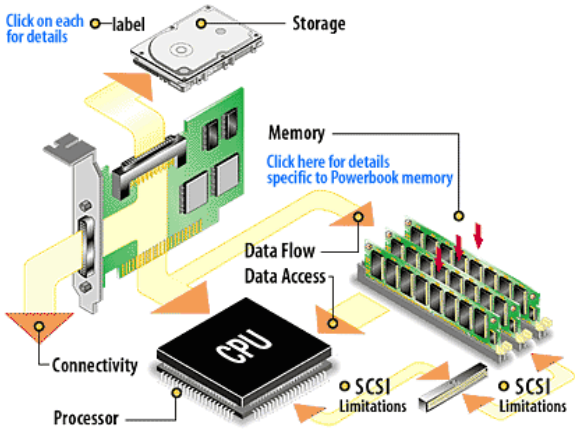
CH#1. 기본개념
(Basic Concepts)

소프트웨어학부

이 의 종 교수

자료구조의 역할 – 왜 중요한가? (1)

- **Computer** : 문제 (계산)를 풀기 위한 장치
- **Algorithm** : 문제를 푸는 방법 (절차)
- **Data Structure** : 문제를 이루는 재료의 구성
- 내가 바라보는 세상을 문제의 세계로 본다면 …
  - 답이 있는 문제
  - 답이 없는 문제
  - 답이 있는지 없는지 모르는 문제



비용의 한계 → 시스템적인 한계

자료구조의 역할 – 왜 중요한가? (2)

- **내가 바라보는 세상을 문제의 세계로 본다면 …**
  - 답이 있는 문제 (Algorithm의 영역)
  - 답이 없는 문제
  - 답이 있는지 없는지 모르는 문제 (Theory of Computation의 영역)

알고리즘은 답이 없는 문제를 다루지 않는다.

알고리즘에서 다루는 문제는 모두 답(해)를 갖고 있다.

알고리즘은 최적의 해(Optimized Solution) 찾는 것이 핵심이다.

알고리즘의 최적의 해는 자료구조의 영향을 받는다.

컴퓨터로 문제를 풀어야 하니, 모든 것은 **비용(Cost)**과 연계된다.

개요: 시스템 생명 주기

시스템 생명 주기(System Life Cycle)(1)

- **요구사항(requirements)**
  - 프로젝트들의 목적을 정의한 명세(specification)들의 집합
  - 입력과 출력에 관한 정보를 기술(description)

- **분석(analysis)**
  - 문제들을 다룰 수 있는 작은 단위들로 나눔
  - 상향식(bottom-up)/하향식(top-down) 접근 방법

- **설계(design)**
  - 추상 데이타 타입(abstract data type) 생성
  - 알고리즘 명세와 설계 기법 고려

시스템 생명 주기(System Life Cycle)(2)

- **정제와 코딩(refinement and coding)**
  - 데이터 객체에 대한 표현 선택
  - 수행되는 연산에 대한 알고리즘 작성

- **검증(verification)**
  - 정확성 증명(correctness proof)
    - 수학적 기법들을 이용해서 증명
    - 어려운 일로 대형 프로젝트에 적용하기 어려움
  - 테스트(testing)
    - 프로그램의 정확한 수행 검증
    - 프로그램의 성능 검사
  - 오류 제거(error removal)
    - 독립적 단위로 테스트 후 전체 시스템으로 통합

포인터와 동적 메모리 할당
(Pointers and Dynamic Memory Allocations)

포인터

- **C언어에서는 어떤 타입 T에 대해 T의 포인터 타입이 존재**
- **포인터 타입의 실제값은 메모리 주소가 됨**

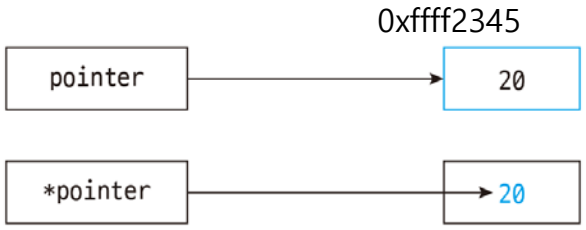
- **포인터 타입에 사용되는 연산자**
  - & : 주소 연산자
  - * : 역참조(dereferencing, 간접 지시) 연산자

    Ex) i(정수 변수), pi(정수에 대한 포인터)

```
int i, *pi
pi = &i;
```

i에 10을 저장하기 위해서는 다음과 같이 할 수 있다.

