Today's class

- Fibonacci numbers
- Search algorithms
  - Linear, Binary
- Towers of Hanoi

Fibonacci numbers

Fibonacci numbers

- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

$f(0) = 0$
$f(1) = 1$
$f(2) = 1$
$f(3) = 2$
$f(4) = 3$
$\vdots$

0 1 1 2 3

Fibonacci numbers

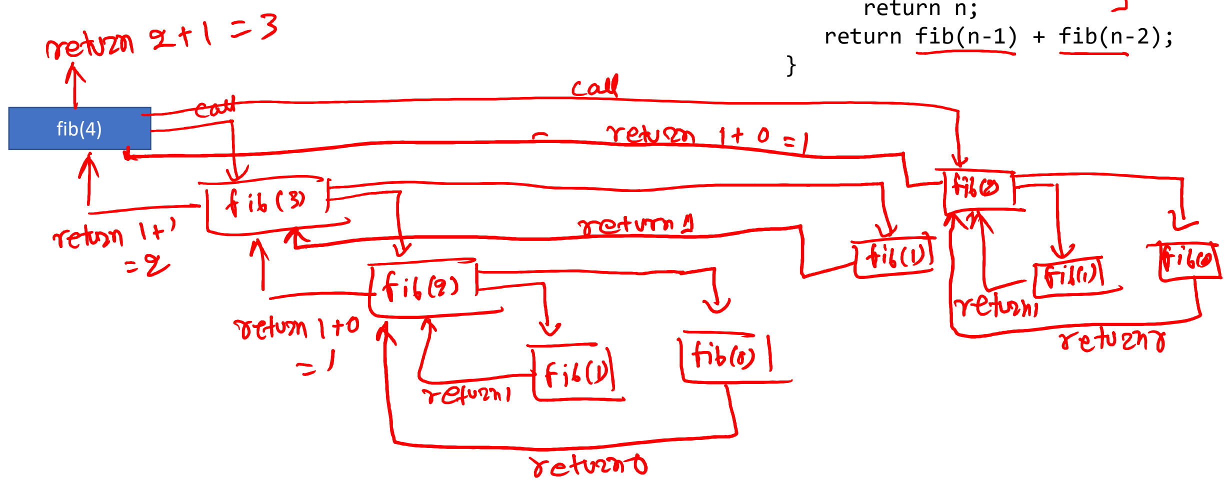
- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



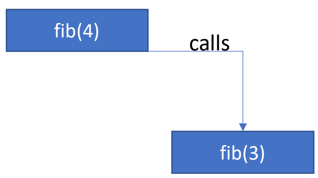
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

fib(4)

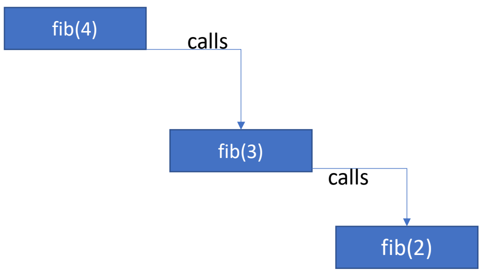
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



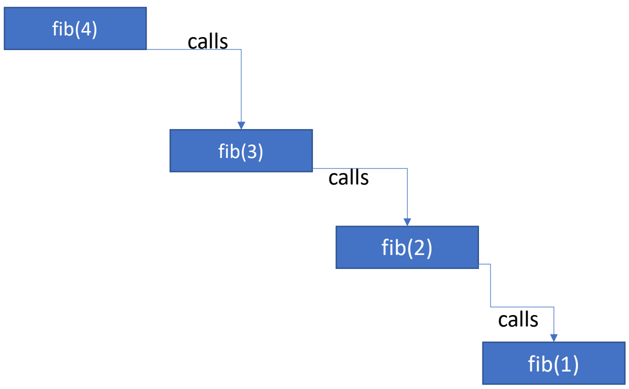
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



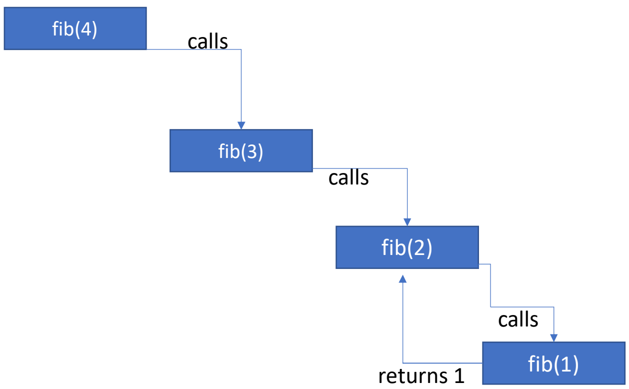
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



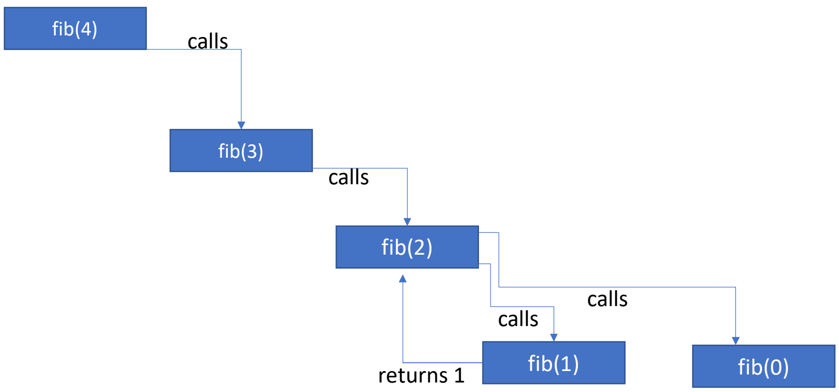
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



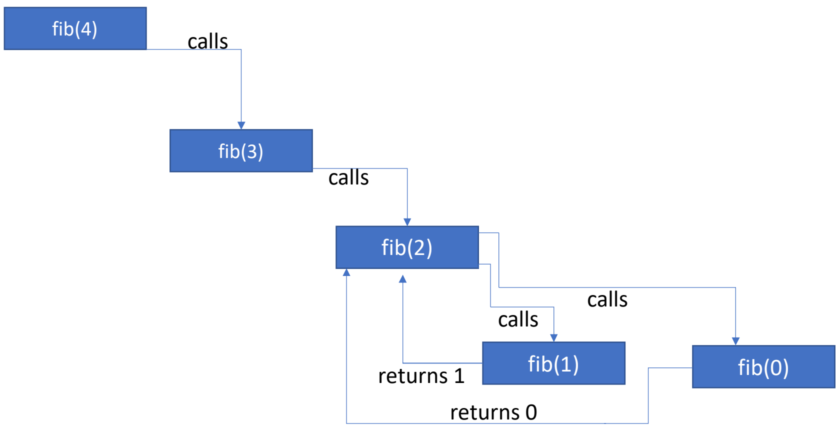
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



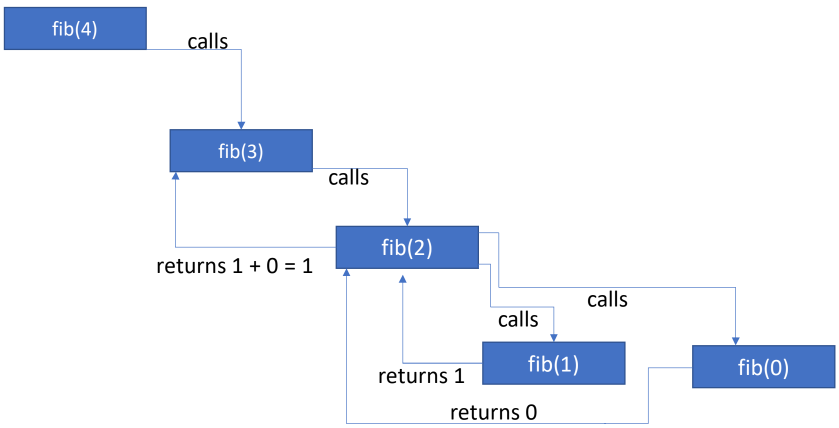
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



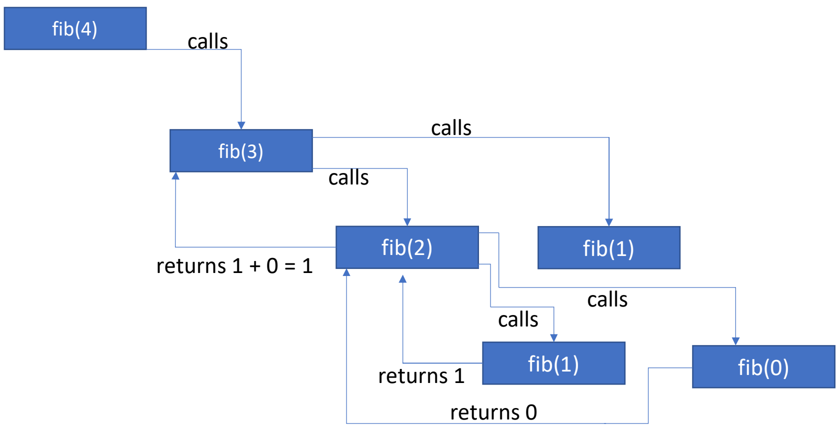
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



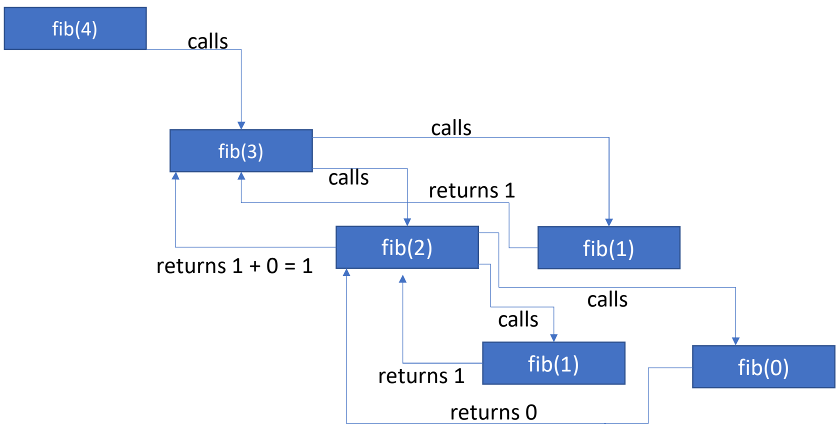
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



