반복과 재귀

n! 계산

- 계승 (factorial)
  - 5! = 5 x 4 x 3 x 2 x 1
  - n! = n x (n-1) x (n-2) x … x 1
- 반복(iteration) 기반 구현

```
public static void main(String[] args) {
        System.out.println(fact(10));
}
private static double fact(int n) {
        double     v=1;
        for (int i = 2; i <= n; i++) v*=i;
        return v;
}
```

n! 계산

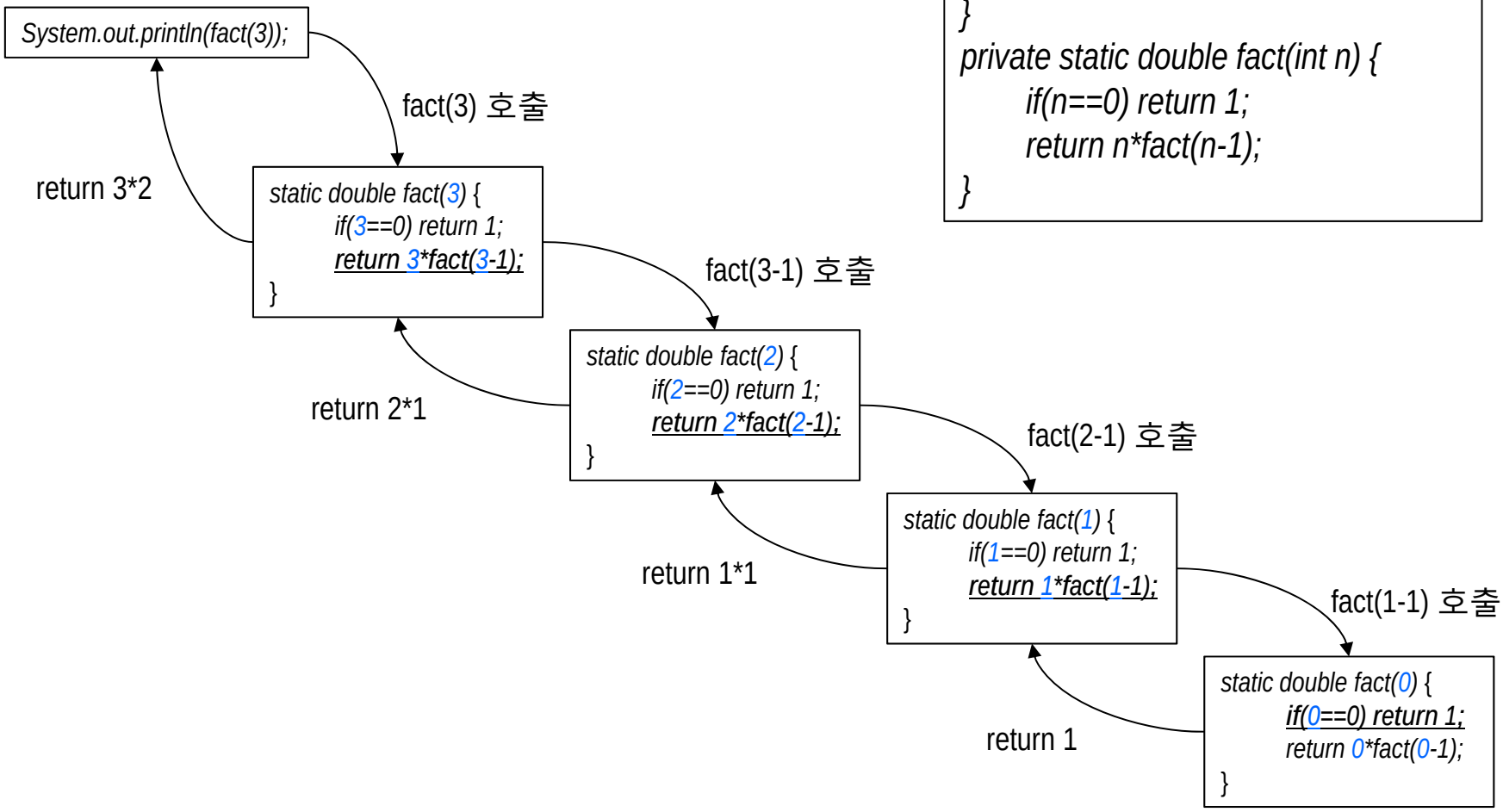
- 계승 (factorial)
  - 5! = 5 x 4 x 3 x 2 x 1
  - n! = n x (n-1) x (n-2) x … x 1
- 재귀(recursion) 기반 구현
  - 재귀식(recurrence relation)
    - n=0: fact(n) = 1 ← Base case: 0! = 1
    - n>0: fact(n) = n x fact(n-1) ← Recursive case: n! = n x (n-1)!

```
public static void main(String[] args) {
        System.out.println(fact(10));
}
private static double fact(int n) {
        if(n==0) return 1;
        return n*fact(n-1);
}
```

n! 계산

- 재귀함수 호출 및 반환

```
public static void main(String[] args) {
        System.out.println(fact(3));
}
private static double fact(int n) {
        if(n==0) return 1;
        return n*fact(n-1);
}
```