`i = 10;` 또는 `*pi = 10;`



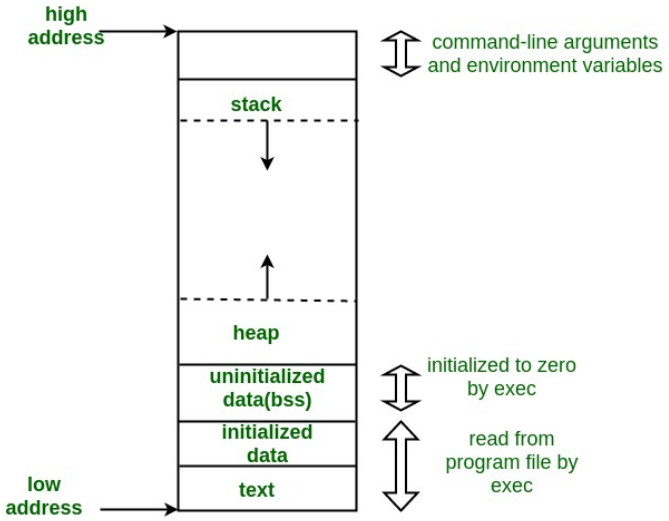
(Source) https://dojang.io/mod/page/view.php?id=605

- **널(null)포인터 : 어떤 객체나 함수도 가리키지 않는다.**
  - 정수 0값으로 표현
  - 널포인터에 대한 검사

    `int (pi == NULL)` 또는 `if(!pi)`

동적 메모리 할당(1)

- **동적 메모리 할당**
  - 프로그램을 작성할 때 얼마나 많은 공간이 필요한지 알 수 없을 때 사용
  - 히프(heap) 사용
  - 새로운 메모리 공간이 필요할 때마다 함수 malloc을 호출해서 필요한 양의공간을 요구




```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

메모리 할당과 반환

포인터의 위험성

- **포인터가 대상을 가리키고 있지 않을 때는 값을 전부 null로**
  - 프로그램 범위 밖이나 합당하지 않은 메모리 영역을 참조할 가능성↓

- **명시적인 타입 변환(type cast): `void *malloc(size_t size);`**

권장되지 않음

malloc-typecasting.c

```
pi = malloc(sizeof(int));    /* pi에 정수에 대한 포인터를 저장 */
pf = (float *) pi;           /* 정수에 대한 포인터를 부동소수에
                                대한 포인터로 변환 */
```

Dangling reference (허상참조)

```
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
pf = (float *) malloc(sizeof(float));   ← 3.14가 저장된 공간에 대한 접근방법??
```

알고리즘 명세
(Algorithm Specification)

개요

- **특정한 일을 수행하기 위한 명령의 유한 집합**
- **조건**
  - 입력 : (외부) 원인 ≥ 0
  - 출력 : 결과 ≥ 1
  - 명백성(definiteness) : 모호하지 않은 명확한 명령
  - 유한성(finiteness) : 종료
  - 유효성(effectiveness) : 기본적, 실행가능 명령

    Ex) program ≡ algorithm (항상 종료된다는 관점에서)

    흐름도(flowchart) ≡ algorithm

    (알고리즘이 작고 단순한 경우)

예제: 선택정렬(Selection Sort) (1)

- **n ≥ 1개의 서로 다른 정수를 정렬**
  - 정렬되지 않은 정수들 중에서 가장 작은 값을 찾고,
  - 정렬된 리스트 다음 자리에 놓음
    - 정수들이 배열(array), `list`에 저장
    - `i`번째 정수는 `list[i]`, 1≤ i ≤ n에 저장

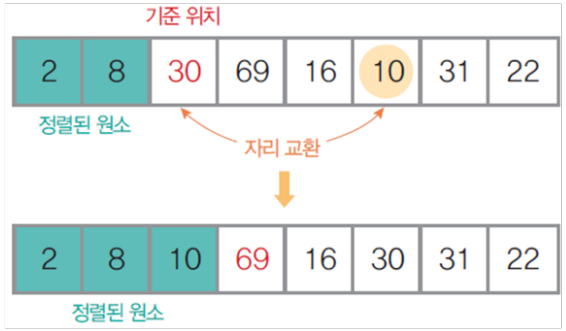
```
for ( i = 0; i < n; i++) {
   list[i]에서부터 list[n-1]까지의 정수 값을 검사한 결과
   list[min]이 가장 작은 정수 값이라 하자;