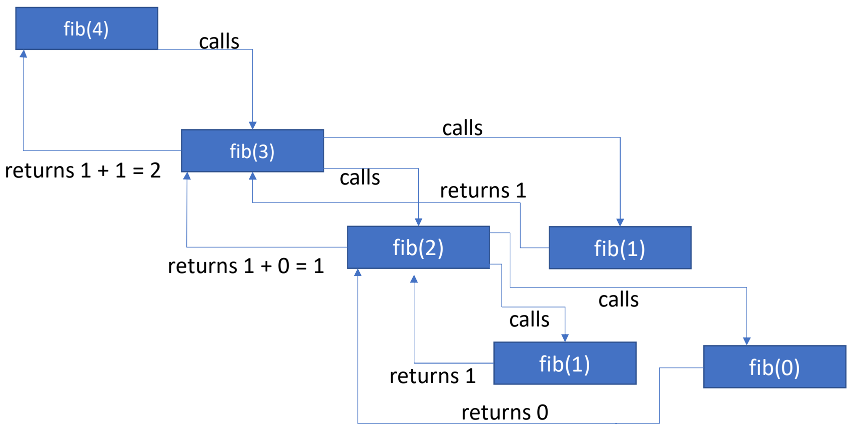
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



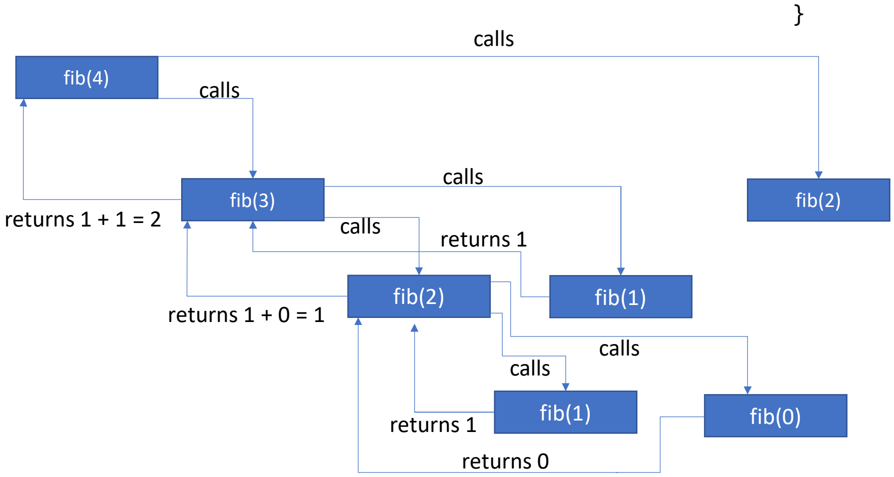
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



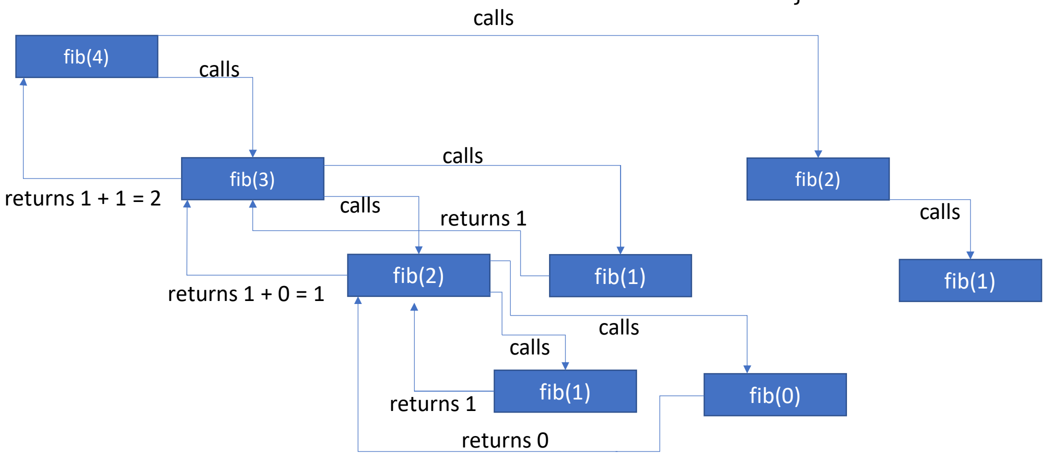
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



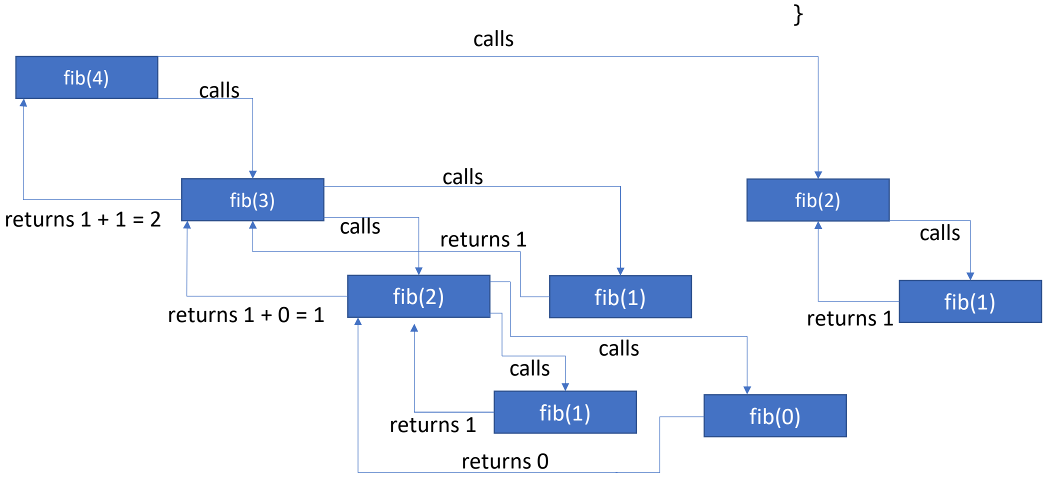
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



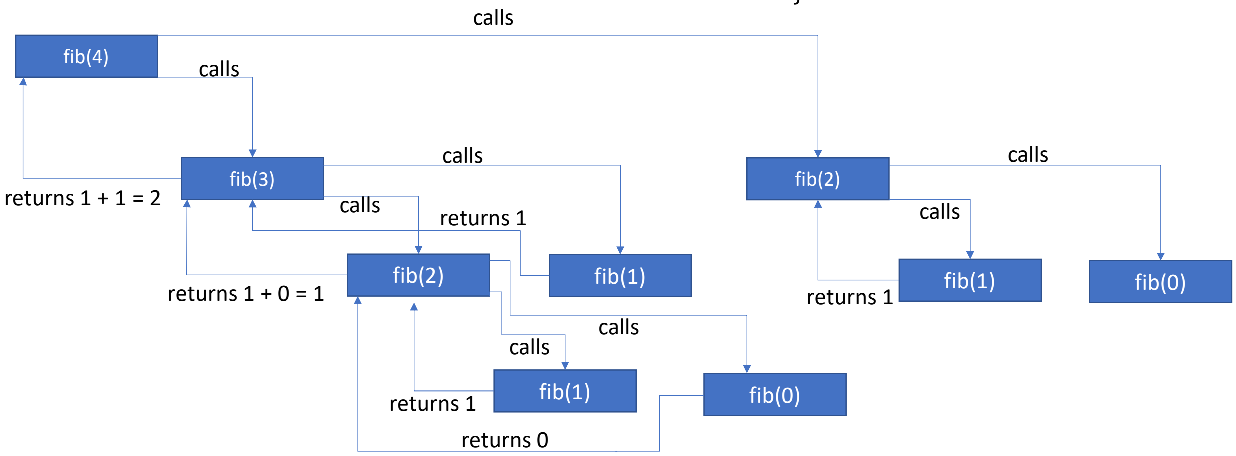
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



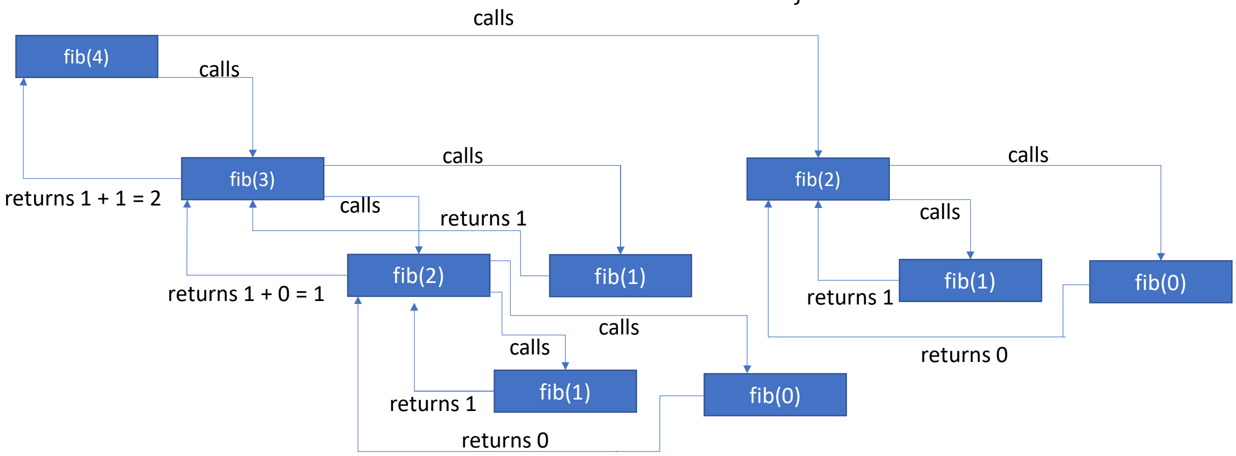
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



