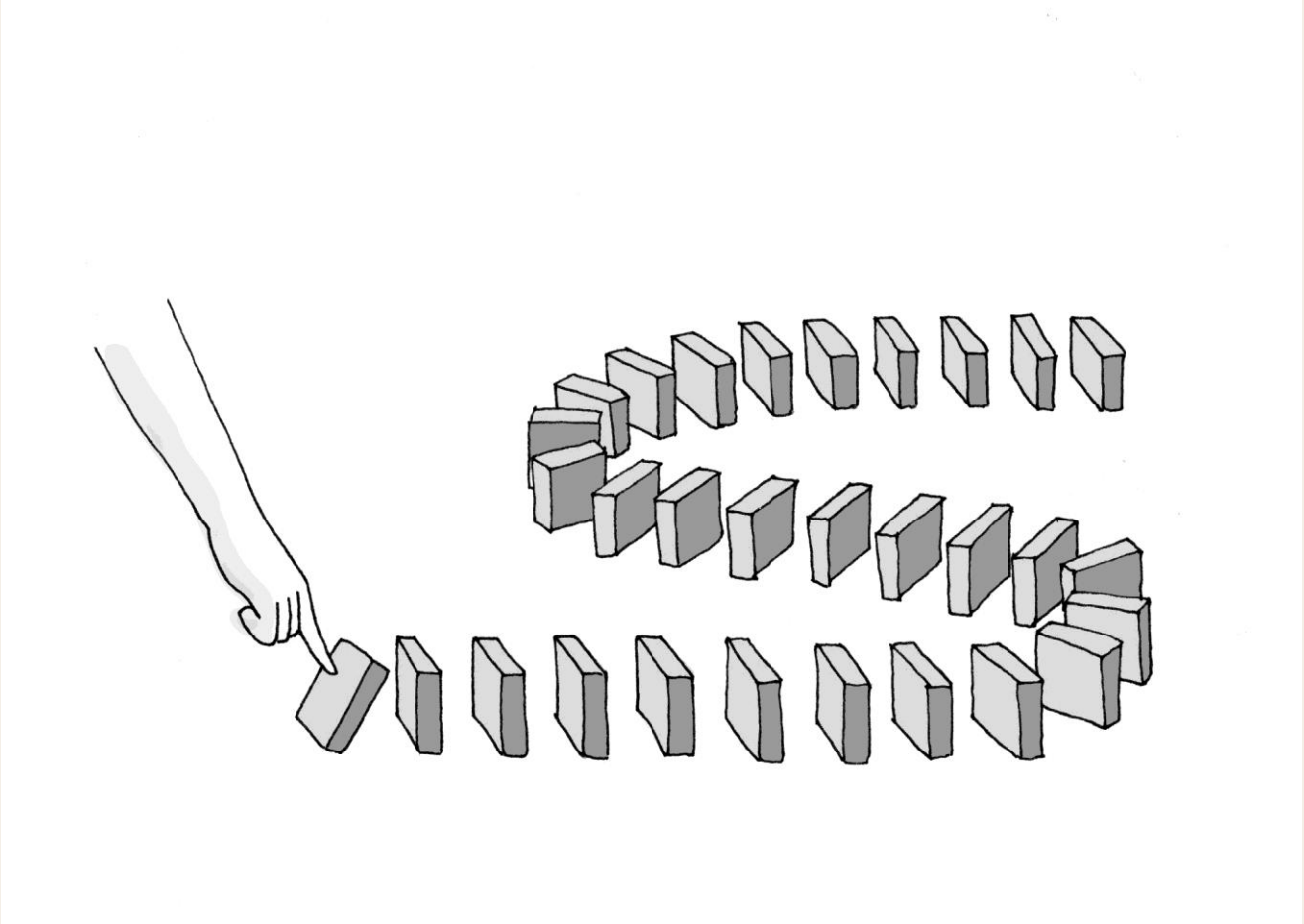
4장. 재귀호출

- 대표적인 분할정복 알고리즘

- 학습목표
  - 재귀호출이라는 개념 자체를 명확히 이해한다.
  - 재귀 호출함수의 내부구조를 이해한다.
  - 재귀호출에 내재하는 효율성에 대해 이해한다.

