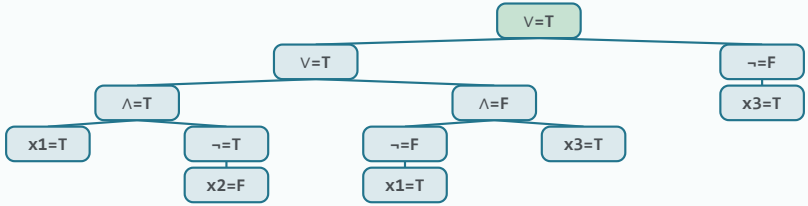
**VARIABLES**

`m = 4` `x1 = 1` `x2 = 0` `x3 = 0` `root = T`

**OUTPUT**

```
000
001
010
011
100
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 101: x1 ∧ ¬x2 = T → root **true: print**.

Step 6 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {    /* 2^3 assignments */
     x1 = (m >> 2) & 1;
     x2 = (m >> 1) & 1;
     x3 = m & 1;
     set_leaves(root, x1, x2, x3);
     post_order_eval(root);
     if (root->value)
       printf("%d%d%d\n", x1, x2, x3);
10 }
```

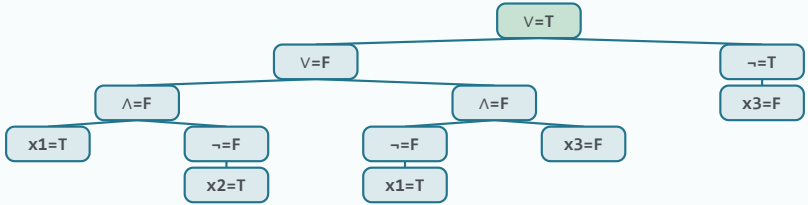
**VARIABLES**

`m = 5` `x1 = 1` `x2 = 0` `x3 = 1` `root = T`

**OUTPUT**

```
000
001
010
011
100
101
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 110: ¬x3 = T → root **true: print**.

Step 7 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {     /* 2^3 assignments */
     x1 = (m >> 2) & 1;
     x2 = (m >> 1) & 1;
     x3 = m & 1;
     set_leaves(root, x1, x2, x3);
     post_order_eval(root);
     if (root->value)
       printf("%d%d%d\n", x1, x2, x3);
10 }
```

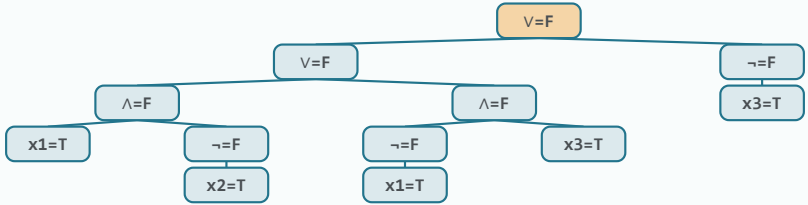
**VARIABLES**

`m = 6` `x1 = 1` `x2 = 1` `x3 = 0` `root = T`

**OUTPUT**

```
000        011        110
001        100
010        101
```

# Code Trace: Trying All $2^3$ Assignments




Assignment 111: no term is T → root **false**.

Step 8 / 9

```
1  int m, x1, x2, x3;
2  for (m = 0; m < 8; m++) {     /* 2^3 assignments */
▶    x1 = (m >> 2) & 1;
▶    x2 = (m >> 1) & 1;
▶    x3 = m & 1;
▶    set_leaves(root, x1, x2, x3);
▶    post_order_eval(root);
▶    if (root->value)
9      printf("%d%d%d\n", x1, x2, x3);
10 }
```

