8장. 재귀 호출

강의 목표

- 재귀 함수에 대한 이해 및 재귀 함수 사용의 이점 설명 (8.1).
- 재귀 함수에서 기본 작업 정의하는 방법 (8.2-8.6).
- 재귀 함수 호출과 호출 스택의 처리 내용 이해 (8.2-8.6).
- 재귀를 사용하여 문제 해결 (8.2-8.6).
- 재귀 함수를 파생하기 위해 오버로딩 보조 함수 사용 (8.5).
- 재귀와 반복의 관계 및 차이점 이해 (8.7).

팩토리얼 계산

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

ComputeFactorial

Run

animation

팩토리얼 계산

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(3)

animation

팩토리얼 계산

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(3) = 3 * factorial(2)

animation

팩토리얼 계산

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

animation

팩토리얼 계산

```
factorial(0) = 1;
factorial(n) = n*factorial(n-1);
```

factorial(3) = 3 * factorial(2)
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))

animation

팩토리얼 계산

```
factorial(0) = 1;
factorial(n) = n*factorial(n-1);
```

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

animation

팩토리얼 계산

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

= 3 * (2 * 1)

animation

팩토리얼 계산

```
factorial(0) = 1;
factorial(n) = n*factorial(n-1);
```

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

= 3 * (2 * (1 * factorial(0)))

= 3 * (2 * (1 * 1)))

= 3 * (2 * 1)

= 3 * 2

animation

팩토리얼 계산

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(3) = 3 * factorial(2)
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))
= 3 * (2 * (1 * 1)))
= 3 * (2 * 1)
= 3 * 2
= 6

animation

재귀호출 factorial 함수 Trace

factorial(4) 수행

factorial(4)

Stack

Main method

animation

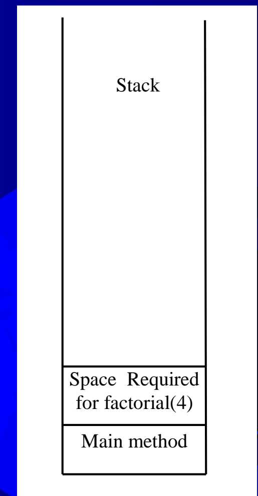
재귀호출 factorial 함수 Trace

factorial(4)

Step 0: executes factorial(4)

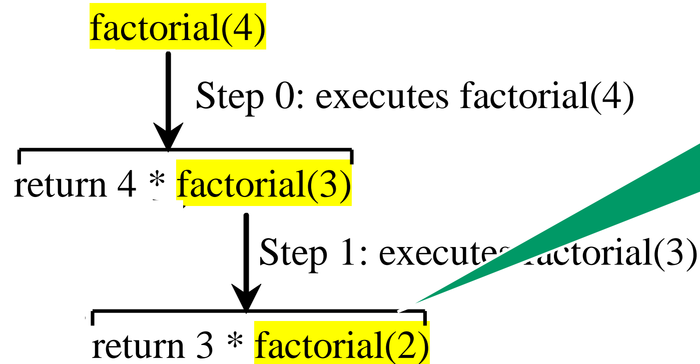
return 4 * factorial(3)