도미노

- 도미노



도미노

- 100번째 것이 반드시 쓰러진다는 사실을 증명하라

- 수학적 귀납법(Mathematical Induction)
  - 처음 것(K=1)은 반드시 쓰러진다.
  - K 번째 막대가 쓰러지면 (K+1)번째 막대도 반드시 쓰러진다

- 재귀적 알고리즘(Recursive Algorithm)
  - 수학적 귀납법의 순서를 역순으로 적용
  - 99번째 것이 쓰러지면 인접한 100번째 것이 쓰러지니, 99번째 것이 반드시 쓰러진다는 사실을 증명하라
  - 98번째 것이 쓰러지면 인접한 99번째 것이 쓰러지니, 98번째 것이 반드시 쓰러진다는 사실을 증명하라
  - ...
  - 처음 것이 반드시 쓰러진다는 사실을 증명하라
  - 그건 직접 밀었기 때문에 반드시 쓰러진다.

재귀호출

- **분할정복**
  - 문제의 크기 N
  - 큰 문제를 작은 문제로 환원
  - 작은 문제 역시 큰 문제와 유사함
- **재귀호출**
  - Self Call
  - Boomerang
- **아주 작은 문제**
  - 직접 해결할 정도로 작아짐
  - 베이스 케이스(디제너릿 케이스)

이진탐색

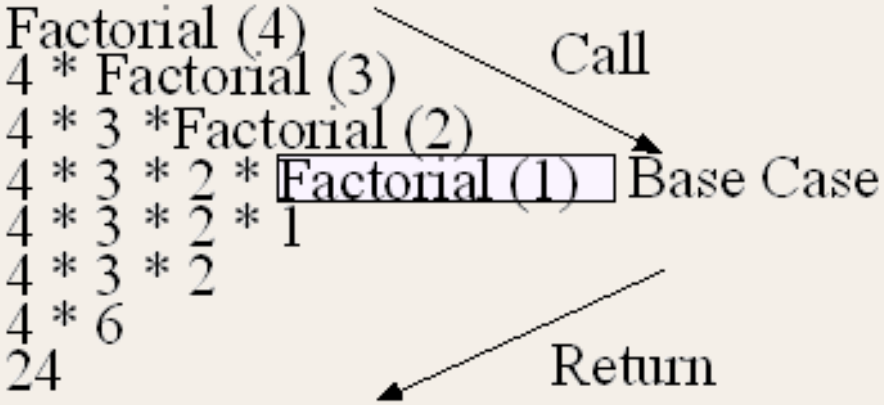
```
BinarySearch(SearchRange)                    괄호 안은 탐색범위
{    if (One Page)                                베이스 케이스
          Scan Until Found;
       else
     {  Unfold the Middle Page;                  가운데 펼침
          Determine Which Half;                  전반부, 후반부 판단
           if First Half
                BinarySearch(First Half);        전반부 재귀호출
           else BinarySearch(Second Half);       후반부 재귀호출
     }
}
```

- 문제 크기 감소: N, N/2, N/4, …., 1
- 재귀호출은 반드시 베이스 케이스에 도달해야 함

팩토리얼 연산

- n! = n • (n-1) • (n-2) • ... • 1 (단, 1! = 0! = 1)

-
```
int Factorial(int n)
{ if (n = = 1)
    return 1;
  else
   return(n * Factorial(n-1));
  }
```



활성화 레코드

```
int Factorial(int n)
{ if (n = = 1)
    return 1;
  else
   return(n * Factorial(n-1));
  }
```