거듭제곱 계산

- 거듭제곱
  - $2^5 = \overbrace{2 \times 2 \times 2 \times 2 \times 2}^{5}$
  - $m^n = \overbrace{m \times m \times \ldots \times m}^{n}$

- 반복 기반 구현

```
public static void main(String[] args) {
        System.out.println(power(2, 10));
}
private static long power(int m, int n) {
        long v=1;
        for (int i = 0; i < n; i++) v*=m;
        return v;
}
```

거듭제곱 계산

- 거듭제곱
  - $2^5 = \overbrace{2 \times 2 \times 2 \times 2 \times 2}^{5}$
  - $m^n = \overbrace{m \times m \times \ldots \times m}^{n}$

```
public static void main(String[] args) {
        System.out.println(power(2, 60));
}
private static long power(int m, int n) {
        if(n==0) return 1;
        return m*power(m,n-1);
}
```

- 재귀 기반 구현
  - 재귀식
    - n=0: power(m, n) = 1 ← Base case: $m^0 = 1$
    - n>0: power(m, n) = m x power(m, n-1) ← Recursive case: $m^n = m \times m^{n-1}$

거듭제곱 계산

- 거듭제곱
  - $2^5 = \overbrace{2 \times 2 \times 2 \times 2 \times 2}^{5}$
  - $m^n = \overbrace{m \times m \times \ldots \times m}^{n}$

```
public class Test {
    public static void main(String[] args) {
        int    m=2, n=60;
        System.out.println(power(m,n));
    }
    private static long power(long m, long n) {
        if(n==0) return 1;
        return power(m*m,n/2)*(n%2==0? 1:m);
    }
}
```

- 재귀 기반 구현
  - 재귀식
    - n=0: power(m, n) = 1
    - n>0, n이 짝수: power(m, n) = power(m x m, n/2)
    - n>0, n이 홀수: power(m, n) = power(m x m, n/2) x m

거듭제곱 계산

- 거듭제곱 계산 #1
  - 재귀식
    - n=0: power(m, n) = 1
    - n>0: power(m, n) = m x power(m, n-1)
  - 예
    - 2^{10} → 2 x 2^{9} → 2 x 2 x 2^{8} → … → 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2
- 거듭제곱 계산 #1
  - 재귀식
    - n=0: power(m, n) = 1
    - n>0, n이 짝수: power(m, n) = power(m x m, n/2)
    - n>0, n이 홀수: power(m, n) = power(m x m, n/2) x m
  - 예
    - 2^{10} → 4^{5} → 4 x 16^{2} → 4 x 256
    - 2^{15} → 2 x 4^{7} → 2 x 4 x 16^{3} → 2 x 4 x 16 x 256

재귀적 접근법 연습: 배열 총합

- 배열 총합
  - 재귀식
    - i=0: sum(n, i)=n[0]
    - i>0: sum(n, i) = n[i] + sum(n, i-1)
  - 최초 sum(n, n.length-1)를 호출하여 배열 총합 계산

- Base case: 0번째까지 수의 총합은 n[0]과 같다
- Recursive case: 배열 n의 0번째부터 i번째까지 수의 총합은 배열 n의 0번째부터 i-1번째까지 수의 총합과 배열 n의 i번째 수를 더한 값이다

```
public class Test {
        public static void main(String[] args) {
                int      n[]={7,2,8,4,1};
                System.out.println(sum(n, n.length-1));
        }
        private static int sum(int[] n, int i) {
        }
}
```

최초 sum(n, 0) 호출을 통해 배열 총합을 계산하려면 위 재귀식을 어떻게 수정해야 하는가?

```
public class Test {
        public static void main(String[] args) {
                int      n[]={1,2,3,4,5};
                System.out.println(sum(n, 0));
        }
        private static int sum(int[] n, int i) {
        }
}
```