factorial(3) 수행

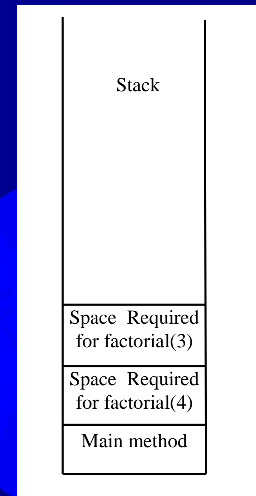


animation

재귀호출 factorial 함수 Trace

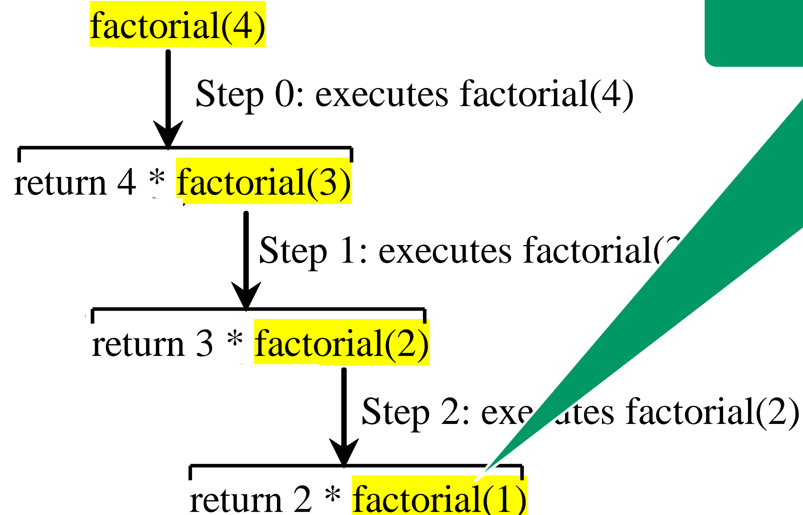


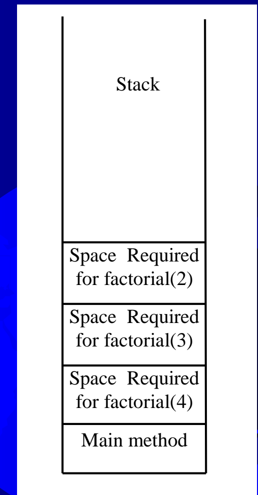
factorial(2) 수행



animation

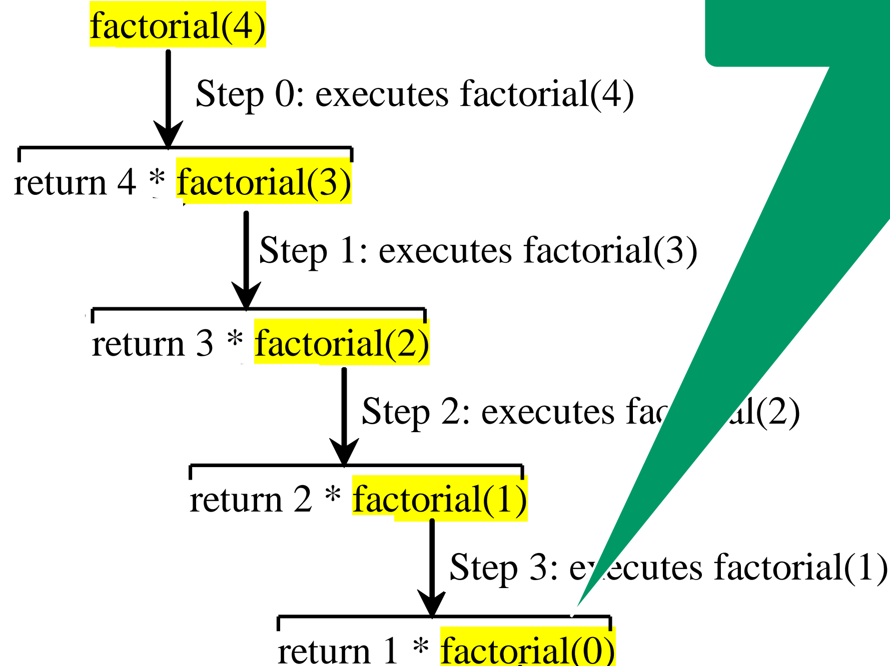