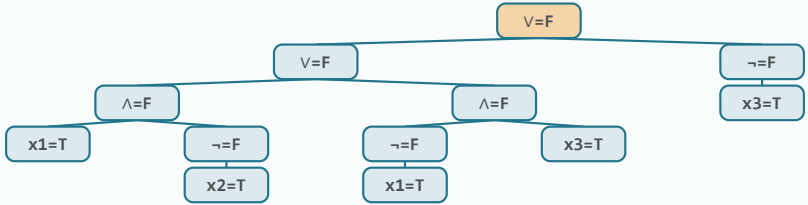
**VARIABLES**

`m = 7` `x1 = 1` `x2 = 1` `x3 = 1` `root = F`

**OUTPUT**

```
000     011     110
001     100
010     101
```

# Code Trace: Trying All $2^3$ Assignments




**7** of the 8 assignments satisfy the formula; only x1 = x2 = x3 = 1 does not. A version that `return`s at the first solution would stop at 000.

Step 9 / 9

```
1  int m, x1, x2, x3;
▶ for (m = 0; m < 8; m++) {    /* 2^3 assignments */
3    x1 = (m >> 2) & 1;
4    x2 = (m >> 1) & 1;
5    x3 = m & 1;
6    set_leaves(root, x1, x2, x3);
7    post_order_eval(root);
8    if (root->value)
9      printf("%d%d%d\n", x1, x2, x3);
10 }
```

**VARIABLES**

`m = 8` `found = 7`

**OUTPUT**

```
000
001
010
011
100
101
110
```

# SAT Variants and Precomputing

### CIRCUIT-SAT

Is there an input assignment that makes the output of a given logic circuit 1?

### Formula SAT

A propositional formula built with ∧, ∨, ¬, as above.

### 3-CNF-SAT

An ∧ of ∨-clauses, each with 3 literals: (x1 ∨ ¬x1 ∨ ¬x2) ∧ (x3 ∨ x2 ∨ x4) ∧ (¬x1 ∨ ¬x3 ∨ ¬x4).

### Precomputing

The formula tree is built **once**; each assignment changes only the leaves. A data structure prepared once for work done many times boosts efficiency a lot.

**INSTRUCTOR** *"Type it into Google and there it is! A fast algorithm: Knuth–Morris–Pratt. Be sure to study greedy algorithms, branch and bound and precomputing in algorithm analysis books!"*

"I haven't failed, I've found 10,000 ways that don't work." — attributed to Thomas Edison.

# The n-Queens Problem

Place n queens on an n×n chessboard so that **no queen attacks another**: no two queens share a row, column or diagonal.

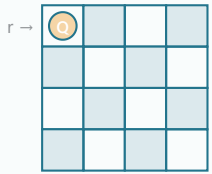
- One queen per row → a solution is an array `q[0..n−1]`.
- Brute force: $n^n$ candidates (one queen per row; 16.8 million for n = 8).
- Backtracking: go row by row; if no column fits, return to the previous row.
- Diagonal test: |q[i] − c| == r − i.

```c
int q[8];  /* q[r]: column of the queen in row r */

int is_safe(int r, int c) {
  int i;
  for (i = 0; i < r; i++)
    if (q[i] == c || abs(q[i] - c) == r - i)
      return 0;          /* same column or diagonal */
  return 1;
}
```

Horowitz–Sahni; Sedgewick. There are 2 solutions for n = 4 and 92 for n = 8.

# Code Trace: `place(0, 4)`




**(0, 0) is safe** → q[0] = 0, go to the next row.

Step 1 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

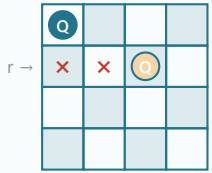
**VARIABLES**

`r = 0` `c = 0` `q = [0, -, -, -]`

**CALL STACK**

- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Columns 0, 1 conflict (×); **(1, 2) is safe** → q[1] = 2, go to the next row.

Step 2 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

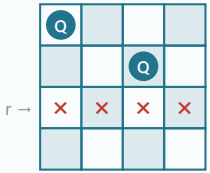
VARIABLES

`r = 1` `c = 2` `q = [0, 2, -, -]`

CALL STACK

- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Columns 0, 1, 2, 3 conflict; no queen fits in row 2 → **return 0**: backtrack, the row above tries its next column.