<table>
<tr><td colspan="8">Stack Expands □</td></tr>
<tr><td>Parm. n = 4
Ret. Val = ?</td><td>□</td><td>Parm. n = 3
Ret. Val = ?</td><td>□</td><td>Parm. n = 2
Ret. Val = ?</td><td>□</td><td>Parm. n = 1
Ret. Val = ?</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>□</td></tr>
<tr><td>Parm. n = 4
Ret. Val = 4 * 6</td><td>□</td><td>Parm. n = 3
Ret. Val = 3 * 2</td><td>□</td><td>Parm. n = 2
Ret. Val = 2 * 1</td><td>□</td><td>Parm. n = 1
Ret. Val = 1</td></tr>
<tr><td colspan="8">Stack Shrinks</td></tr>
</table>

문자열 뒤집기 I

```
void Reverse(char S[ ], int Size)
{ if (Size = = 0)
     return;                    호출함수로 되돌아감
  else
  {  printf("%c", S[Size-1]);    마지막 문자를 쓰기
     Reverse(S, Size-1);        재귀호출
   }
 }
```

- 마지막 문자를 먼저 제거
  - 문자열 "PET"에 대해서 추적해 보라.

문자열 뒤집기 II

```
void Reverse(char S[ ], int First, int Last)
{ if (First > Last)
    return;
  else
  {  printf("%c", S[First]);
     Reverse(S, First+1, Last);
  }
 }
```

- 첫 문자를 먼저 제거
  - 위 코드는 제대로 돌지 않음
  - 어떻게 고쳐야 하는가.

문자열 뒤집기 II

```
void Reverse(char S[ ], int First, int Last)
{ if (First > Last)
    return;
  else
  {  printf("%c", S[First]);
     Reverse(S, First+1, Last);
  }
 }
```

First = 0 Last = 3 Reverse(S, 1, 3)		First = 1 Last = 3 Reverse(S, 2, 3)	□	First = 2 Last = 3 Reverse(S, 3, 3)	□	First = 3 Last = 3 Reverse(S, 4, 3)	□	First = 4 Last = 3
								□
First = 0 Last = 3 printf(S[0])		First = 1 Last = 3 printf(S[1])	□	First = 2 Last = 3 printf(S[2])	□	First = 3 Last = 3 printf(S[3])	□	First = 4 Fast = 3 return

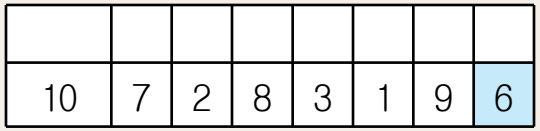
K 번째 작은 수 찾기

- 10, 7, 2, 8, 3, 1, 9, 6 이라는 숫자 중에서 세 번째 작은 수는 3

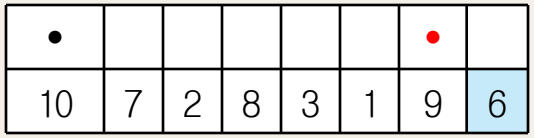
- 재귀적 방법론
  - 10, 7, 2, 8 과 3, 1, 9, 6으로 분할
  - 10, 7, 2, 8 중 세 번째 작은 수는 8
  - 3, 1, 9, 6 중 세 번째 작은 수는 6
  - 작은 문제의 해결책이 큰 문제의 해결책으로 이어지지 않는다.

파티션

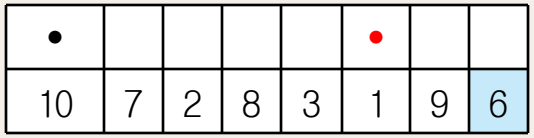
❑ 1) 임의로 피벗 설정



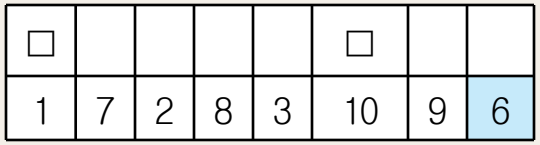
❑ 2) 다운 포인터와 업 포인터설정



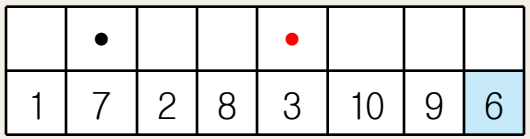
❑ 3) 다운은 피벗보다 작거나 같은 것,
업은 피벗보다 크거나 같은 것 찾음