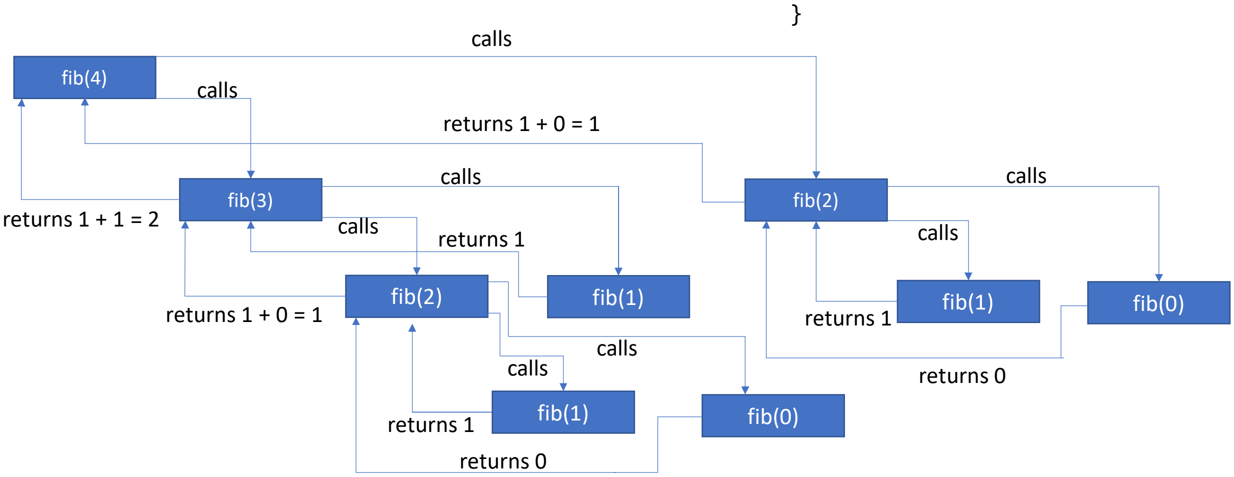
Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



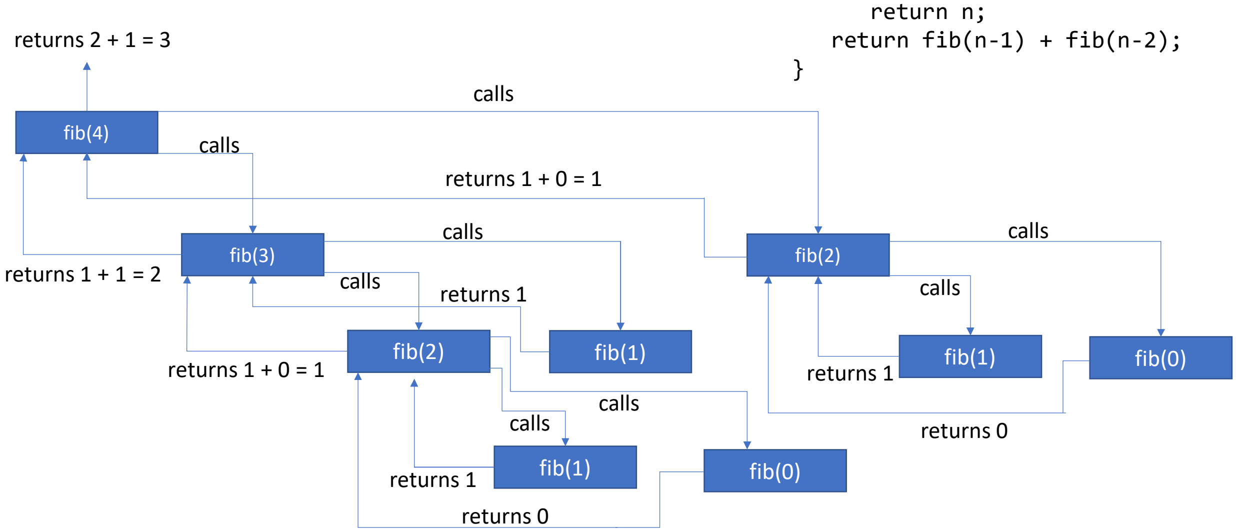
Fibonacci numbers

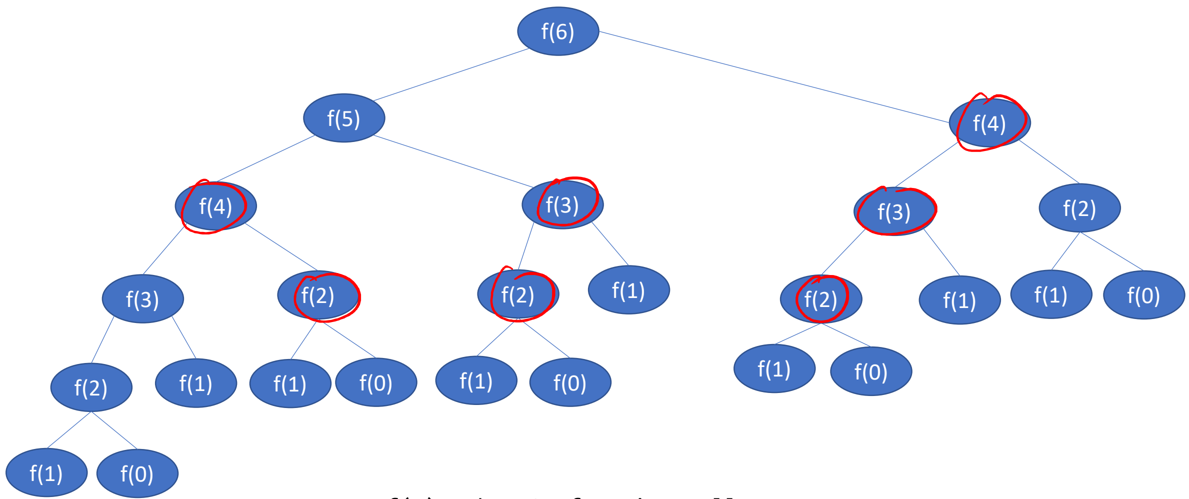
```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```



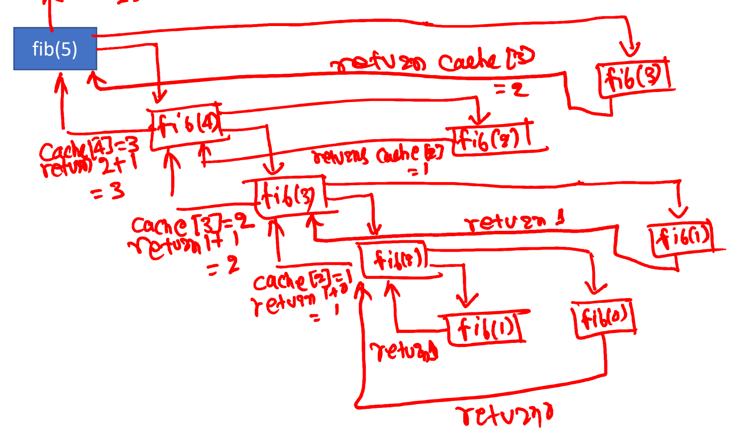


f(6) makes 24 function calls.

Faster solution

Fibonacci numbers

cache[5]=5
return 3+2 =5




```
int cache[1000];

int fib(int n) {
  int r;
   if (cache[n] != 0)
       return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
 cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

A cache can be used to reduce the number of repeating computations. We can save the result of a computation that might be used later in a cache (can be implemented using array / other data structures) and retrieve the result from the cache when needed, thus eliminating the need to recompute the result. However, such an approach may take more memory than the algorithm that doesn't use a cache.

Fibonacci numbers

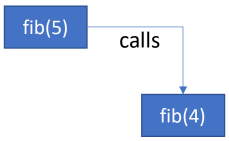
fib(5)

```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

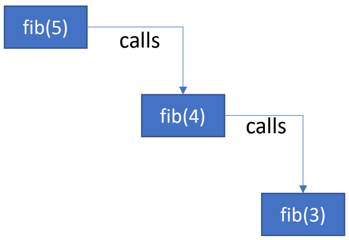



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

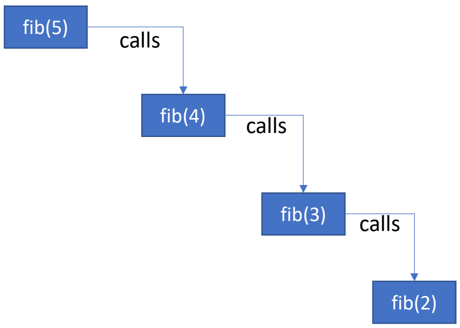



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf(“enter the value of n\n”);
  scanf(“%d”, &n);
  assert(n < 1000);
  r = fib(n);
  printf(“nth fib number is %d\n”, r);
  return 0;
}
```

Fibonacci numbers

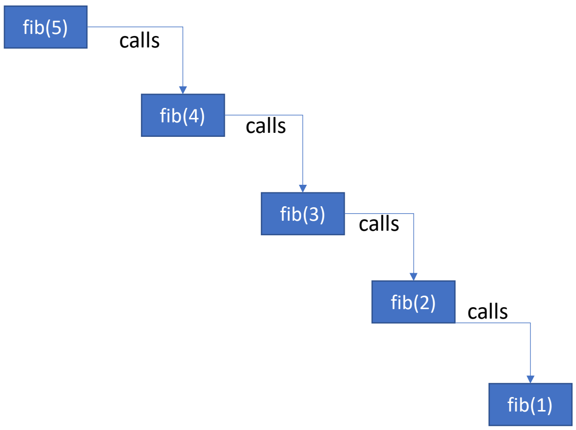



```c
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

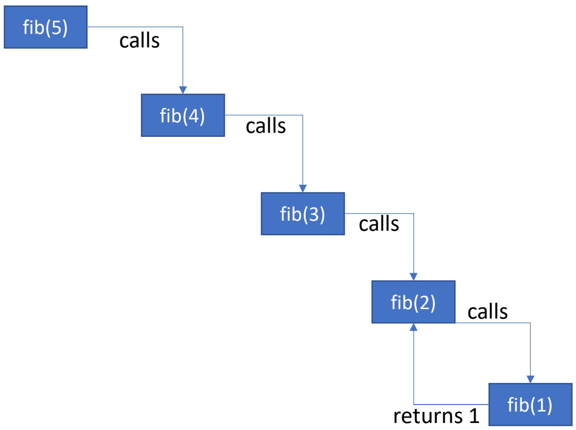



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf(“enter the value of n\n”);
  scanf(“%d”, &n);
  assert(n < 1000);
  r = fib(n);
  printf(“nth fib number is %d\n”, r);
  return 0;
}
```