Step 3 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
4    for (c = 0; c < n; c++)
5      if (is_safe(r, c)) {
6        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
▶    return 0;                    /* backtrack */
10 }
```

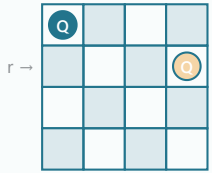
**VARIABLES**

`r = 2` `q = [0, 2, -, -]`

**CALL STACK**

- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




**(1, 3) is safe** → q[1] = 3, go to the next row.

Step 4 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

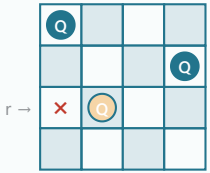
VARIABLES

`r = 1` `c = 3` `q = [0, 3, -, -]`

CALL STACK

- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Column 0 conflicts (×); **(2, 1) is safe** → q[2] = 1, go to the next row.

Step 5 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

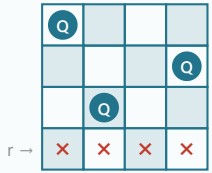
**VARIABLES**

`r = 2` `c = 1` `q = [0, 3, 1, -]`

**CALL STACK**

- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Columns 0, 1, 2, 3 conflict; no queen fits in row 3 → **return 0**: backtrack, the row above tries its next column.

Step 6 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
4    for (c = 0; c < n; c++)
5      if (is_safe(r, c)) {
6        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
▶    return 0;                    /* backtrack */
10 }
```

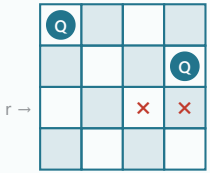
**VARIABLES**

`r = 3` `q = [0, 3, 1, -]`

**CALL STACK**

- `place(3)`
- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Columns 2, 3 conflict; no queen fits in row 2 → **return 0**: backtrack, the row above tries its next column.

Step 7 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
4    for (c = 0; c < n; c++)
5      if (is_safe(r, c)) {
6        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
▶    return 0;                    /* backtrack */
10 }
```

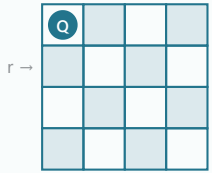
VARIABLES

`r = 2` `q = [0, 3, -, -]`

CALL STACK

- **place(2)**
- place(1)
- place(0)
- main()

# Code Trace: `place(0, 4)`




No columns left; no queen fits in row 1 → **return 0**: backtrack, the row above tries its next column.

Step 8 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
4    for (c = 0; c < n; c++)
5      if (is_safe(r, c)) {
6        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
▶    return 0;                    /* backtrack */
10 }
```

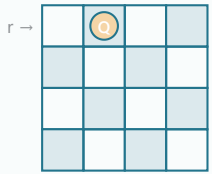
**VARIABLES**

`r = 1` `q = [0, –, –, –]`

**CALL STACK**

- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




**(0, 1) is safe** → q[0] = 1, go to the next row.

Step 9 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

VARIABLES

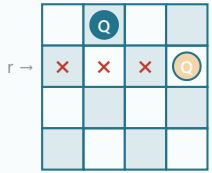
`r = 0` `c = 1` `q = [1, -, -, -]`

CALL STACK

`place(0)`

`main()`

# Code Trace: `place(0, 4)`




Columns 0, 1, 2 conflict (×); **(1, 3) is safe** → q[1] = 3, go to the next row.

Step 10 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

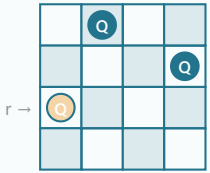
VARIABLES

`r = 1` `c = 3` `q = [1, 3, -, -]`

CALL STACK

- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




**(2, 0) is safe** → q[2] = 0, go to the next row.

Step 11 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

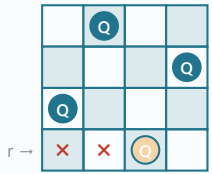
VARIABLES

`r = 2` `c = 0` `q = [1, 3, 0, -]`

CALL STACK

- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`




Columns 0, 1 conflict (×); **(3, 2) is safe** → q[3] = 2, go to the next row.