재귀호출 factorial 함수 Trace





animation

재귀호출 factorial 함수 Trace

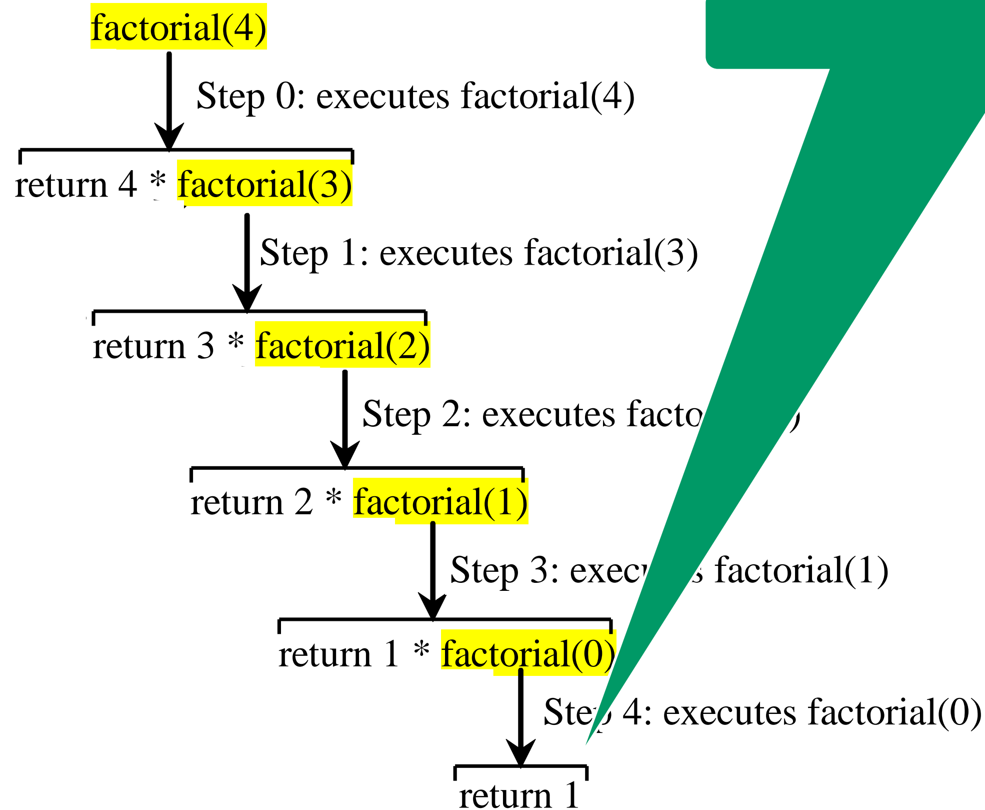


Stack

Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

animation

재귀호출 factorial 함수 Trace

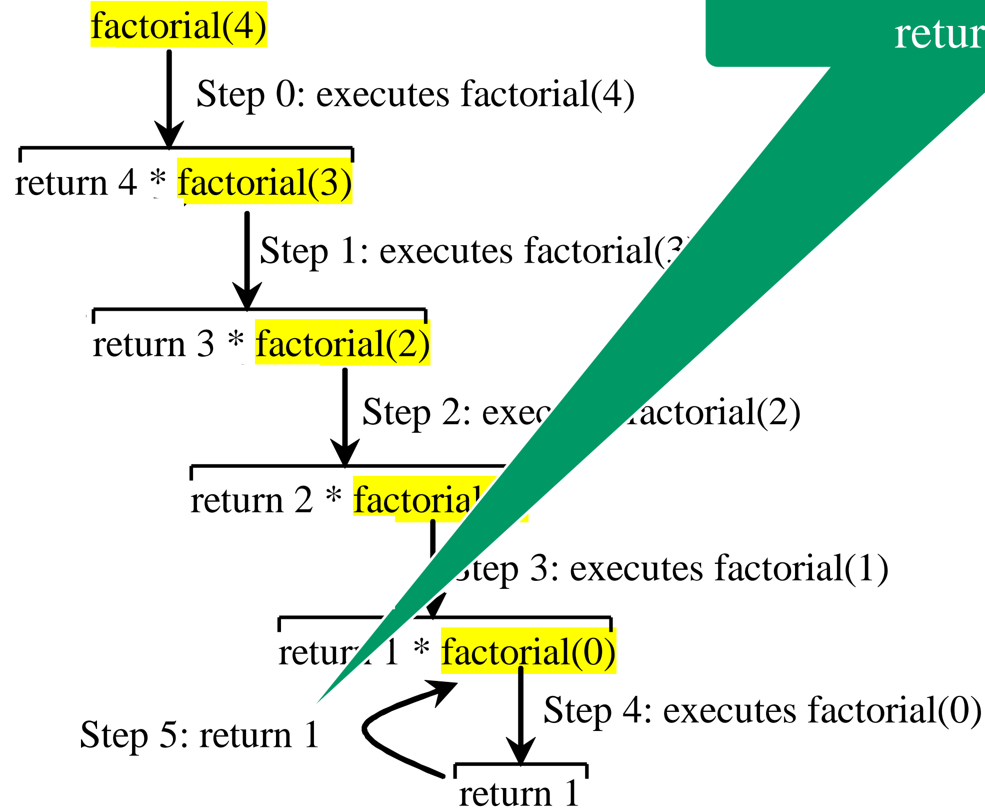