재귀적 접근법 연습: 배열 최대값

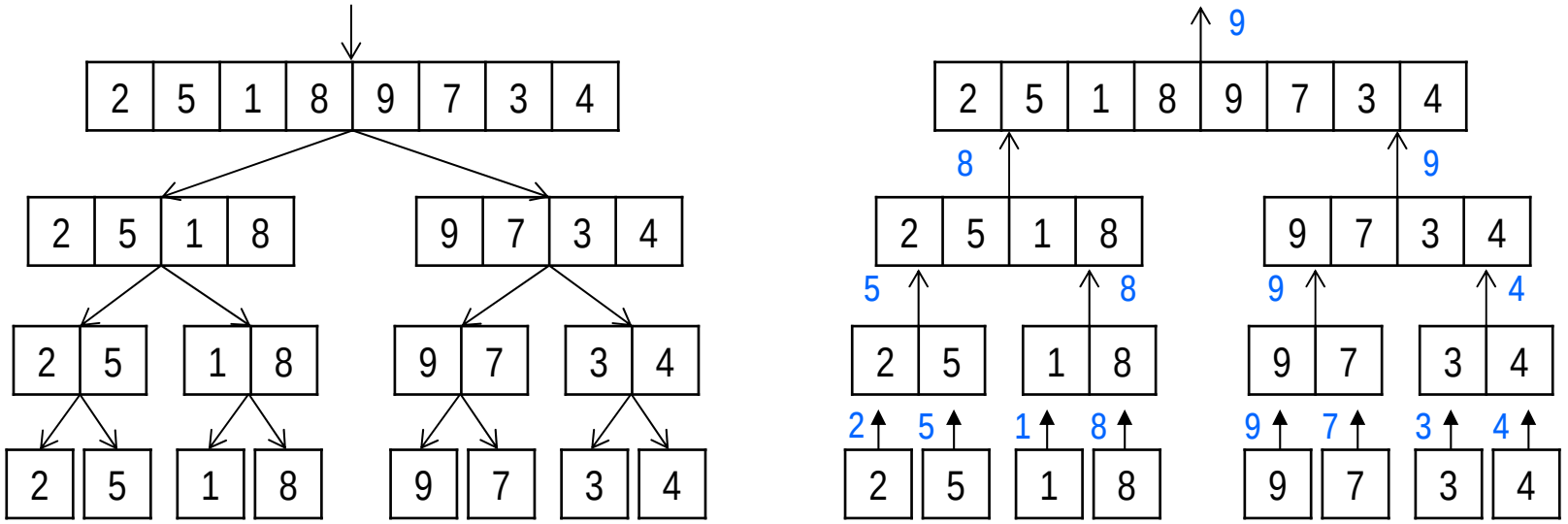
- 배열 최대값
  - 재귀식
    - i=0: 0번째까지 수의 최대값은 0번째 수이다.
    - i>0: 0번째부터 i번째까지 수의 최대값은
      - 0번째부터 i-1번째까지 수의 최대값과 i번째 수 중 큰 값

```
public class Test {
        public static void main(String[] args) {
                int    n[]={5,1,8,3,2};
                System.out.println(max(n, n.length-1));
        }
        private static int max(int[] n, int i) {



        }
}
```

재귀적 접근법 연습: 배열 최대값

- 재귀적 접근
  - 배열의 최대값은 왼쪽 반의 최대값과 오른쪽 반의 최대값 중 큰 값

```
public class Test {
        public static void main(String[] args) {
                int     n[]={2,5,1,8,9,7,3,4};
                System.out.println(max(n, 0, n.length-1));
        }
        private static int max(int[] n, int left, int right) { … }
}
```

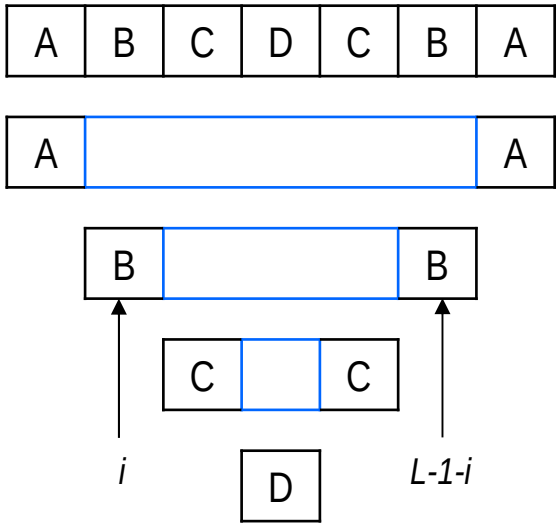


재귀적 접근법 연습

- 재귀적 접근법 연습
  - 배열 내 소문자를 대문자로 변환
  - 배열 내 대문자 출현 횟수

재귀적 접근법 연습: 회문 검사

- 회문(palindrome)
  - 순방향, 역방향 모두 같은 순서열이 되는 문자열
    - ABBA, ABCBA, AAA
- 회문 검사
  - 재귀적 접근

