

Repeated work in recursive Virahanka computation

$$V_4 = \underbrace{V_3}_{V_2+V_1} + V_2.$$

The function computes V_2 once inside V_3 and then computes it again.

Call	V_4	V_3	V_2	V_1	V_0
Number of invocations	1	1	2	3	2

- Nine function invocations compute the value $V_4 = 5$.
- A returned answer is not automatically saved for future calls.
- Computing the values in increasing order avoids this repeated work.

Quiz 5

```
int fib(int n)
{
    printf("call %d\n", n);
    if (n <= 1) return n;
    int a = fib(n - 1);
    int b = fib(n - 2);
    return a + b;
}
```

Adapted from `Fibonacci.c` in the folder.

For `fib(3)`:

1. Write the printed argument sequence.
2. Give the returned value.

Quiz 5

```
int fib(int n)
{
    printf("call %d\n", n);
    if (n <= 1) return n;
    int a = fib(n - 1);
    int b = fib(n - 2);
    return a + b;
}
```

Adapted from `Fibonacci.c` in the folder.

For `fib(3)`:

1. Write the printed argument sequence.
2. Give the returned value.

Answer

Printed arguments: 3, 2, 1, 0, 1, each after call. The return value is 2. The two `fib(1)` invocations are separate calls.

Why the trace uses two separate statements

```
int a = fib(n - 1);  
int b = fib(n - 2);  
return a + b;
```

These statements complete the first call before starting the second.

The shorter expression

```
return fib(n - 1) + fib(n - 2);
```

computes the same value, but C does not specify which operand of + is evaluated first. Its printed trace can therefore have a different order.

Virahanka numbers with a loop

```
int virahanka_iter(int n)
{
    int prev = 1, curr = 1;
    for (int i = 2; i <= n; i++) {
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

This version takes $O(n)$ time and $O(1)$ extra space.

Before iteration i :

$$\text{prev} = V_{i-2}, \quad \text{curr} = V_{i-1}.$$

The new value is their sum.

For $n = 4$, successive values of next are 2, 3, 5.

For $n = 0$ or 1 , the loop does not run and the function returns 1.

Recursive GCD: another program from the folder

```
int gcd(int a, int b)
{
    printf("gcd(%d, %d)\n", a, b);
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

Euclid's recurrence

$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b) \quad (b > 0).$$

Base case

$$\text{gcd}(a, 0) = a.$$

Use nonnegative inputs, not both zero. The base case comes before $a \% b$.

Quiz 6

The program calls `gcd(48, 18)`.

1. Write all argument pairs, in the order printed.
2. What does each call return?
3. Why must `b == 0` be checked before `a % b`?

Quiz 6

The program calls `gcd(48, 18)`.

1. Write all argument pairs, in the order printed.
2. What does each call return?
3. Why must `b == 0` be checked before `a % b`?

Answer

`(48, 18)`, `(18, 12)`, `(12, 6)`, `(6, 0)`.

The base call returns 6. Each waiting call returns that same 6.

Taking the remainder with a zero divisor is invalid.

Largest element: a smaller array problem

For the array $[4, 9, 2, 7]$, compare the first element with the largest of the rest.

$$\max(4, 9, 2, 7) = \max(4, \max(9, 2, 7)).$$

- Describe the remaining part using indices i through j , both inclusive.
- The smaller problem uses indices $i + 1$ through j .
- When $i = j$, only one element remains. Return that element.

Each call reduces the number of elements by one.

Largest element: the recursive function

```
int largest(const int a[], int i, int j)
{
    if (i == j) return a[i];
    int rest = largest(a, i + 1, j);
    if (a[i] > rest) return a[i];
    return rest;
}
```

largest(a, 0, 3) examines four elements.

rest stores the largest value in a[i + 1] through a[j].

const means this function does not modify array elements through a.

Precondition: valid indices with $0 \leq i \leq j$.

Largest element: following the returns

For $a = \{4, 9, 2, 7\}$, the calls use first indices 0, 1, 2, 3.

Call	Work after the smaller call returns	Result
<code>largest(a, 3, 3)</code>	Base case: return <code>a[3]</code>	7
<code>largest(a, 2, 3)</code>	Compare <code>a[2] = 2</code> with <code>rest = 7</code>	7
<code>largest(a, 1, 3)</code>	Compare <code>a[1] = 9</code> with <code>rest = 7</code>	9
<code>largest(a, 0, 3)</code>	Compare <code>a[0] = 4</code> with <code>rest = 9</code>	9

The recursive call returns a **value**. The array elements stay unchanged.

Quiz 7

Suppose $a = \{-8, -3, -11\}$.

1. What should `largest(a, 0, 2)` return?
2. A student replaces the base case with `if (i == j) return 0;`. What result does this version give?
3. Why is `return a[i];` the right base case?

Quiz 7

Suppose $a = \{-8, -3, -11\}$.

1. What should `largest(a, 0, 2)` return?
2. A student replaces the base case with `if (i == j) return 0;`. What result does this version give?
3. Why is `return a[i];` the right base case?

Answer

The correct result is -3 . The changed version returns 0, which is not even in the array. For a one-element array, its only element is its largest element.

Optional: splitting the array into two halves

```
int largest_halves(const int a[], int i, int j)
{
    if (i == j) return a[i];
    int mid = i + (j - i) / 2;
    int left = largest_halves(a, i, mid);
    int right = largest_halves(a, mid + 1, j);
    if (left > right) return left;
    return right;
}
```

Adapted from LargestRecursive.c. Both halves must be examined.
For indices 0 to 3, the recursive calls cover [0, 1] and [2, 3].

Splitting changes the depth, not the amount of work

For an array of $n \geq 1$ elements:

	Remove one element	Split into halves
Value comparisons	$n - 1$	$n - 1$
Total time	$O(n)$	$O(n)$
Maximum call depth	$O(n)$	$O(\log n)$

- The halves version has fewer nested calls at one time.
- It still visits every array element.
- A loop also finds the largest in $O(n)$ time, using $O(1)$ extra space.

Quiz 8

```
int sum(int n)
{
    if (n == 0) return 0;
    return n + sum(n);
}
```

The function should return $1 + 2 + \dots + n$ for $n \geq 0$.

What happens for `sum(3)`? Change one expression to fix it.

Quiz 8

```
int sum(int n)
{
    if (n == 0) return 0;
    return n + sum(n);
}
```

The function should return $1 + 2 + \dots + n$ for $n \geq 0$.

What happens for `sum(3)`? Change one expression to fix it.

Answer

Every call receives 3, so the base case is never reached. Replace `sum(n)` with `sum(n - 1)`. Then `sum(3)` returns 6.