Stack

Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

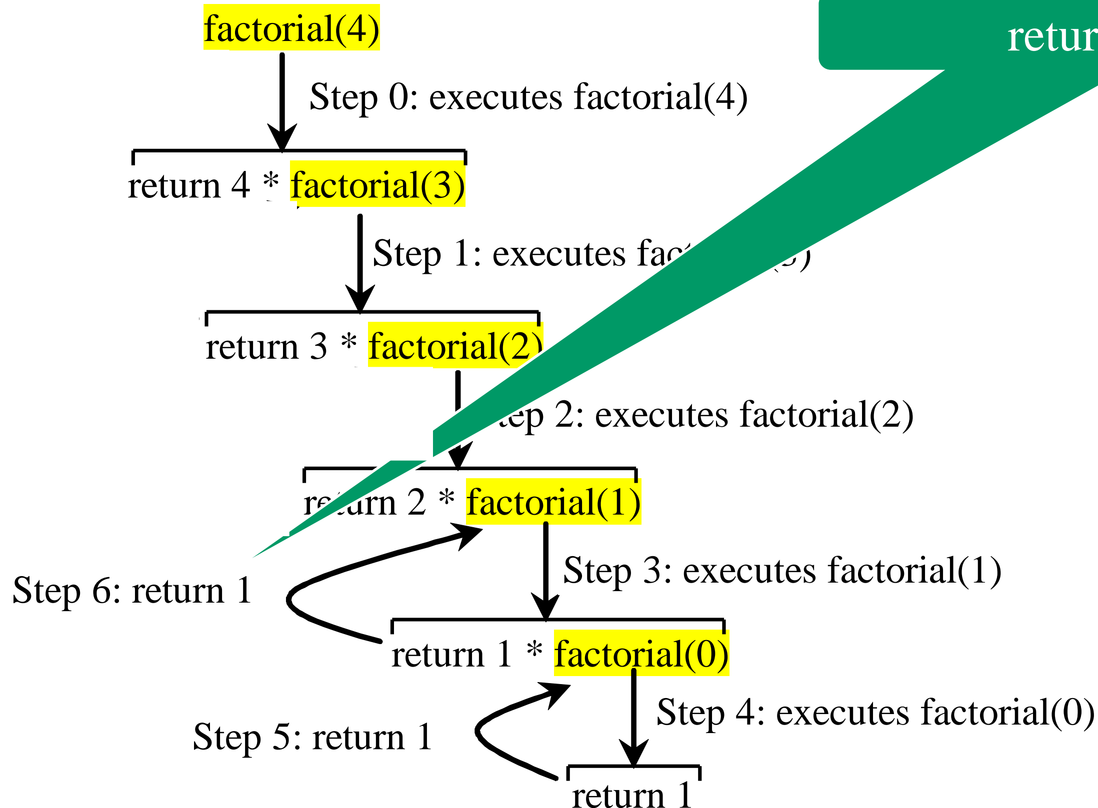
animation

재귀호출 factorial 함수 Trace



animation

재귀호출 factorial 함수 Trace



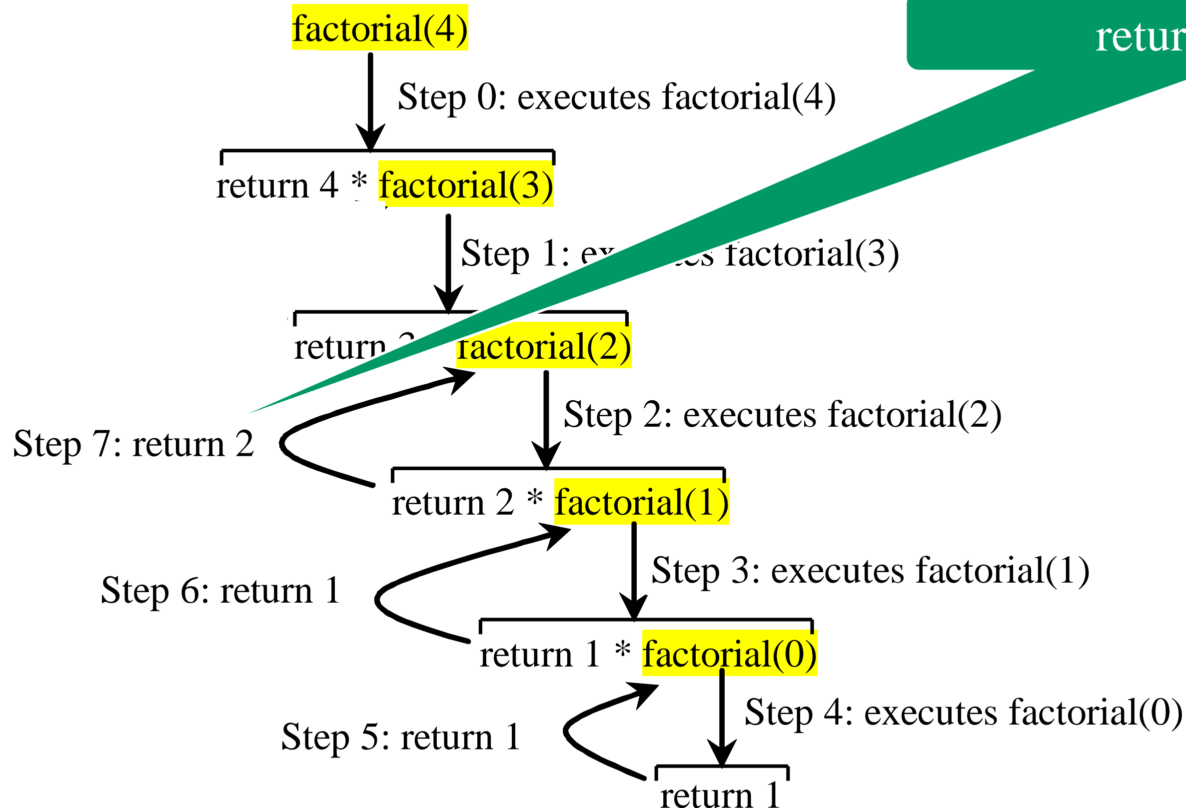
returns factorial(1)