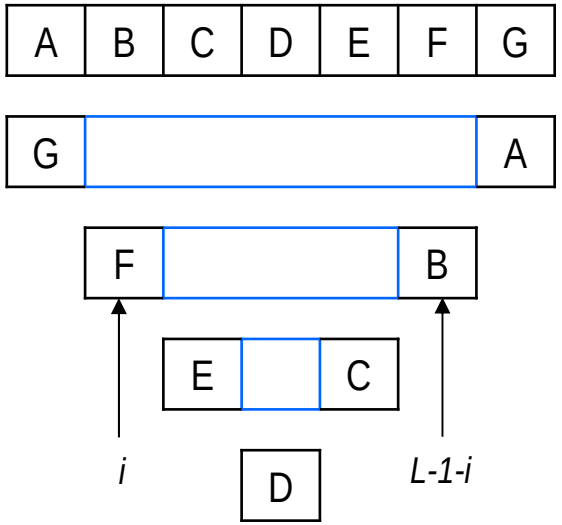


- Recursive case: 문자열이 회문이 되려면 양 끝 문자가 같아야 하고 양 끝 문자를 제외한 나머지 문자열이 회문이여야 한다
- Base case: 길이 1 이하 문자열은 회문이다

```
public class Test {
        public static void main(String[] args) {
                String s="ABCDEDCBA";
                System.out.println(palindrome(s, 0));
        }
        private static boolean palindrome(String s, int i) {
        }
}
```

재귀적 접근법 연습: 문자열 역순

- 문자열 역순 (reverse) 변환
  - 예
    - ABCDE → EDCBA
  - 재귀적 접근



- Recursive case: 양 끝 문자 교환 후 양 끝 문자를 제외한 나머지 문자열을 역순 처리한다
- Base case: 길이 1 이하 문자열은 이미 역순 문자열이다

```
public class Test {
        public static void main(String[] args) {
                int     n[]={1,2,3,4,5,6,7,8};
                reverse(n, 0);
                System.out.println(Arrays.toString(n));
        }
        private static void reverse(int[] n, int i) {
        }
}
```

재귀적 접근법 연습: 최대공약수

- 최대공약수 (GCD, Greatest Common Divisor)
  - 재귀식
    - b=0: gcd(a, b) = a
    - b>0: gcd(a, b) = gcd(b, a%b)
    - 예
      - gcd(64,48)=gcd(48,16)=gcd(16,0)=16
  - 반복 기반 구현
  - 재귀 기반 구현

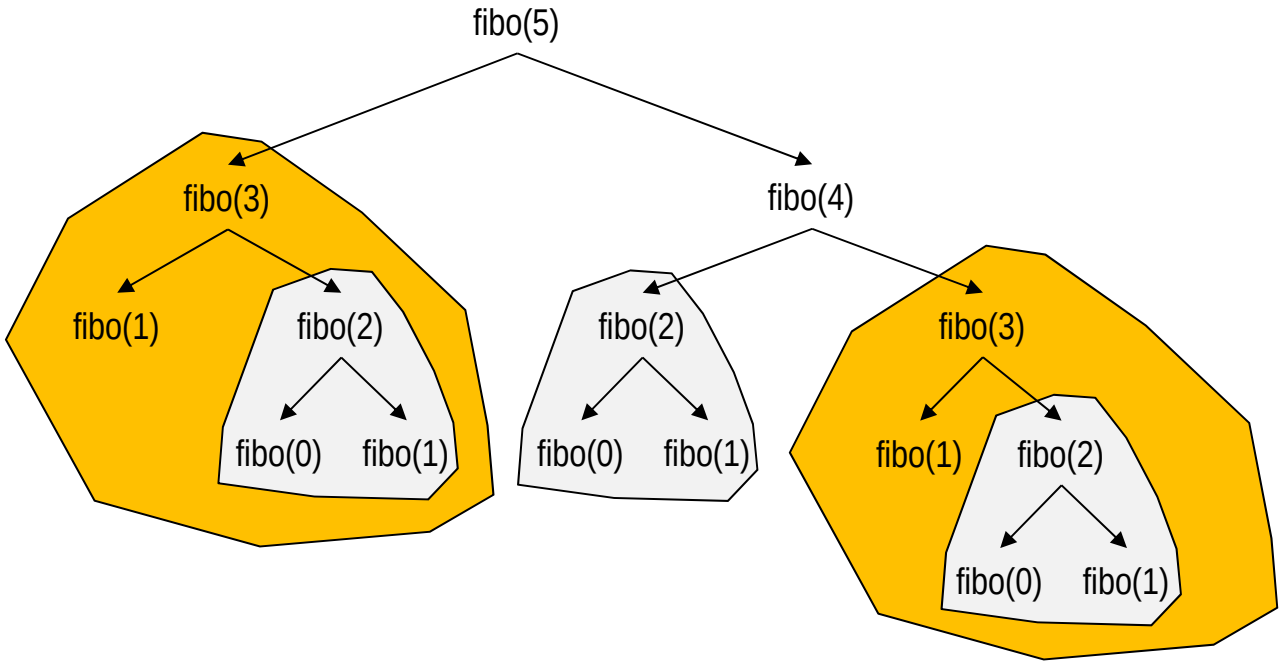
```
public class Test {
    public static void main(String[] args) {
        System.out.println(gcd(64,48));
    }
    private static int gcd(int a, int b) {
    }
}
```

피보나치 수열

- 피보나치 수열
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
    - 0번째 수=0, 6번째 수 = 8
- 특정 순번 피보나치 수 탐색 (0번째부터 시작)
  - 재귀식
    - n<=1: fibo(n) = n
    - n>=2: fibo(n) = fibo(n-2) + fibo(n-1)