Fibonacci numbers

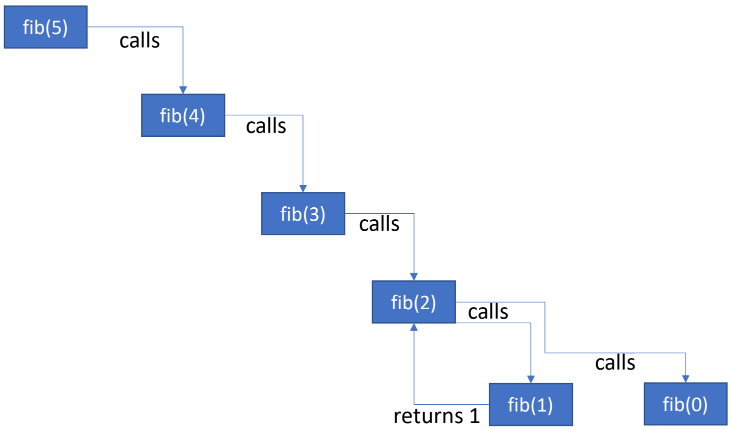



```c
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

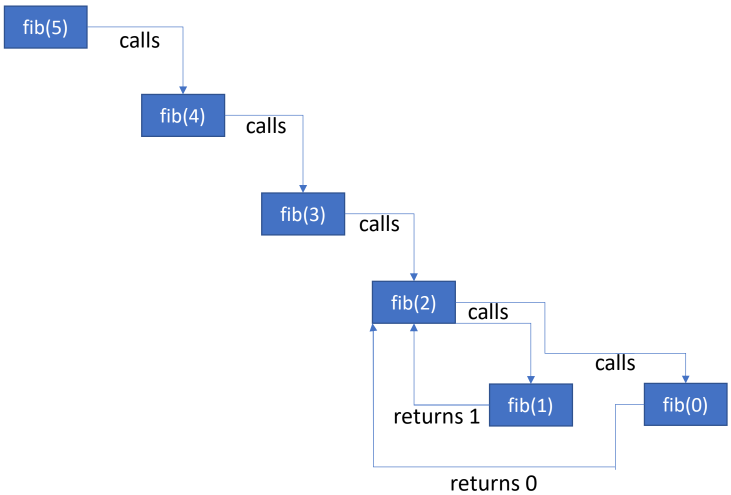



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

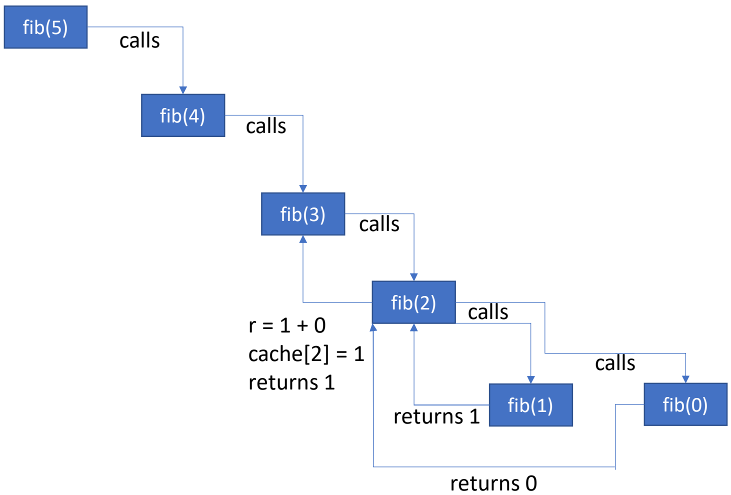



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

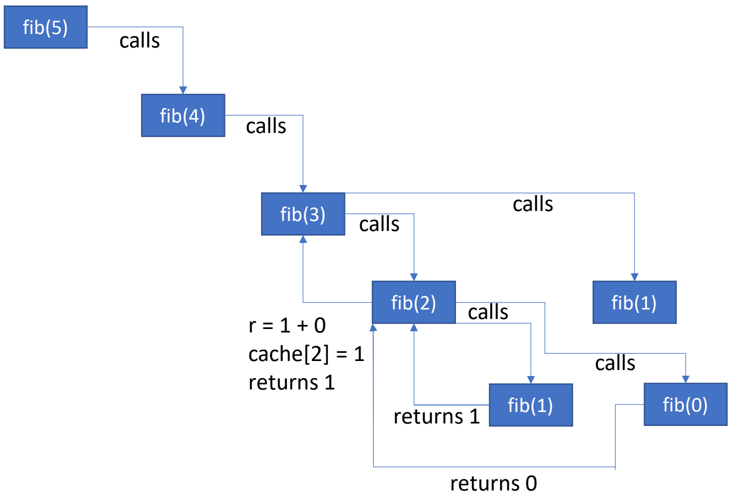



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

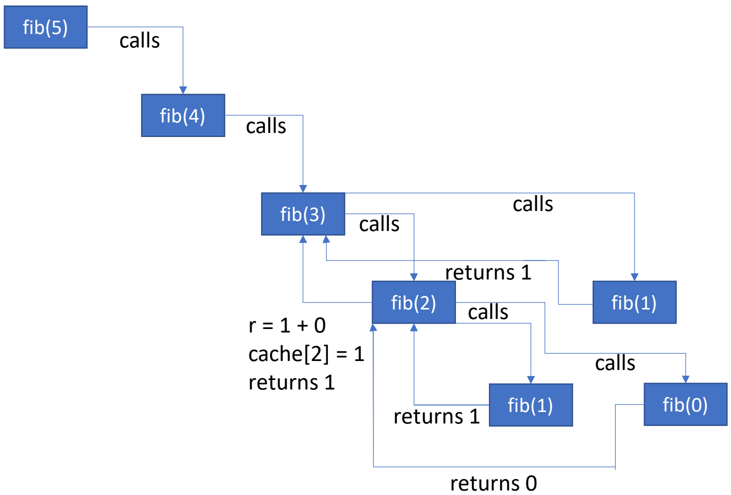



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

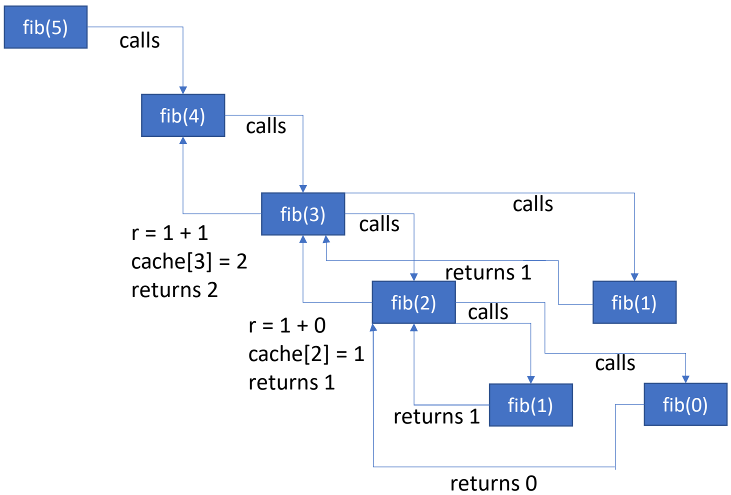