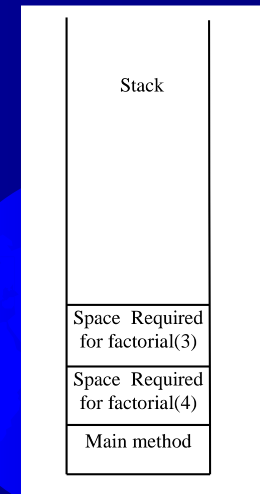
Stack

Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

animation

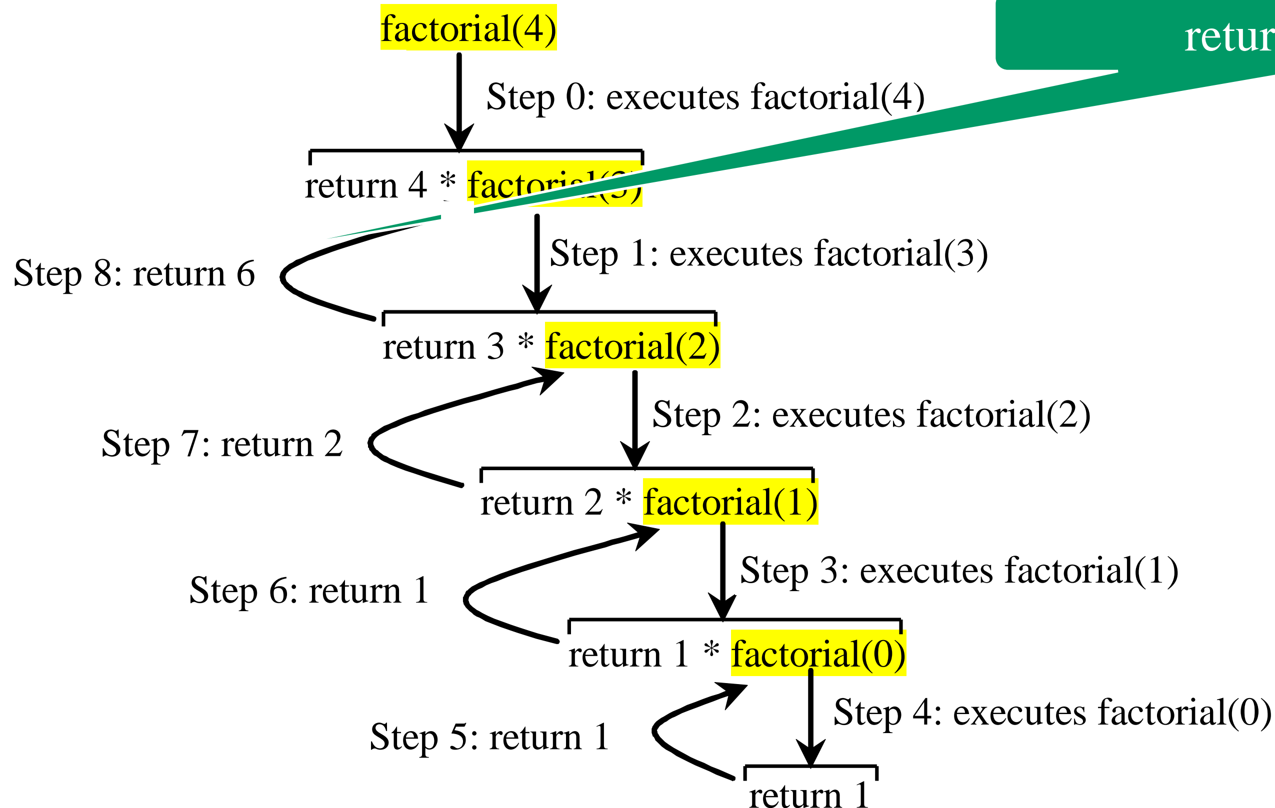
재귀호출 factorial 함수 Trace





animation

재귀호출 factorial 함수 Trace



returns factorial(3)