피보나치 수열: 비효율적 재귀 구현

```
private static long fibo(int n) {
        if(n<2) return n;
        return fibo(n-2)+fibo(n-1);
}
```



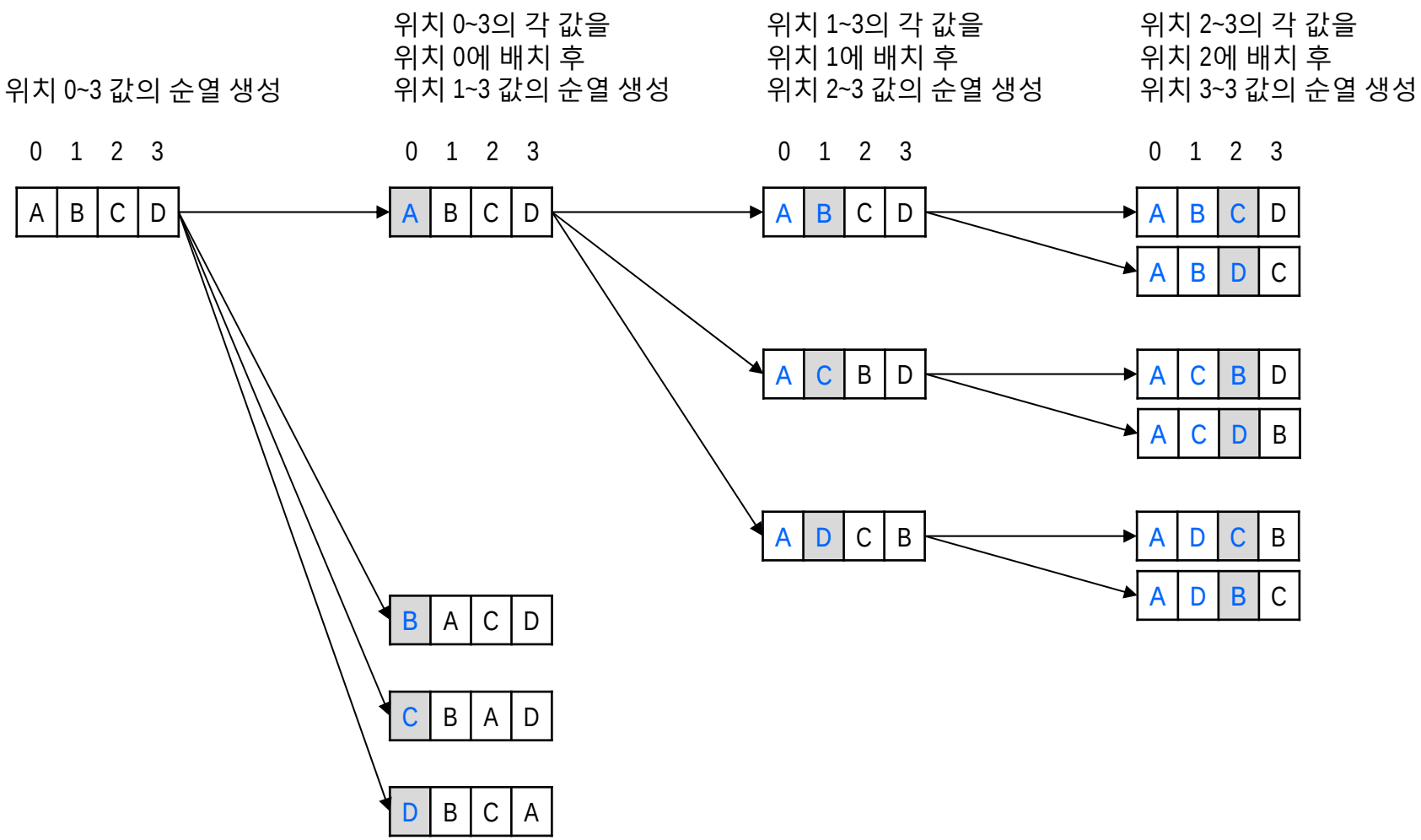
피보나치 수열: 효율적 재귀 구현


```
public class Test {
	public static void main(String[] args) {
		System.out.println(fibo(50));
	}
	private static long fibo(int n) {
		long	memo[]=new long[n+1];
		return fiboMemo(n, memo);
	}
	private static long fiboMemo(int n, long[] memo) {
		if(memo[n]>0) return memo[n];
		if(n<2) memo[n]=n;
		else memo[n]=fiboMemo(n-2, memo)+fiboMemo(n-1, memo);
		return memo[n];
	}
}
```

순열

- 순열(permutation)
  - 순서가 부여된 나열
  - 집합 내 모든 원소들에 순서를 부여한 나열
  - 집합 S에 대한 순열의 개수는 |S|!
  - 예
    - {A, B, C} → ABC, ACB, BAC, BCA, CAB, CBA