```c
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

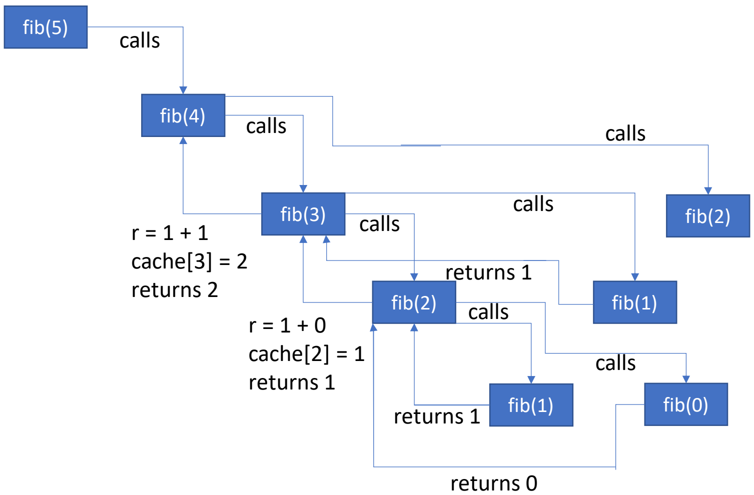



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf(“enter the value of n\n”);
  scanf(“%d”, &n);
  assert(n < 1000);
  r = fib(n);
  printf(“nth fib number is %d\n”, r);
  return 0;
}
```

Fibonacci numbers

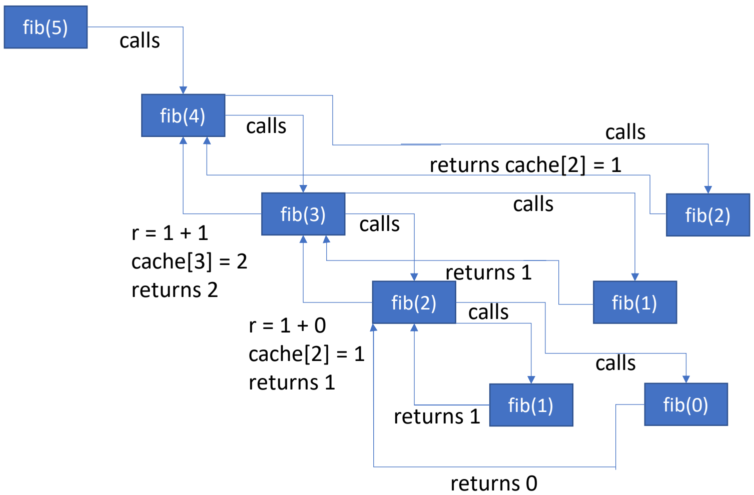



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

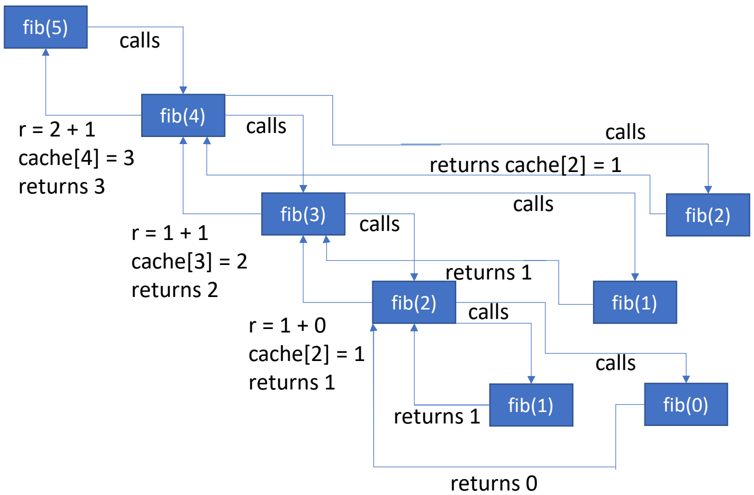



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

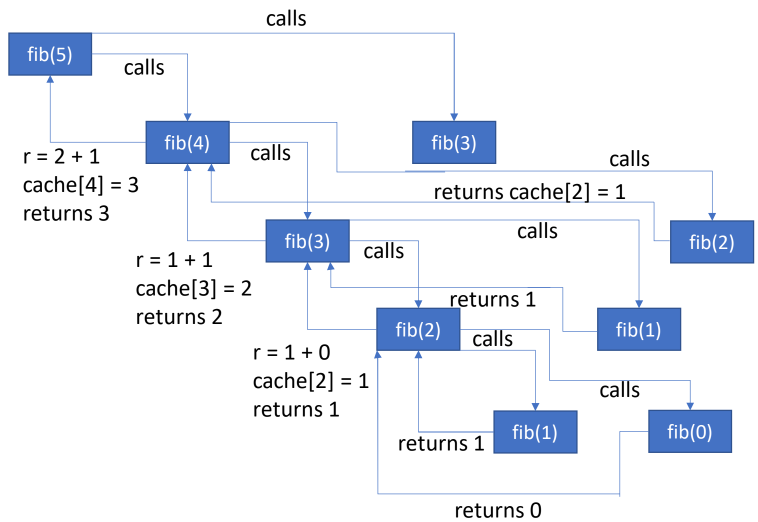



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

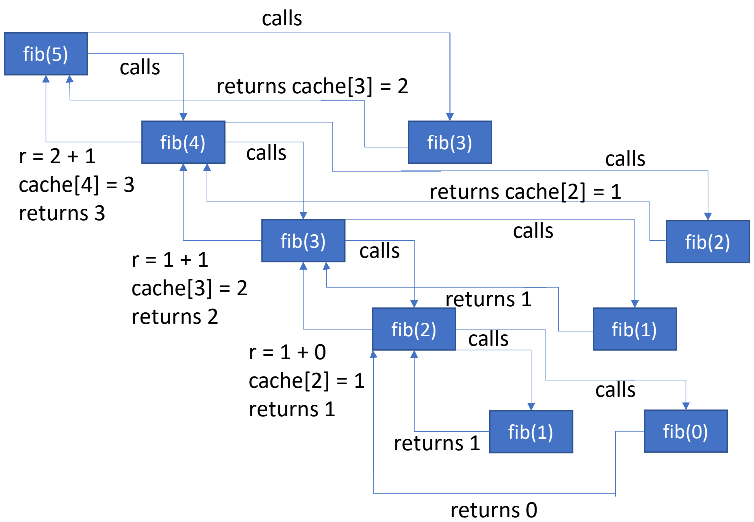



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers

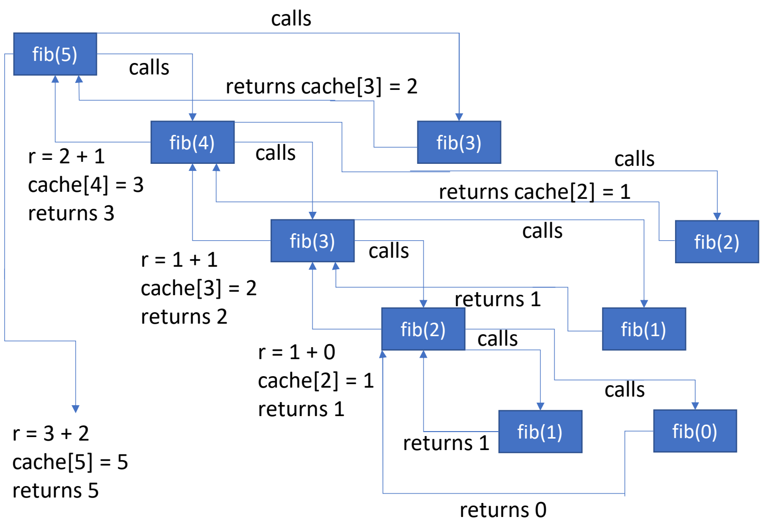



```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

Fibonacci numbers




```
int cache[1000];

int fib(int n) {
  int r;
  if (cache[n] != 0)
    return cache[n];
  if (n == 0 || n == 1)
    return n;
  r = fib(n-1) + fib(n-2);
  cache[n] = r;
  return r;
}

int main() {
  int n, r, i;
  for (i = 0; i < 1000; i++) {
    cache[i] = 0;
  }
  printf("enter the value of n\n");
  scanf("%d", &n);
  assert(n < 1000);
  r = fib(n);
  printf("nth fib number is %d\n", r);
  return 0;
}
```

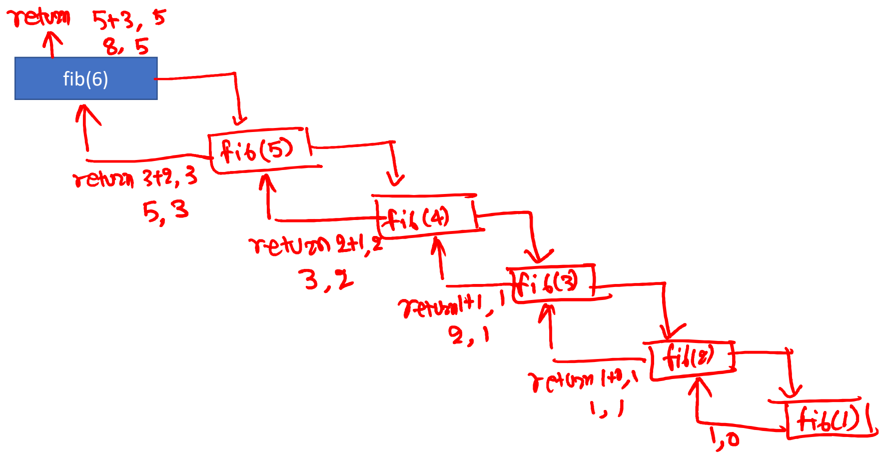
Faster recursive solution without caching

Faster recursive solution

- Let’s say fib(n) returns two values: fib(n) and fib(n-1), i.e.,
  - fib(5) returns fib(5) and fib(4)
  - fib(4) returns fib(4) and fib(3)
  - fib(3) returns fib(3) and fib(2)
  - fib(2) returns fib(2) and fib(1)
  - fib(1) returns 1 and 0

- How many function calls are needed to compute fib(6)?
  - The previous solution without caching requires 24 calls

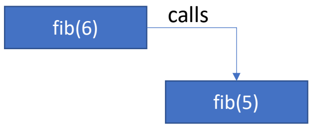
Faster recursive solution



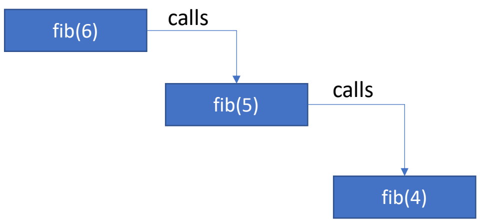
Faster recursive solution

fib(6)