Stack

Space Required for factorial(4)

Main method

animation

재귀호출 factorial 함수 Trace

returns factorial(4)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

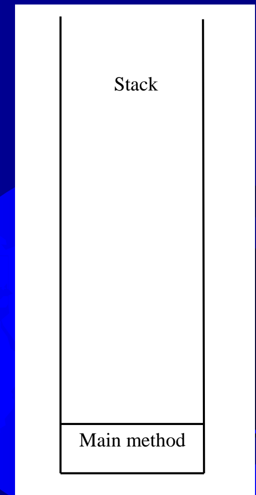
Step 6: return 1

return 1 * factorial(0)

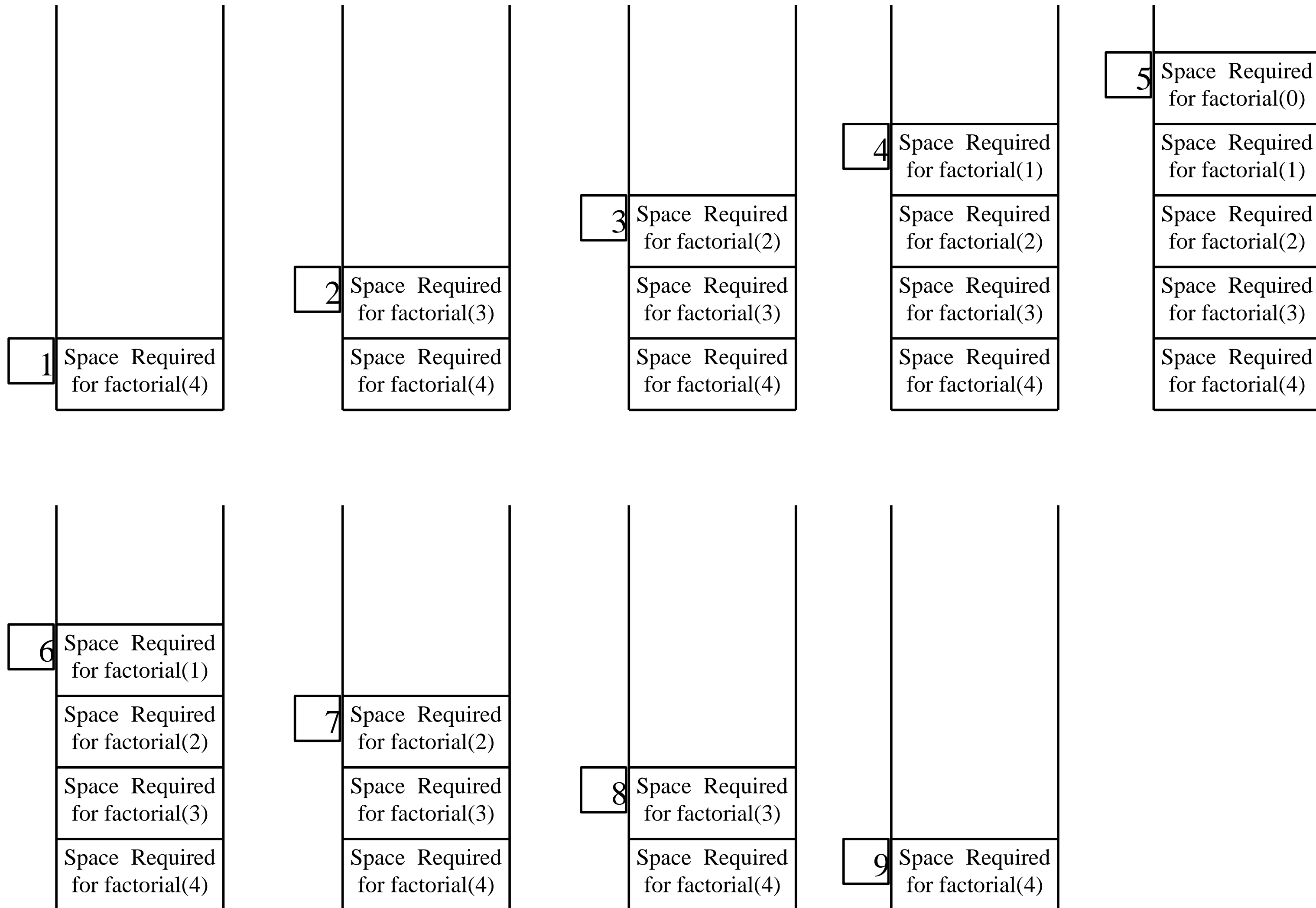
Step 4: executes factorial(0)