순열 생성


위치 0~3 값의 순열 생성
위치 0~3의 각 값을
위치 0에 배치 후
위치 1~3 값의 순열 생성
위치 1~3의 각 값을
위치 1에 배치 후
위치 2~3 값의 순열 생성
위치 2~3의 각 값을
위치 2에 배치 후
위치 3~3 값의 순열 생성
0 1 2 3
A B C D
B A C D
C B A D
D B C A
A B C D
A C B D
A D C B
A B C D
A B D C
A C B D
A C D B
A D C B
A D B C

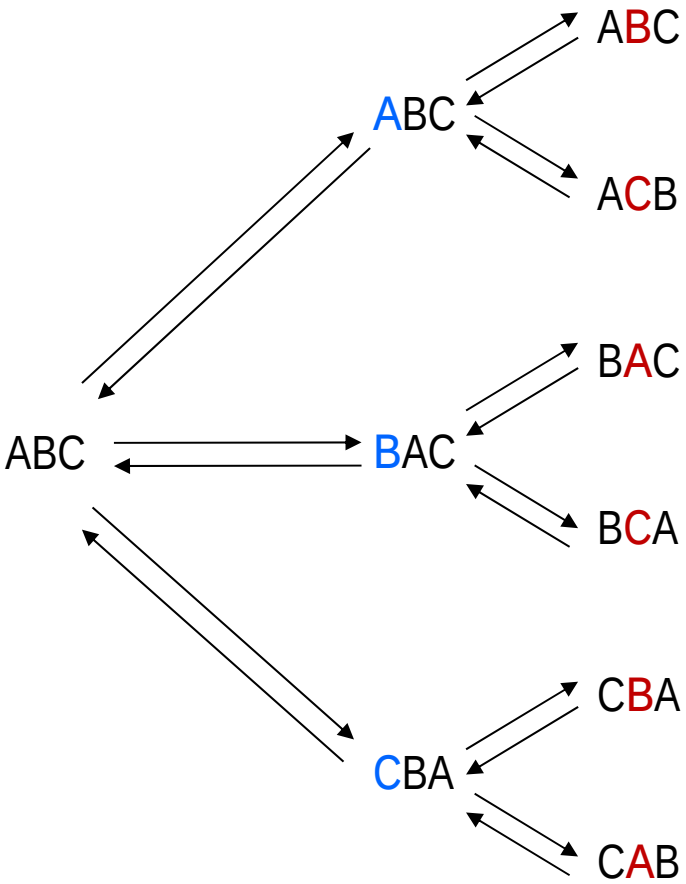
순열 생성

- 순열 생성
  - 배열 n에 저장된 값들의 순열 생성
    - perm(n, 0) → 배열 n의 위치 0부터 끝(n.length-1)까지 값들의 순열
  - perm(n, i) → 배열 n의 위치 i부터 끝까지 값들의 순열
    - i=n.length
      - 배열 n 내 특정 순열 처리
    - i<n.length
      - j = i ~ n.length-1
        - swap(n[i], n[j])
        - perm(n, i+1)
        - swap(n[i], n[j])