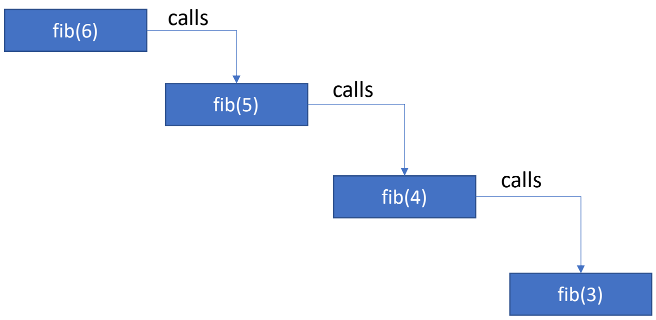
Faster recursive solution



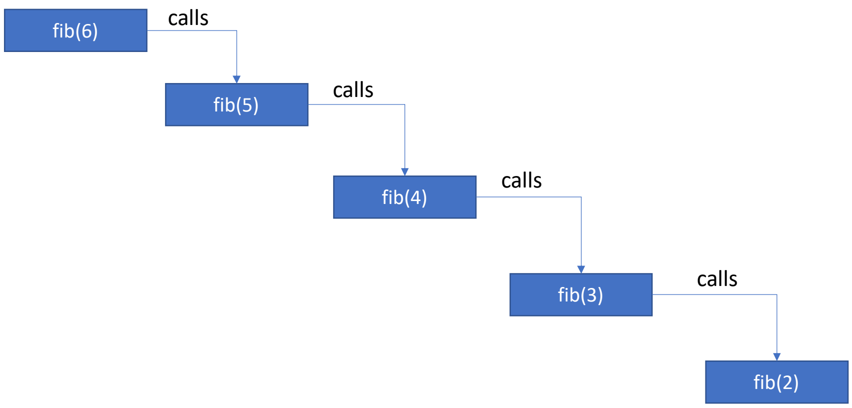
Faster recursive solution



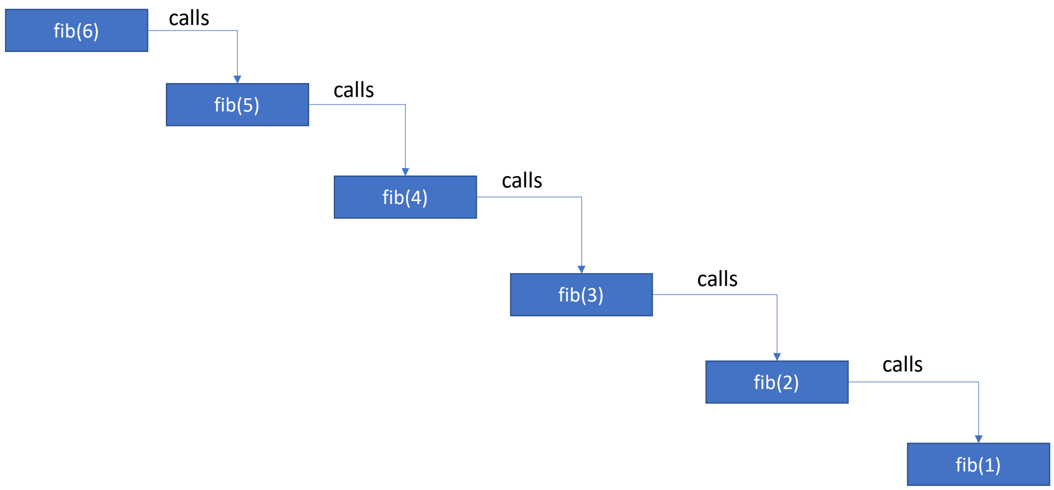
Faster recursive solution



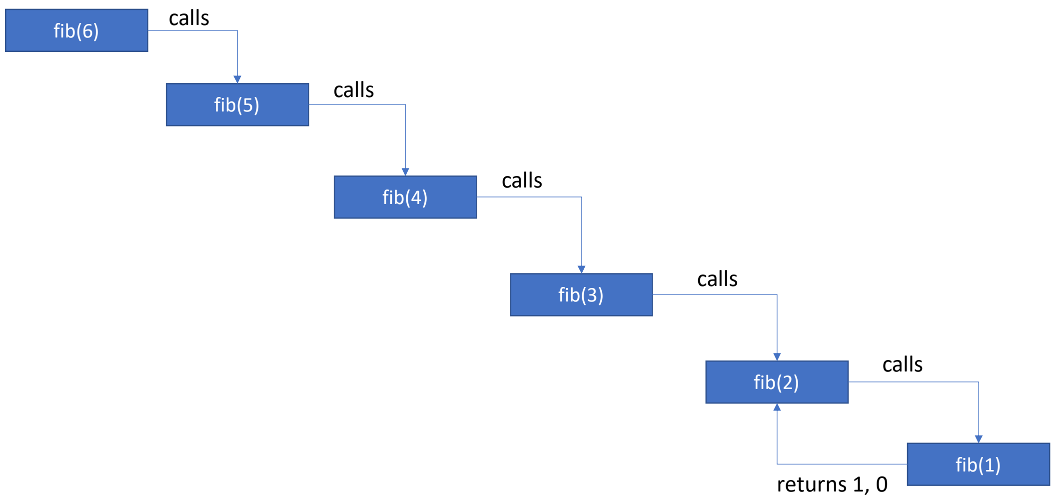
Faster recursive solution



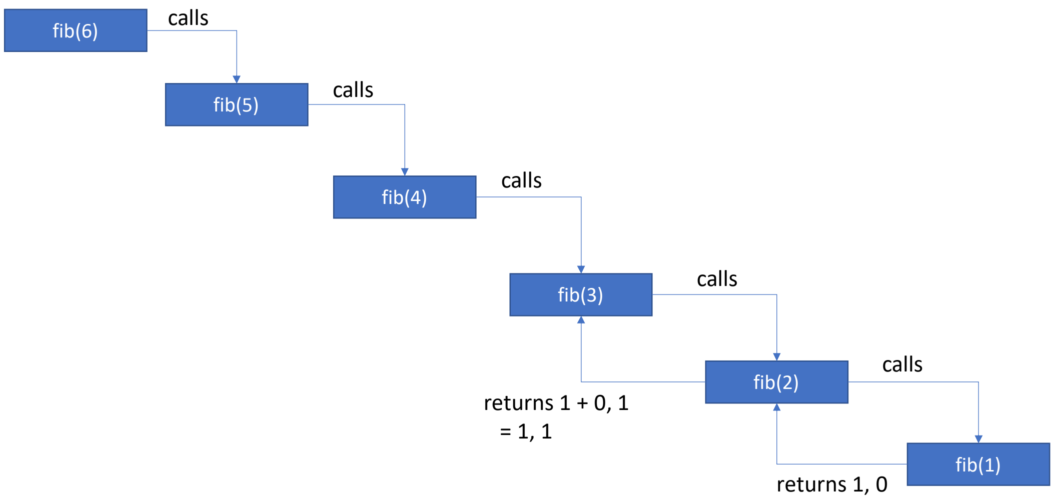
Faster recursive solution



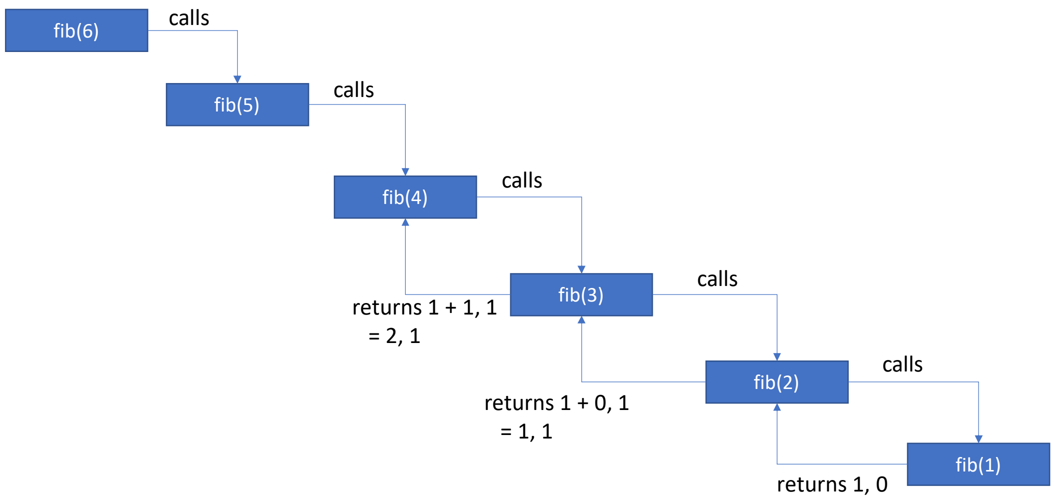
Faster recursive solution



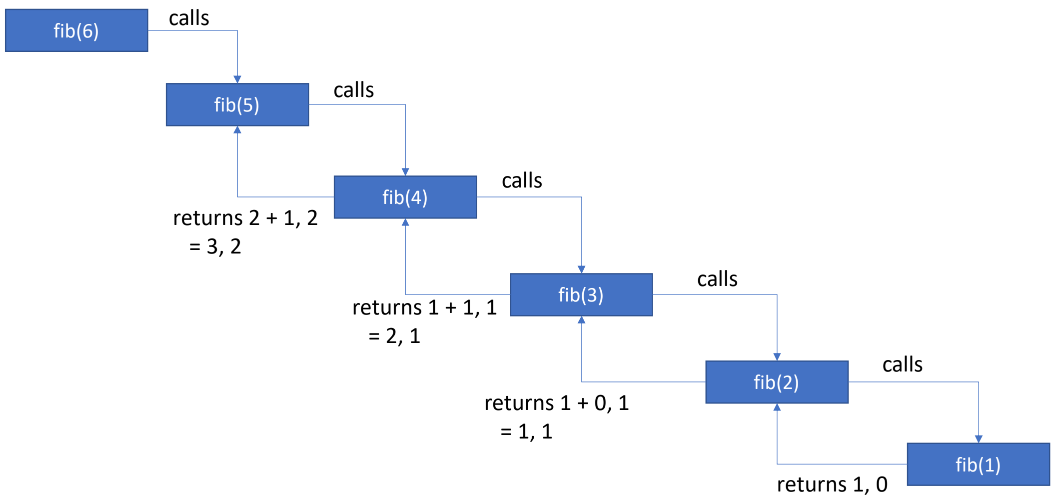
Faster recursive solution



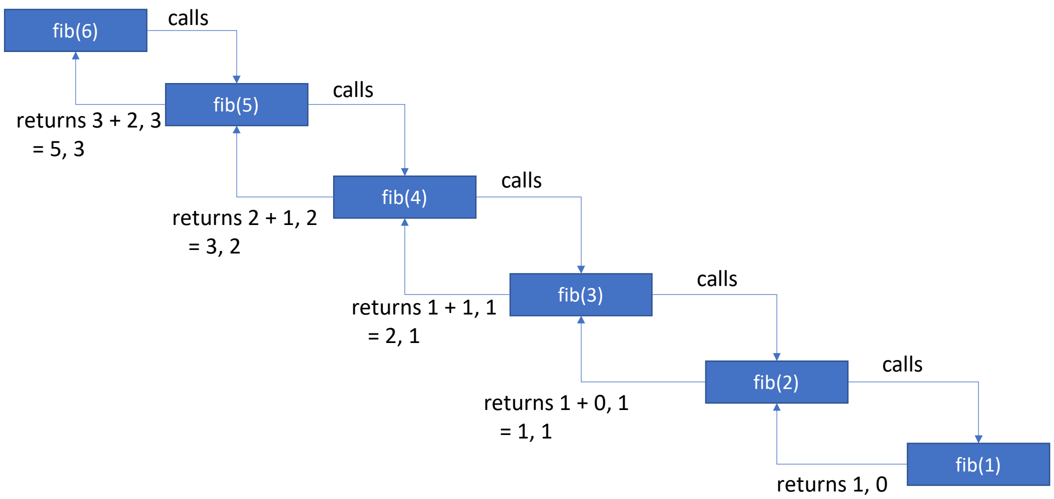
Faster recursive solution



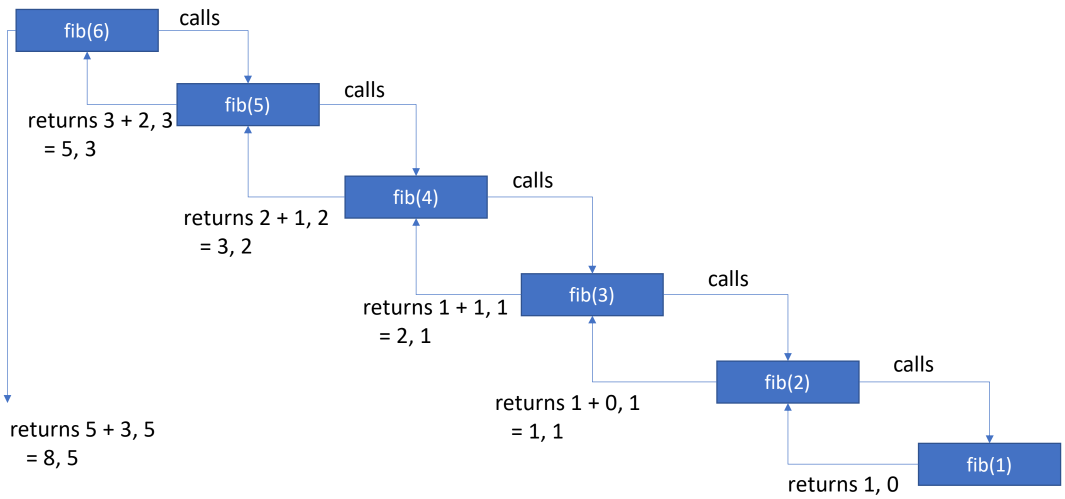
Faster recursive solution



Faster recursive solution



Faster recursive solution



Faster recursive solution

- Base case
  - `if (n == 1) return (1, 0)`
- Recursive step
  - Recursively call for `n-1` to obtain `(x, y) = (fib(n-1), fib(n-2))`
  - `return (x+y, x)`

In this recursive algorithm, instead of just returning fib(n), the fib function returns two values: fib(n) and fib(n-1), resulting in an efficient algorithm that doesn't require multiple recursive calls or caches.