Step 12 / 13

```
1  int place(int r, int n) {
2    int c;
3    if (r == n) return 1;              /* solved */
▶    for (c = 0; c < n; c++)
▶      if (is_safe(r, c)) {
▶        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

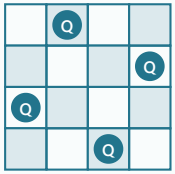
**VARIABLES**

`r = 3` `c = 2` `q = [1, 3, 0, 2]`

**CALL STACK**

- `place(3)`
- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `place(0, 4)`



r = n: four queens placed → **solution q = [1, 3, 0, 2]**. We made 8 placements; brute force has $4^4 = 256$ candidates.

Step 13 / 13

```
1  int place(int r, int n) {
2    int c;
▶    if (r == n) return 1;              /* solved */
4    for (c = 0; c < n; c++)
5      if (is_safe(r, c)) {
6        q[r] = c;
7        if (place(r + 1, n)) return 1;
8      }
9    return 0;                    /* backtrack */
10 }
```

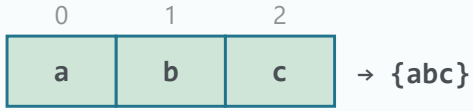
**VARIABLES**

`r = 4` `q = [1, 3, 0, 2]`

**CALL STACK**

- `place(4)`
- `place(3)`
- `place(2)`
- `place(1)`
- `place(0)`
- `main()`

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: take a, take b, take c → **{abc}**.

Step 1 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

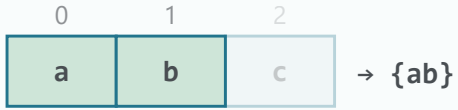
**VARIABLES**

`pick = [1, 1, 1]` `i = 3`

**OUTPUT**

```
{abc}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: take a, take b, skip c → **{ab}**.

Step 2 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

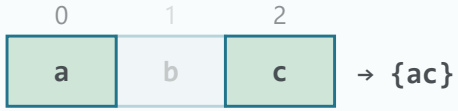
**VARIABLES**

`pick = [1, 1, 0]` `i = 3`

**OUTPUT**

```
{abc}
{ab}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: take a, skip b, take c → **{ac}**.

Step 3 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

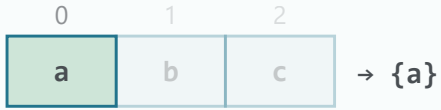
**VARIABLES**

`pick = [1, 0, 1]` `i = 3`

**OUTPUT**

```
{abc}
{ab}
{ac}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: take a, skip b, skip c → **{a}**.

Step 4 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

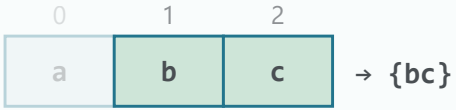
**VARIABLES**

`pick = [1, 0, 0]` `i = 3`

**OUTPUT**

```
{abc}      {a}
{ab}
{ac}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: skip a, take b, take c → **{bc}**.

Step 5 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

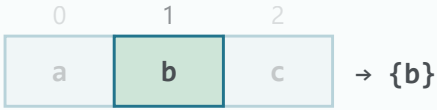
VARIABLES

`pick = [0, 1, 1]` `i = 3`

OUTPUT

```
{abc}       {a}
{ab}        {bc}
{ac}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: skip a, take b, skip c → **{b}**.