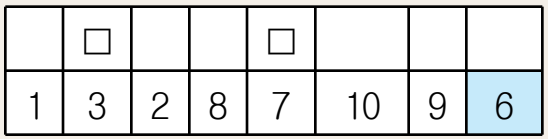


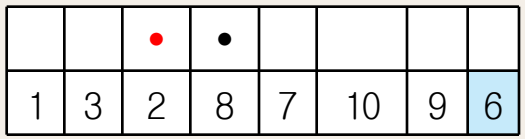
❑ 4) 스와핑



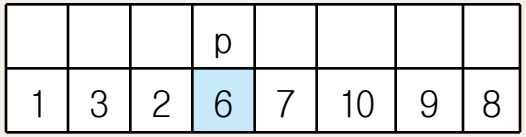
❑ 5) 포인터가 일치하거나 교차할 때까지 3),4)를 반복







❑ 6) 업 포인터 위치에 있는 숫자와 피벗을 스와핑



파티션

❑ 파티션

- 피벗보다 작은 것은 왼쪽으로, 피벗보다 큰 것은 오른쪽으로
- 전체 데이터가 정렬된 상태는 아님
- 모든 데이터가 정렬되어도 피벗 위치는 불변
- K가 4라면 네 번째 작은 수를 이미 찾은 것임

			p				
1	3	2	6	7	10	9	8

❑ 세번째 작은 수 찾기

- 분할된 왼쪽에 대해서 다시 파티션을 가함 (결과 p = 2)

•	•	
1	3	2

	p	
1	2	3

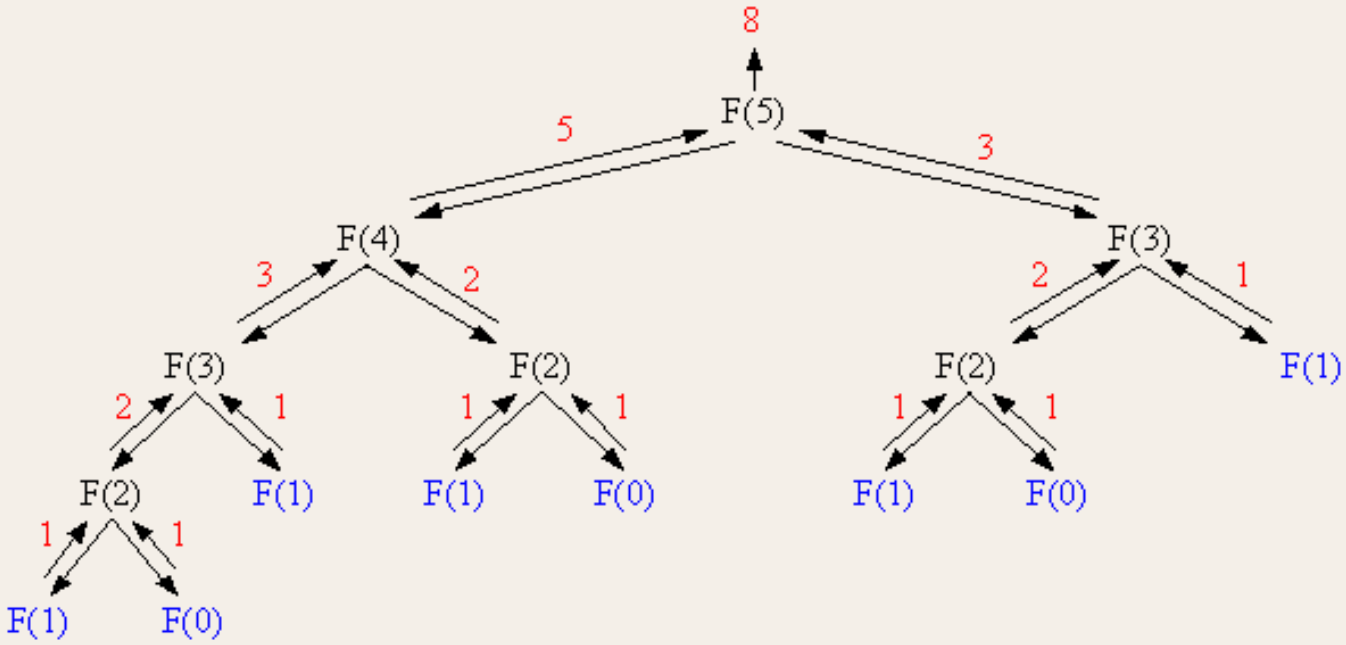
- 분할된 오른쪽에 대해서 다시 파티션을 가함: self-swap (결과 p = 3)