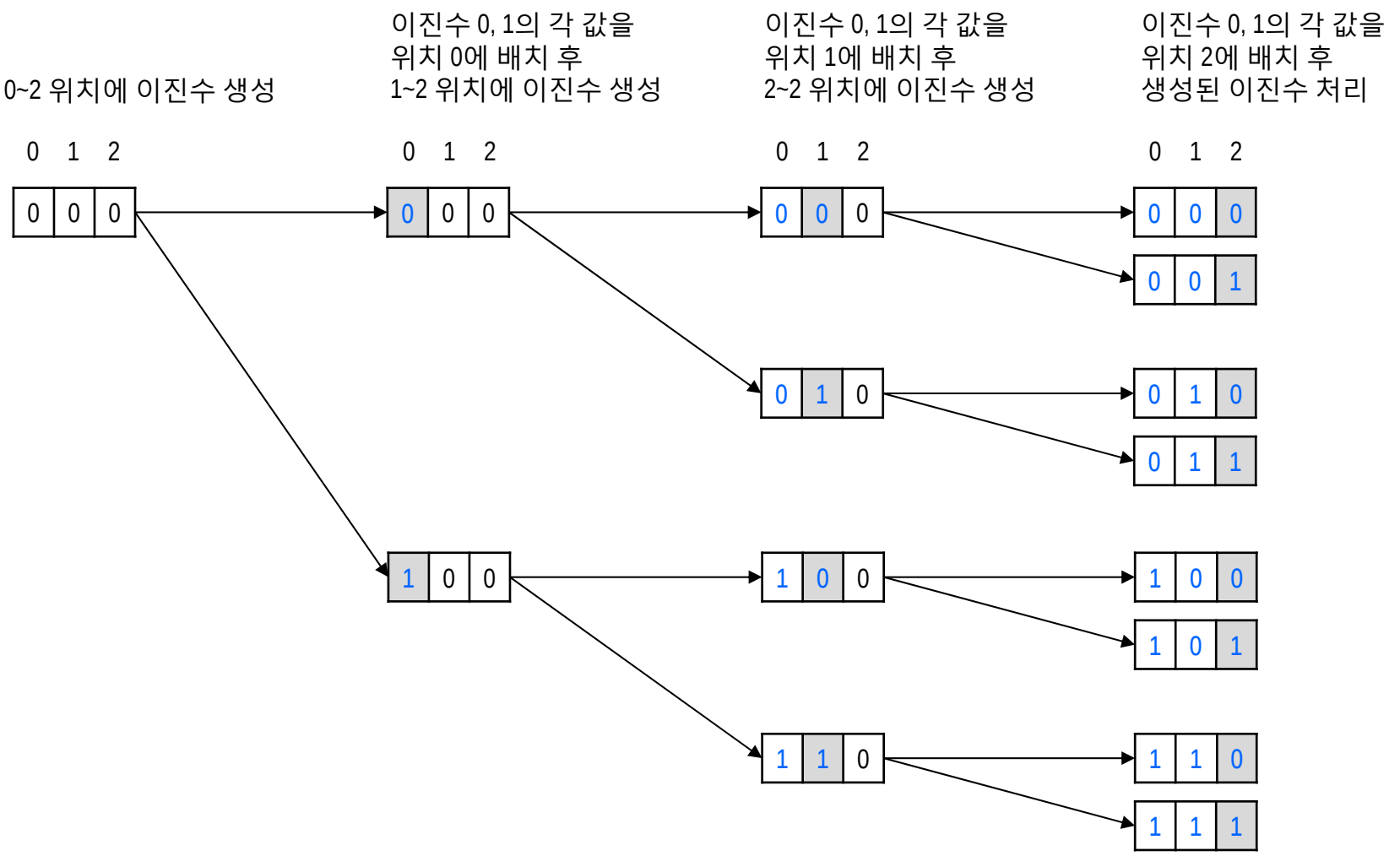
이 복원 단계 제거 후
순열 코드 실행해 보시오

```
public static void main(String[] args) {
        int     n[]=new int[3];
        for (int i = 0; i < n.length; i++) n[i]=i;
        perm(n, 0);
}
private static void perm(int[] n, int i) {
        if(i==n.length){
                System.out.println(Arrays.toString(n));
                return;
        }
        for (int j = i; j < n.length; j++) {
                int temp=n[i]; n[i]=n[j]; n[j]=temp; // swap
                perm(n, i+1);
                temp=n[i]; n[i]=n[j]; n[j]=temp; // swap
        }
}
```

순열 생성



3-비트 모든 이진수 생성


0~2 위치에 이진수 생성
이진수 0, 1의 각 값을
위치 0에 배치 후
1~2 위치에 이진수 생성
이진수 0, 1의 각 값을
위치 1에 배치 후
2~2 위치에 이진수 생성
이진수 0, 1의 각 값을
위치 2에 배치 후
생성된 이진수 처리
0 1 2
0 0 0
0 0 0
0 0 0
0 0 1
0 1 0
0 1 0
0 1 1
1 0 0
1 0 0
1 0 0
1 0 1
1 1 0
1 1 0
1 1 1