Step 6 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

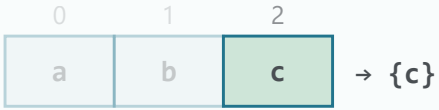
**VARIABLES**

`pick = [0, 1, 0]` `i = 3`

**OUTPUT**

```
{abc}       {a}
{ab}        {bc}
{ac}        {b}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: skip a, skip b, take c → **{c}**.

Step 7 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

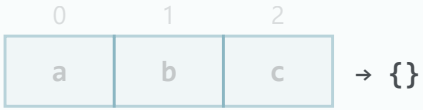
**VARIABLES**

`pick = [0, 0, 1]` `i = 3`

**OUTPUT**

```
{abc}      {a}        {c}
{ab}       {bc}
{ac}       {b}
```

# Code Trace: `subset("abc", 3, 0, pick)`




Choices: skip a, skip b, skip c → **{}**.

Step 8 / 8

```
1  void subset(char *s, int n, int i,
2              int pick[]) {
3    int j;
▶    if (i == n) {                    /* all decided */
▶      printf("{");
▶      for (j = 0; j < n; j++)
▶        if (pick[j]) printf("%c", s[j]);
▶      printf("}\n");
9      return;
10   }
11   pick[i] = 1; subset(s, n, i + 1, pick);
12   pick[i] = 0; subset(s, n, i + 1, pick);
13 }
```

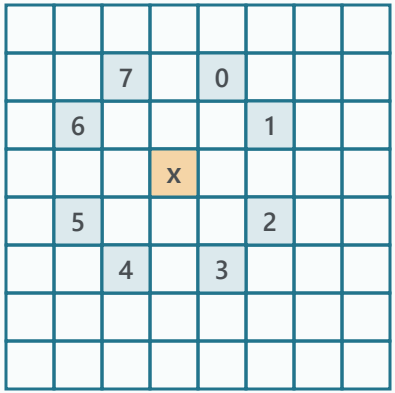
**VARIABLES**

`pick = [0, 0, 0]` `i = 3`

**OUTPUT**