Fibonacci numbers

```
struct retval {
  int x;
  int y;
};

struct retval fib(int n) {
  struct retval ret;
  struct retval r;
  if (n == 1) {
    r.x = 1;
    r.y = 0;
    return r;
  }
  ret = fib(n - 1);
  r.x = ret.x + ret.y;
  r.y = ret.x;
  return r;
}
```

struct retval r;   int r;

r.x = 10;

r.y = 20;

int z;

z = r.x;

In C, “struct” is a way to define a new data type. A struct may contain multiple fields of possibly different types, including the struct type.

Iterative solution

Iterative solution for Fibonacci numbers

Iterative solution for Fibonacci numbers

pprev = fib(0) prev = fib(1)

n = 4

i = 2 ; i <= 4 ;

res = 1 + 0 = 1 fib(2)

pprev = fib(1) = 1

prev = fib(2) = 1

i = 3

i = 3 ; i <= 4

res = 1 + 1 = 2

pprev = 1

prev = 2

i = 4

i = 4 ; i ≤ 4

res = 1 + 2 = 3

pprev = 2

prev = 3

i = 5

```
int fib(int n) {
  if (n == 0 || n == 1) {
    return n;
  }

  int prev = 1;
  int pprev = 0;
  int res, i;

  for (i = 2; i <= n; i++) {
    res = prev + pprev;
    pprev = prev;
    prev = res;
  }
  return res;
}
```

Iterative solution for Fibonacci numbers

compute fib(5):

prev = 1
pprev = 0

iteration 1 : i = 2; i <= 5
res = 1 + 0 = 1
pprev = 1
prev = 1
i = 3

iteartion2: i = 3; i <= 5
res = 1 + 1 = 2
pprev = 1
prev = 2
i = 4

iteration 3 : i = 4; i <= 5
res = 2 + 1 = 3
pprev = 2
prev = 3
i = 5

iteartion4: i = 5; i <= 5
res = 3 + 2 = 5
pprev = 3
prev = 5
i = 6

return res = 5

```
int fib(int n) {
  if (n == 0 || n == 1) {
    return n;
  }

  int prev = 1;
  int pprev = 0;
  int res, i;

  for (i = 2; i <= n; i++) {
    res = prev + pprev;
    pprev = prev;
    prev = res;
  }
  return res;
}
```

The iterative algorithm computes fib(2), fib(3), fib(4), …, fib(n) in an iterative manner. It keeps track of the last two Fibonacci numbers needed to compute the next Fibonacci number.

Can we do better?

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 \times f(n-1) + 1 \times f(n-2) \\ 1 \times f(n-1) + 0 \times f(n-2) \end{bmatrix}$$

$$= \begin{bmatrix} f(n-1) + f(n-2) \\ f(n-1) \end{bmatrix}$$

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix}$$

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \begin{bmatrix} f(n-3) \\ f(n-4) \end{bmatrix} = \dots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Fibonacci numbers

- Let $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = A^{n-1} * \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

How to compute A^{n-1} fast?

Fibonacci numbers

```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  int A[2][2];
  int R[2][2];

  A[0][0] = 1; A[0][1] = 1;
  A[1][0] = 1; A[1][1] = 0;

  mul(A, R, n-1);
  // R contains A^(n-1)
  return R[0][0];
}
```

The fib routine initializes A (a 2x2 matrix) with the value discussed in the previous slide. The mul routine takes two 2x2 matrices, A and R; the value of n; and returns A^(n-1) in R. The mul2 routine takes two 2x2 matrices, P and Q; computes P x Q; and stores them in P before returning. The recursive algorithm to compute A^n is similar to the faster algorithm we discussed for computing x^n.

Fibonacci numbers

mul2 implementation?

mul2 (int A[2][2], int B[2][2])

int t[2][2];

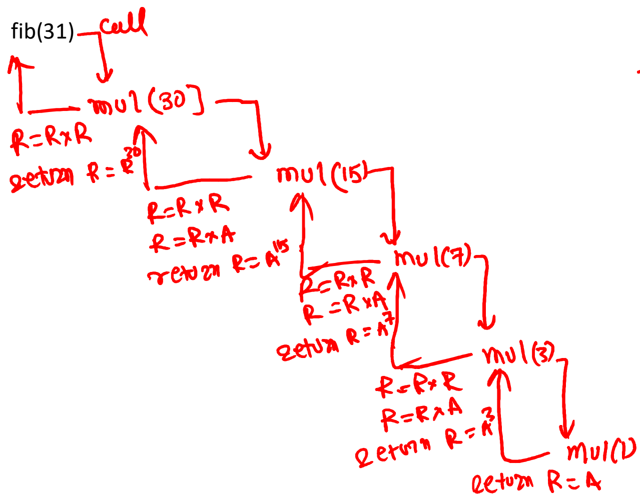
t[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0]

$$\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \begin{bmatrix} p_1 & p_2 \\ q_1 & q_2 \end{bmatrix}$$

$[x_1 p_1 + x_2 q_1$

```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers




```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers

fib(31)

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers

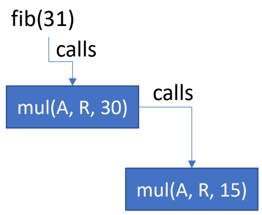
fib(31)

calls

mul(A, R, 30)

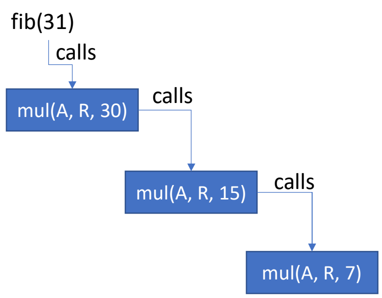
```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



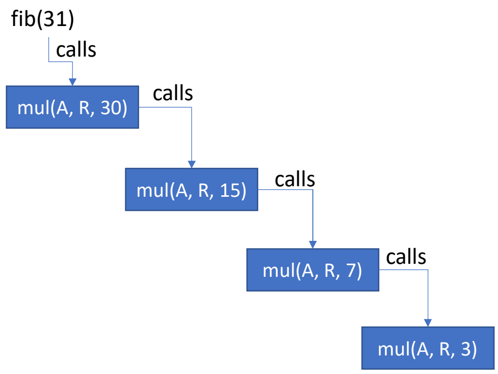

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



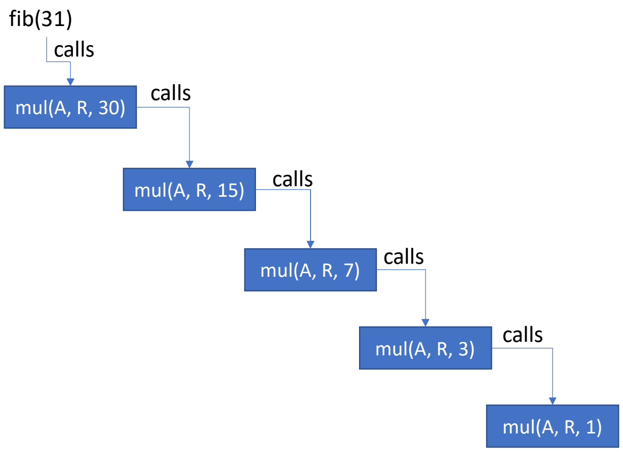

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



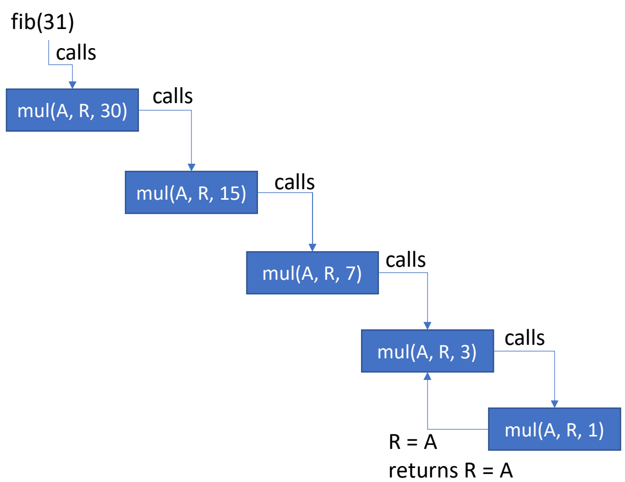