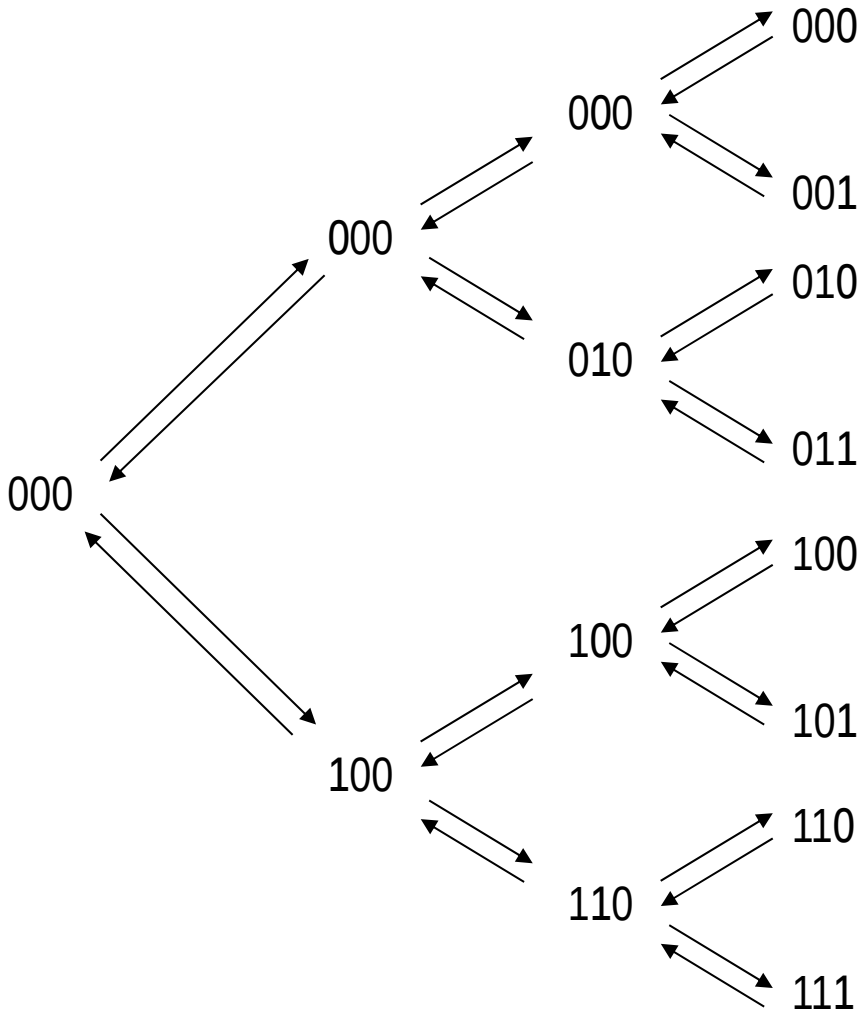
n-비트 이진수 생성

- n-비트 이진수 생성
  - 배열 v에 이진수 생성
    - bitString(v, 0) → 배열 v의 0부터 끝(v.length-1)까지 위치에 이진수 생성
  - bitString(v, i) → 배열 v의 i부터 끝까지 위치에 이진수 생성
    - i=v.length
      - 배열 v 내 이진수 처리
    - i<v.length
      - v[i]=0
      - bitString(v, i+1)
      - v[i]=1
      - bitString(v, i+1)
      - v[i]=0

- 이 복원 단계 불필요 이유는?
- 필요한 상황은 어떤 경우인가?

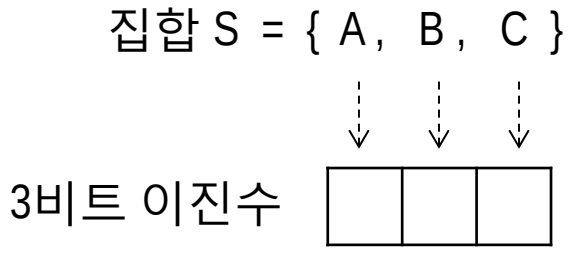
```
public class Test {
        public static void main(String[] args) {
                int     v[]=new int[3];
                bitString(v, 0);
        }
        private static void bitString(int[] v, int i) {
                if(i==v.length){
                        System.out.println(Arrays.toString(v));
                        return;
                }
                v[i]=0;
                bitString(v, i+1);
                v[i]=1;
                bitString(v, i+1);
        }
}
```

3-비트 이진수 생성



모든 부분집합 생성

- 모든 부분집합 생성
  - 3비트 이진수 → 크기 3 집합의 특정 부분집합 표현으로 해석
    - 비트 값 0
      - 집합 내 대응하는 원소를 부분집합의 원소로 미포함
    - 비트 값 1
      - 집합 내 대응하는 원소를 부분집합의 원소로 포함
  - 집합 S = {A, B, C}의 모든 부분집합
    - 000 → {}
    - 001 → {C}
    - 010 → {B}
    - 011 → {B, C}
    - 100 → {A}
    - 101 → {A, C}
    - 110 → {A, B}
    - 111 → {A, B, C}



부분집합의 합 문제

https://en.wikipedia.org/wiki/Subset_sum_problem

- 부분집합의 합 문제 (subset sum problem)
  - 정수 집합 S와 특정한 정수 K에 대해, S의 부분집합의 모든 원소들의 합이 K가 되는 부분집합이 존재하는가
    - S={1,3,4,5,8,9}, K=21
      - {4,8,9}, {3,4,5,9}, {1,3,8,9}, {1,3,4,5,8}

References

- C로 쓴 자료구조론 (Fundamentals of Data Structures in C, Horowitz et al.). 이석호 역. 사이텍미디어. 1993.
- 쉽게 배우는 알고리즘: 관계 중심의 사고법. 문병로. 한빛아카데미. 2013.
- C언어로 쉽게 풀어 쓴 자료구조. 천인국 외 2인. 생능출판사. 2017.
- 프로그래밍 콘테스트 챌린징, Akiba 등 공저, 로드북, 2011.
- C언어로 배우는 알고리즘 입문, 카사이아사오 저, 진명조 역, 한빛미디어, 2004.
- https://introcs.cs.princeton.edu/
- Introduction to Algorithms, Cormen et al., 3rd Edition (The MIT Press)
- https://en.wikipedia.org/wiki/Subset_sum_problem