```
{abc}    {a}    {c}
{ab}     {bc}   {}
{ac}     {b}
```

# The Knight's Tour on a Chessboard




The knight must visit every square, each **exactly once**. It moves in an L: 2 across + 1 up/down, or 2 up/down + 1 across.

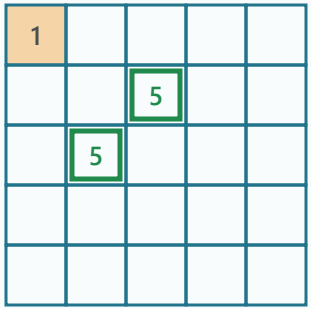
| Move | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Δi | −2 | −1 | +1 | +2 | +2 | +1 | −1 | −2 |
| Δj | +1 | +2 | +2 | +1 | −1 | −2 | −2 | −1 |

- **Backtracking:** try a move; at a dead end, undo the last move and try the next. It always finds a solution, but it is essentially brute force: very slow on 8×8.
- **Heuristic**: a "smart" choice; it finds near-optimal solutions without guarantees (like Minesweeper).

Backtracking is not "rewinding the scenario from the end": it builds the partial solution forward and undoes the last choice at a dead end.

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.



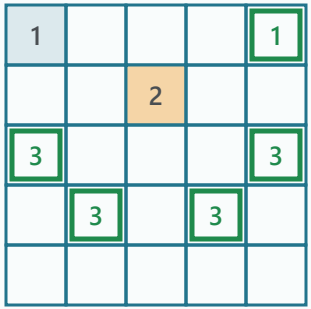

Square 1. Candidates (green = onward moves from there): 5, 5 → smallest positive: **5**, choose (1, 2).

Step 1 / 9

The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.



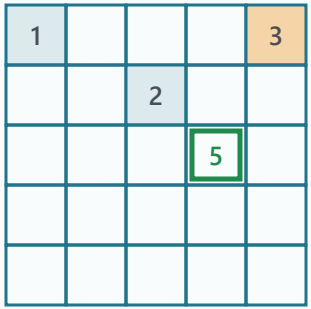

Square 2. Candidates (green = onward moves from there): 1, 3, 3, 3, 3 → smallest positive: **1**, choose (0, 4).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.



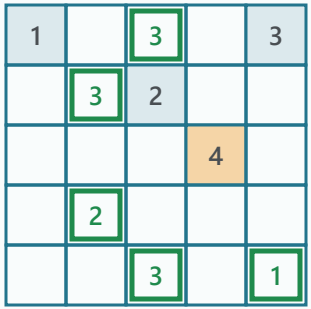

Square 3. Candidates (green = onward moves from there): 5 → smallest positive: **5**, choose (2, 3).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.



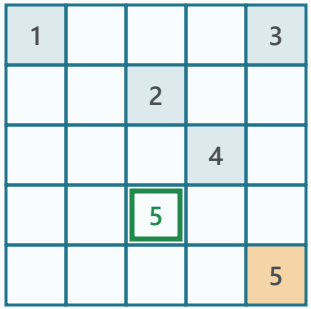

Square 4. Candidates (green = onward moves from there): 1, 3, 2, 3, 3 → smallest positive: **1**, choose (4, 4).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.




Square 5. Candidates (green = onward moves from there): 5 → smallest positive: **5**, choose (3, 2).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.




Square 9. Candidates (green = onward moves from there): 2, 2 → smallest positive: **2**, choose (1, 4).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.




Square 16. Candidates (green = onward moves from there): 1, 4 → smallest positive: **1**, choose (4, 2).



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff’s Heuristic (1823)

“Move the knight to the square with the **fewest onward moves, but not zero**.” Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don’t pile up at the end.

| | | | | |
|---|---|---|---|---|
| 1 | 14 | 9 | 20 | 3 |
| 24 | 19 | 2 | 15 | 10 |
| 13 | 8 | 0 | 4 | 21 |
| 18 | 23 | 6 | 11 | 16 |
| 7 | 12 | 17 | 22 | 5 |

Square 24. Candidates (green = onward moves from there): 0 → the last free square is the only candidate: choose **(2, 2)**.



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

# Warnsdorff's Heuristic (1823)

"Move the knight to the square with the **fewest onward moves, but not zero**." Onward moves: the number of free squares reachable directly from that square. Corners and edges are used up first, so dead ends don't pile up at the end.

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

All 25 squares visited without backtracking once. Warnsdorff looks at only 8 neighbors per step: $O(n^2)$ work in total.



The rule is named after H. C. von Warnsdorff. Ties are broken by the order in the move table (0…7).

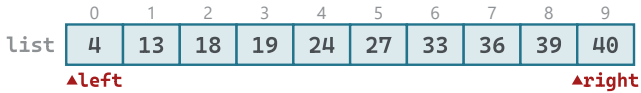
# Lab: Binary Search

Searching for 18

Input (key: numbers in increasing order) `18: 4, 13, 18, 19, 24, 27, 33, 36, 39, 40` Apply

**CODE**

```
 1 int binsearch(int list[], int searchnum,
 2               int left, int right) {
 3   while (left <= right) {
 4     int middle = (left + right) / 2;
 5     switch (COMPARE(list[middle],
 6                     searchnum)) {
 7       case -1: left = middle + 1; break;
 8       case 0: return middle;
 9       case 1: right = middle - 1;
10     }
11   }
12   return -1;
13 }
```




`searchnum 18` `left 0` `right 9` `rounds 0`

**OUTPUT** (none yet)

`binsearch(list, 18, 0, 9)` : the search range is the whole array. In each round the middle element of the range is compared with the key, and the range is halved.

LAB · C CODE STEP BY STEP

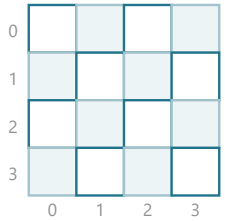
# Lab: n-Queens and Backtracking

4 queens

Input (board size n, 1–8) 4 Apply

CODE