p
3

피보나치 수열

- F(n) = F(n-1) + F(n-2) (단, F(0) = 0, F(1) = 1)

- int Fibonacci(int n)
  { if (n < 2) 베이스 케이스
    return 1; F(0) = F(1) = 1
   else return (Fibonacci(n-1) + Fibonacci(n-2)); 재귀호출
  }



재귀함수 작성

- Step 1
  - 더 작은 문제로 표시할 수 있는지 시도
    - 문제 크기를 하나씩 줄이는 방법
    - 반으로 줄이는 방법
    - 다른 여러 개의 작은 문제의 조합으로 표시하는 방법
  - 문제 크기 파라미터 N을 확인
- Step 2
  - 문제를 직접 풀 수 있는 것이 어떤 경우인지 베이스 케이스 확인
- Step 3
  - N이 줄어서 반드시 베이스 케이스를 만나는지 확인
  - N이 양수인지 음수인지, 짝수인지 홀수인지, 또는 부동소수인지 정수인지 모든 경우에 대해 모두 검증.
- Step 4
  - 베이스 케이스와 베이스 케이스가 아닌 경우를 나누어서 코드를 작성

재귀호출의 효율성

❑ 활성화 레코드의 비효율

- 공간적 비효율(저장공간)
- 시간적 비효율(저장, 복원에 걸리는 시간)
- 가능하다면 반복문으로 대치하는 것이 유리

재귀호출의 반복문 변환

```
int Factorial(int n)              팩토리얼
{ int product = 1;                      곱셈의 결과 값을 초기화
  for (int i = 1; i <= n; i++)          1부터 n까지
    product *= i;               계속 곱해서 저장
  return product;                       결과를 리턴
}


void Reverse(char S[ ], int Size) 문자열 뒤집기
{ while (Size > 0)                      한 글자라도 남아 있을 때까지
  { printf("%c", S[Size-1]);            일단 마지막 문자를 찍고
   --Size;                      문자열 마지막을 한 칸 앞으로
  }
}


int Fibonacci(int n)              피보나치 수열
{ int A[Max];                   배열 크기를 n보다 크게 잡음
  F[0] = 1; F[1] = 1;                   수열의 처음 두 숫자 초기화
  for (int i = 2; i <= n; i++)          F[2]부터 n까지
    F[i] = F[i-2] + F[i-1];             앞에서 뒤로 채워나감
  return (F[i]);                        배열의 마지막 요소를 돌려줌
}
```

꼬리 재귀

❑ 재귀호출 명령이 함수 마지막에 위치

- 되돌아올 때 할 일이 없는 재귀호출
- 새로운 활성화 레코드 공간을 만들지 않고 이전 공간 재사용
- return (N * Factorial(N-1)); 는 꼬리재귀 아님
- Factorial(N-1) 결과가 리턴 되면 거기에 N을 곱하는 일이 남아 있음.

❑ 꼬리 재귀를 사용한 팩토리얼

```
int Factorial(int n, int a)
{ if (n = = 0)
    return a;                          a에 결과 값이 축적됨
  else
    return Factorial(n-1, n*a);        꼬리 재귀
}
```