```c
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



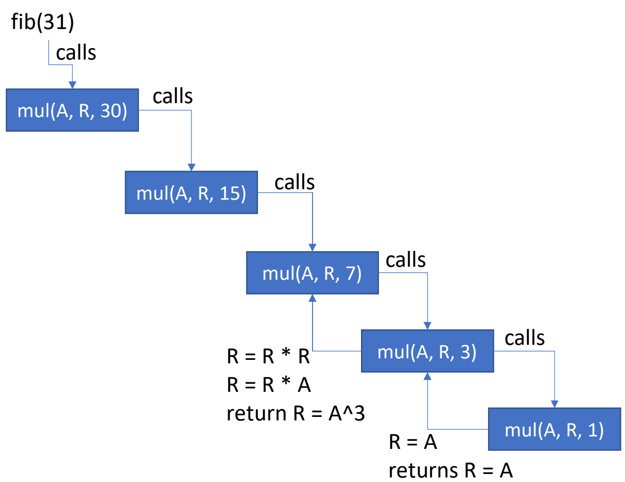

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



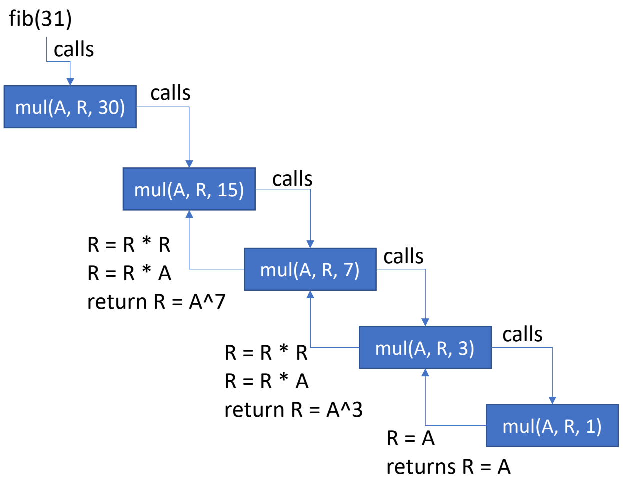

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



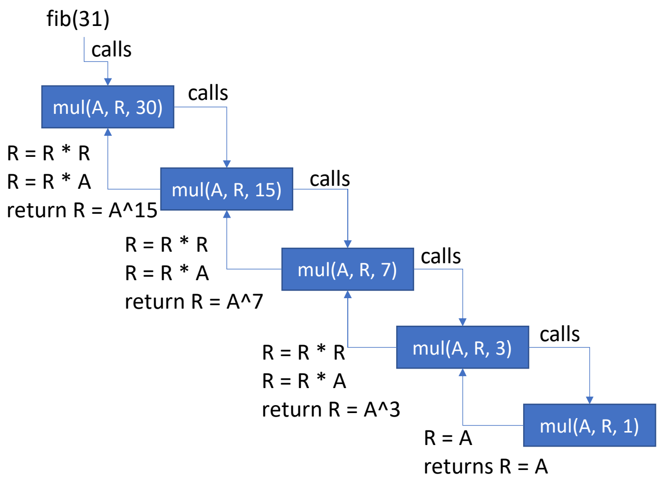

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers



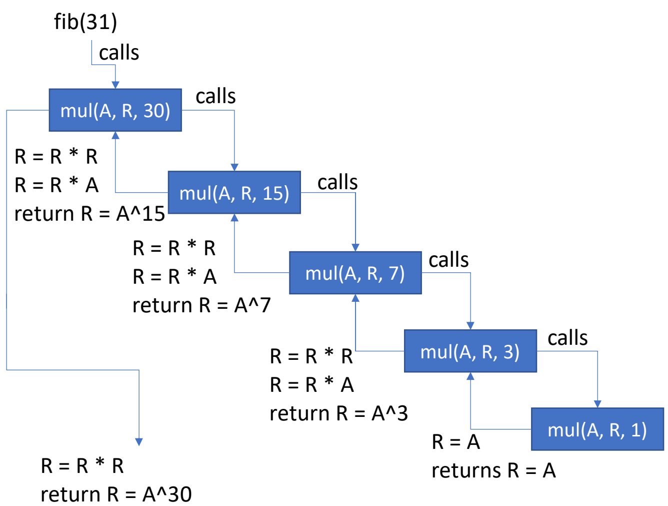

```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers




```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Fibonacci numbers




```
// mul takes a matrix A and R as input
// returns the result of A^n in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

Notice that this algorithm only makes five recursive calls to compute f(31) in contrast to the previous recursive algorithm that makes around 30 calls.

Homework

- Modify the fib(n) routines (matrix mul and iterative) discussed in the class to return “nth Fibonacci number % 1000” instead of “nth Fibonacci number”
- Compare the runtimes of matrix mul vs. iterative algorithms for various inputs

Search algorithms

Linear search

Linear search

- Let `arr` be an input array of length `n`. Given a value `x`, we want to find an index i (`0 <= i < n`), such that `arr[i] == x`. If no such index exists, then the algorithm returns `-1`.

Linear search

12	11	23	13	41	19	25
0	1	2	3	4	5	6

Return 3

```
int lsearch(int arr[], int len, int val);

What is the output of lsearch(arr, 7, 13)?
```
3

```
What is the output of lsearch(arr, 7, 30)?
```
-1

Iterative linear search

```
int lsearch(int arr[], int val, int len) {
   int i;
   for (i = len-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

This linear search algorithm iterates all the elements in the array, starting from the last index to the start index. If the value of an array element is equals to val, it returns the corresponding index; otherwise, if val is not present in the array, lsearch returns -1.

Recursive linear search

- `int lsearch(int arr[], int val, int len);`
  - Initially, `len` is the length of the input `arr`
  - `val` is the value being searched

Recursive linear search

- `int lsearch(int arr[], int val, int len);`
  - Initially, `len` is the length of the input `arr`
  - `val` is the value being searched
- Base cases
  - `if (len == 0) return -1`
  - `if (arr[len-1] == val) return len-1`
- Recursive step
  - Decrement the length of the array and recursively call search

Recursive linear search

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Recursive linear search

12	11	23	13	41	19	25
0	1	2	3	4	5	6

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

lsearch(arr, 13, 7) call

lsearch(arr, 13, 6)

return 3

call

lsearch(arr, 13, 5)

return 3

call

lsearch(arr, 13, 4)

return 3

return 3

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
        calls
      lsearch(arr, 13, 6)
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
        calls
    lsearch(arr, 13, 6)
            calls
        lsearch(arr, 13, 5)
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
    calls
        lsearch(arr, 13, 6)
            calls
                lsearch(arr, 13, 5)
                    calls
                        lsearch(arr, 13, 4)
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
         calls
    lsearch(arr, 13, 6)
              calls
         lsearch(arr, 13, 5)
                    calls
                   lsearch(arr, 13, 4)
          returns 3
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

lsearch(arr, 13, 4) returns 3 because arr[len-1], i.e., arr[3] is equal to the value we are searching for, i.e., 13.

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
    calls
        lsearch(arr, 13, 6)
            calls
returns 3       lsearch(arr, 13, 5)
                    calls
                        lsearch(arr, 13, 4)
            returns 3
```

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

lsearch(arr, 13, 5) returns the return value of lsearch(arr, 13, 4), which is 3.

Recursive linear search

12	11	23	13	41	19	25

```
lsearch(arr, 13, 7)
        calls
returns 3    lsearch(arr, 13, 6)
                    calls
        returns 3    lsearch(arr, 13, 5)
                            calls
                returns 3    lsearch(arr, 13, 4)
```

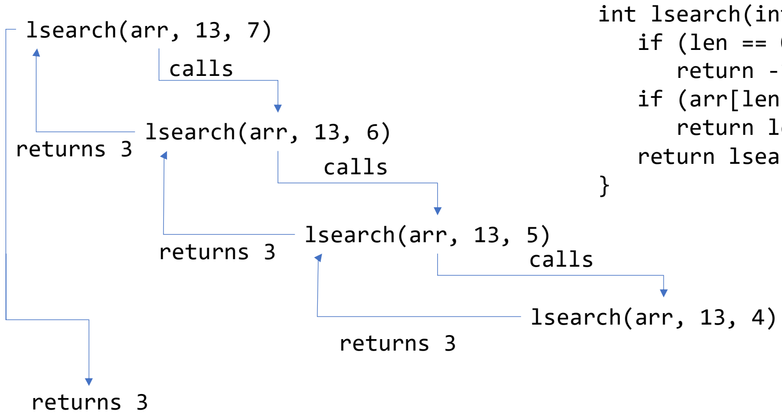
```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

lsearch(arr, 13, 6) returns the return value of lsearch(arr, 13, 5), which is 3.

Recursive linear search

12	11	23	13	41	19	25

```
int lsearch(int arr[], int val, int len) {
    if (len == 0)
        return -1;
    if (arr[len-1] == val)
        return len-1;
    return lsearch(arr, val, len-1);
}
```



lsearch(arr, 13, 7) returns the return value of lsearch(arr, 13, 6), which is 3.

Binary search

Binary search

- Let `arr` be an input sorted array (in ascending order) of length n. Given a value x, we want to find an index i (`0 <= i < n`), such that `arr[i] == x`. If no such index exists, then the algorithm returns `-1`.

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

The binary search algorithm works on a sorted array. The algorithm first computes the middle index (mid), which is equal to (start+end)/2, where start and end are the first and last index of the array. In this case, initially, mid is 5. If the val == arr[mid], then the algorithm simply returns mid; otherwise, if val > arr[mid], then the searching is performed in the subarray starting from mid+1 to end, else we search the element in the subarray starting from start to mid-1. If the element is not present in the array, the start index will eventually become lesser than the end index, and in that case, the algorithm returns -1.

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

Searching 15 in the range (0,10). mid = (0+10)/2 = 5. 15 < arr[5], search in the range (0, 4).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

Searching 15 in the range (0,4). mid = (0+4)/2 = 2. 15 < arr[2], search in the range (0, 1).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

Searching 15 in the range (0,1). mid = (0+1)/2 = 0. 15 > arr[0], search in the range (1, 1).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 15
```

Searching 15 in the range (1,1). mid = (1+1)/2 = 1. 15 == arr[1], returns 1.

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Searching 74 in the range (0,10). mid = (0+10)/2 = 5. 74 > arr[5], search in the range (6, 10).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Searching 74 in the range (6,10). mid = (6+10)/2 = 8. 74 < arr[8], search in the range (6, 7).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Searching 74 in the range (6,7). mid = (6+7)/2 = 6. 74 > arr[6], search in the range (7, 7).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Searching 74 in the range (7,7). mid = (7+7)/2 = 7. 74 < arr[7], search in the range (7, 6).

Binary search

11	15	19	23	41	59	65	75	79	81	87
0	1	2	3	4	5	6	7	8	9	10

```
int bsearch(int arr[], int len, int val);
search 74
```

Searching 74 in the range (7,6). start > end. The value is not present. Returns -1.

Recursive solution

- `int bsearch(int arr[], int val, int lo, int hi);`
  - Initially, `lo` = 0 and hi = length(`arr`) `- 1`
  - `val` is the value being searched
- Base cases
  - `if (lo > hi) return -1`
  - `mid = (lo + hi)/2`
  - `if (arr[mid] == val) return mid`
- Recursive step
  - If `val` > `arr[mid],` search the second half (`mid+1, lo`), recursively
  - Otherwise, search the first half (`lo, mid-1`)

Recursive solution

```
int bsearch(int arr[], int val, int lo, int hi) {
   if (hi < lo)
      return -1;
   int mid = (lo + hi) / 2;
   if (arr[mid] == val)
      return mid;
   if (arr[mid] > val)
      return bsearch(arr, val, lo, mid-1);
   else
      return bsearch(arr, val, mid+1, hi);
}
```