```
 1 int q[8];
 2
 3 int place(int r, int n) {
 4   int c;
 5   if (r == n) return 1;           /* solved */
 6   for (c = 0; c < n; c++)
 7     if (is_safe(r, c)) {
 8       q[r] = c;
 9       if (place(r + 1, n)) return 1;
10     }
11   return 0;                   /* backtrack */
12 }
13 /* is_safe(r, c): 1 if no queen in q[0..r-1]
14    shares its column or a diagonal */
```




n 4 r 0 q [] calls –

OUTPUT (none yet)

`place(0, 4)` : one queen goes in each row. If a row has no safe column, the call returns 0 and the previous row tries its next column (backtracking).

SUMMARY

# Complexity Summary

| Algorithm | Time | Stack / extra space | Note |
|---|---|---|---|
| `fact` (recursive) | $\Theta(n)$ | $O(n)$ | the tail version can get down to $O(1)$ |
| Binary search (iterative) | $O(\log n)$ | $O(1)$ | array must be sorted |
| Binary search (recursive) | $O(\log n)$ | $O(\log n)$ | tail recursion |
| Hanoi | $\Theta(2^n)$ | $O(n)$ | $2^n - 1$ moves |
| Permutations | $\Theta(n \cdot n!)$ | $O(n)$ | $n!$ arrangements |
| Subset generation | $\Theta(n \cdot 2^n)$ | $O(n)$ | $2^n$ subsets |
| SAT (brute force) | $O(2^n \cdot \lvert F \rvert)$ | $O(\lvert F \rvert)$ | $\lvert F \rvert$: size of the formula tree |
| n-queens (backtracking) | $O(n!)$ upper bound | $O(n)$ | far less in practice thanks to pruning |
| Knight's tour (Warnsdorff) | $O(n^2)$ | $O(n^2)$ | heuristic; backtracking is exponential |

SUMMARY

# Common Mistakes

1. Forgetting the base case, or writing a step that never reaches it → stack overflow.

2. Not `return`ing the result of the recursive call: calling `binsearch( ... )` and discarding the value.

3. Using binary search on an unsorted array or a linked list.

4. Returning the **value** found: it can be confused with −1; return the index.

5. Forgetting to undo a choice in backtracking (the second SWAP, `pick[i] = 0`).

6. Running an exponential algorithm on large n because “the machine is fast”.

# Exercises

1. Write a recursive function that finds the largest element of an array. What is the stack depth?
2. At most how many rounds does `binsearch` take on a 1000-element array? And if the key is absent?
3. How many times does hanoi(4, …) call `printf`, and how many times does it call itself?
4. Write the first 6 lines printed by perm("abcd", 0, 4).
5. Find by brute force the assignments that satisfy (x1 ∨ x2) ∧ (¬x1 ∨ ¬x2).
6. Find the first solution of 5-queens by backtracking (the q array).

▶ **Show answers / hints**

SUMMARY

# Chapter Summary

**Recursion**

Base case + shrinking step. Each call is a stack frame; tail recursion can become a loop.

**Binary search**

Halves the range of a sorted array: O(log n). Return the index; −1 means not found.

**Hanoi**

$h(n) = 2h(n-1) + 1 = 2^n - 1$. The call tree = the history of the stack.

**Permutations**

Swap + recurse + swap back: n! lines of output, O(n·n!).

**Backtracking**

Build a partial solution, prune and undo when a rule breaks: n-queens, subsets (SAT was solved by brute force here).

**Heuristics**

Warnsdorff: the square with the fewest onward moves. Never stop looking for a better solution.

SUMMARY

# References


- Horowitz, Sahni, Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 1.2 (recursive algorithms).
- Cormen et al., *Introduction to Algorithms*, 3rd ed.: Chapter 4 (recurrences), Chapter 34 (NP-completeness, SAT).
- Sedgewick, *Algorithms in C*, Chapter 5 (recursion).



← Chapter 2: Arrays and Structures

Course page

Chapter 4: Linked Lists

# The End

## Chapter 3: Recursion, Binary Search and Backtracking

Next chapter: Chapter 4: Linked Lists · Course page