Step 5: return 1

return 1



factorial(4)의 스택 상태



재귀호출 factorial 함수 규칙

f(0) = 0;

f(n) = n + f(n-1);

피보나치 수

피보나치 급수: 0 1 1 2 3 5 8 13 21 34 55 89…

인덱스: 0 1 2 3 4 5 6 7 8 9 10 11

fib(0) = 0;

fib(1) = 1;

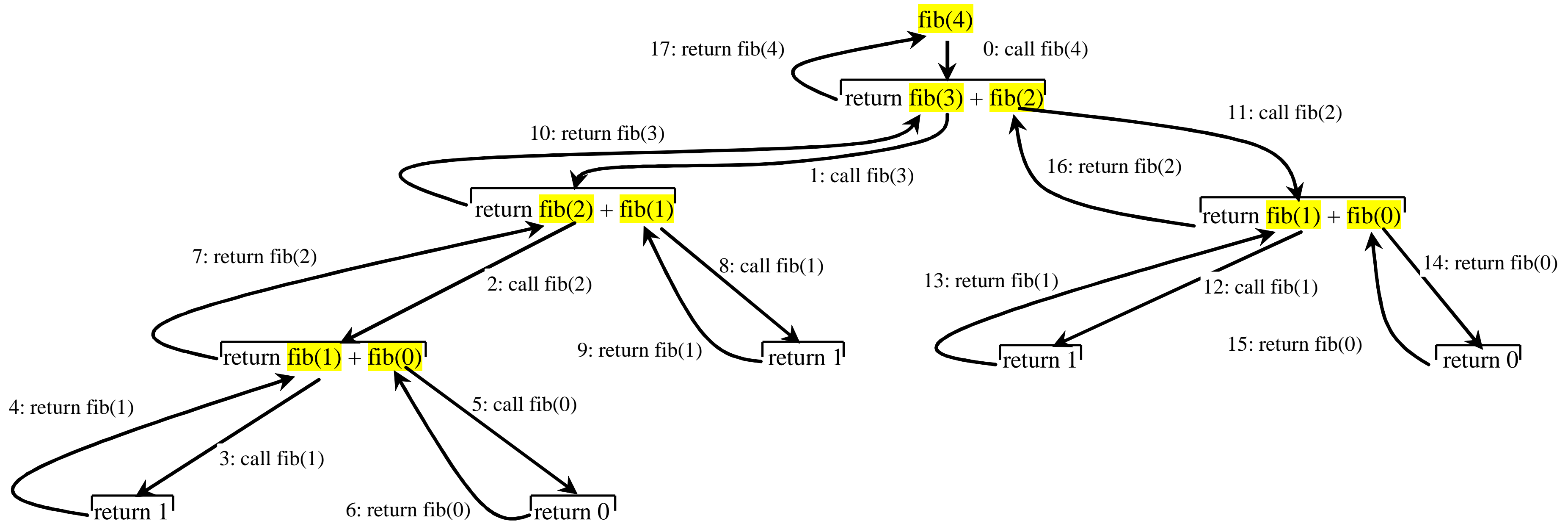
fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1) = (fib(1) + fib(0)) + fib(1) = (1 + 0) +fib(1) = 1 + fib(1) = 1 + 1 = 2

ComputeFibonacci

Run

피보나치 수, cont.



재귀호출의 특징

- 모든 재귀 함수는 다음의 특징을 가지고 있다.
  - 재귀 함수는 여러 경우를 처리할 수 있게 하는 if-else나 switch 문을 사용하여 구현된다.
  - 하나 이상의 종료 조건을 사용하여 재귀 호출을 종료하도록 한다.
  - 모든 재귀 호출은 원 문제를 부분 문제로 줄여가면서 종료 조건이 될 때까지 반복한다.
- 일반적으로 재귀 호출을 사용하여 문제를 해결하려면 우선 부분 문제로 나누어야 한다. 부분 문제가 원 문제와 닮아 있으면 부분 문제를 재귀적으로 해결하기 위해 원 문제에서와 같은 방법을 적용할 수 있다. 이 부분 문제는 원 문제보다 크기는 작지만 문제의 내용은 본질적으로 거의 같게 된다.

재귀호출을 사용한 문제 해결

n번 메시지를 출력하는 간단한 문제를 생각해 보자. 이 문제를 두 개의 부분 문제로 분할할 수 있다. 하나는 메시지를 한 번 출력하는 것이고, 다른 하나는 n-1번 출력하는 것이다. 두 번째 부분 문제는 원 문제와 매우 유사하고 크기가 작다. 이 문제의 종료 조건은 n==0일 때로 하면 될 것이다. 이 문제를 다음과 같이 재귀호출을 사용하여 해결해 보자.

```
void nPrintln(char * message, int times)
{
  if (times >= 1) {
    cout << message << endl;
    nPrintln(message, times - 1);
  } // The base case is n == 0
}
```

재귀적인 사고

☞ 이전에 나온 많은 문제들도 재귀적인 사고만 한다면 재귀호출 형태로 문제를 해결할 수 있다. 리스트 7.16의 회문(palindrome) 문제도 재귀호출을 사용하여 다음과 같이 작성할 수 있다.

```
bool isPalindrome(const char * const s)
{
  if (strlen(s) <= 1) // Base case
    return true;
  else if (s[0] != s[strlen(s) - 1]) // Base case
    return false;
  else
    return isPalindrome(substring(s, 1, strlen(s) - 2));
}
```

재귀 보조 함수

이전의 isPalindrome 재귀 함수는 재귀 호출 때마다 새로운 문자열을 만들어야 하기 때문에 효율적이지 않다. 새 문자열 생성을 피하기 위해, low와 high 인덱스를 사용하여 부분 문자열의 범위를 알려주는 재귀 보조 함수를 사용할 수 있다.

```
bool isPalindrome(const char * const s, int low, int high)
{
  if (high <= low) // Base case
    return true;
  else if (s[low] != s[high]) // Base case
    return false;
  else
    return isPalindrome(s, low + 1, high - 1);
}
bool isPalindrome(const char * const s)
{
  return isPalindrome(s, 0, strlen(s) - 1);
}
```

선택 정렬(재귀호출)

- 목록에서 최대 요소 값을 찾아 맨 뒤 요소와 교환
- 마지막 요소를 무시하고 남아 있는 목록에 대해 재귀적으로 정렬을 수행

RecursiveSelectionSort

이진 탐색(재귀호출)

- 경우 1: 키가 중앙 요소보다 작으면 배열의 처음 절반에서 재귀 검색을 계속한다.
- 경우 2: 키가 중앙 요소와 같으면 원하는 요소를 찾았으므로 검색이 완료된다.
- 경우 3: 키가 중앙 요소보다 크면 배열의 나머지 절반에서 재귀 검색을 계속한다.

RecursiveBinarySort

Optional

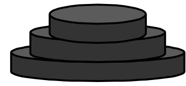
이진 탐색 구현

```
int binarySearch(const int list[], int key, int low, int high)
{
  if (low > high)  // The list has been exhausted without a match
    return -low - 1; // Return -insertion point - 1
  int mid = (low + high) / 2;
  if (key < list[mid])
    return binarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return binarySearch(list, key, mid + 1, high);
}
int binarySearch(const int list[], int key, int size)
{
  int low = 0;
  int high = size - 1;
  return binarySearch(list, key, low, high);
}
```

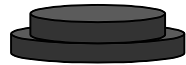
하노이 탑

- 1, 2, 3, ..., n이 표시된 n개의 디스크가 있고, A, B, C 세 탑이 있다.
- 디스크 위에는 항상 자기보다 작은 디스크가 와야 한다.
- 초기에 모든 디스크는 A 탑에 놓여 있다.
- 한 번에 한 디스크씩만 이동시켜야 하며, 탑의 제일 위로 이동되어야 한다.

하노이 탑,



Step 0: Starting status



Step 1: Move disk 1 from A to B

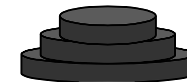
Step 2: Move disk 2 from A to C

Step 3: Move disk 1 from B to C

Step 4: Move disk 3 from A to B

Step 5: Move disk 1 from C to A

Step 6: Move disk 2 from C to B



Step 7: Mve disk 1 from A to B

하노이 탑 구현

TowersOfHanoi

Run