   list[i]와 list[min]을 서로 교환;
}
```

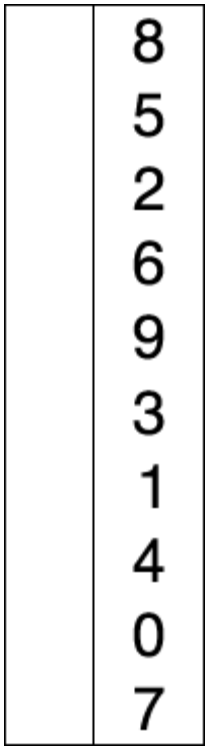
< 선택 정렬 알고리즘 >

  - 정렬되지 않은 정수들 중에서 가장 작은 값을 찾아서 정렬된 리스트 다음 자리에 놓음

예제: 선택정렬(Selection Sort) (2)

- {69, 10, 30, 2, 16, 8, 31, 22}





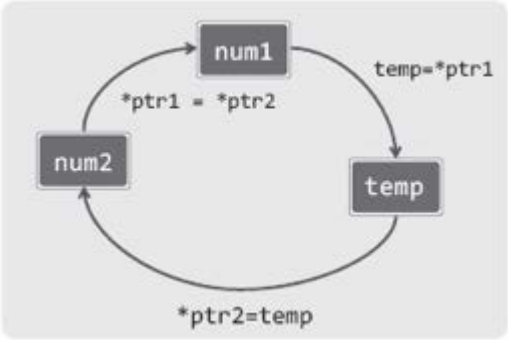
https://en.wikipedia.org/wiki/Selection_sort#/media/File:Selection-Sort-Animation.gif

예제: 선택정렬(Selection Sort) (3)

- **최소 정수 값 `list[min]`을 `list[i]` 값과 교환하는 작업**
  - 함수 정의 : `swap(&a, &b)`

```
void swap(int *x, int *y)
/* 매개 변수 x, y는 정수형을 갖는 포인터 변수이다. */
{
     int temp = *x;    /* temp변수를 int로 선언, x가
                          가르키는 주소의 내용을 지정한다. */
     *x = *y           /* y가 가리키는 주소의 내용을
                          x가 가리키는 주소에 저장한다. */
     *y = temp         /* temp의 내용을 y가 가리키는
                          주소에 저장 */
```

< swap 함수 >



  - 매크로 정의

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

예제: 선택정렬(Selection Sort) (4)

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int);          /* selection sort */
void main(void)
{
    int i, n;
    int list[MAX_SIZE];

    printf("Enter the number of numbers to generate: ");
    scanf("%d", &n);

    if(n<1 || n> MAX_SIZE) {
         fprintf(stderr, "Improper value of n\n");
         exit(EXIT_FAILURE);
    }
    for (i = 0; i < n; i++) {   /* randomly generate numbers*/
         list[i] = rand()%1000;
         printf("%d", list[i]);
    }
```

예제: 선택정렬(Selection Sort) (5)

```
      sort(list, n);
      printf("\n Sorted array:\n");
      for(i=0; i<n; i++)    /*print out sorted numbers */
                printf("%d", list[i]);
      printf("\n");
}

void sort(int list[], int n)
{
      int i, j, min, temp;
      for(i = 0; i < n-1; i++) {
          min = i;
          for (j = i+1; j < n; j++)
            if(list[j] <list[min])
                  min =j;
          SWAP(list[i], list[min], temp);
      }
}
```

list[min] = 정렬되지 않은 값들 중 가장 작은 값

list[i] = 다음 번 최소값이 들어갈 위치

예제: 선택정렬(Selection Sort) (6)

- **정리 1.1**
  - 함수 sort(list, n)는 n≥1개의 정수를 정확하게 정렬한다.
  - 그 결과는 list[0], … , list[n-1]로 되고
  - 여기서 list[0]≤list[1] ≤ list[n-1]이다.

- **증명**
  - $i = q$에 대해, 외부 for 루프가 완료되면

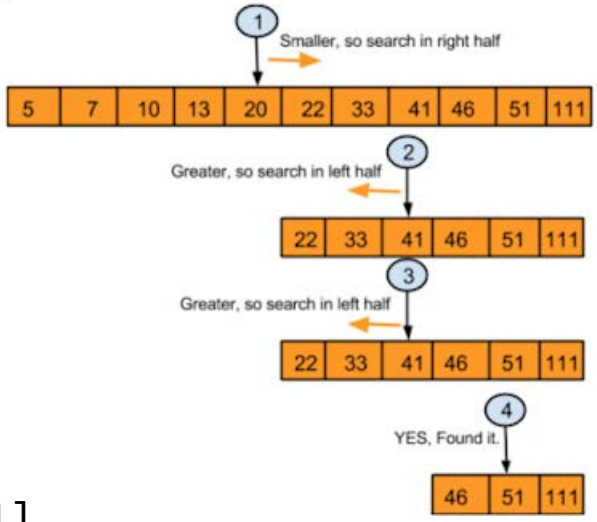
    list[q] ≤ list[r], q<r<n 이다.
  - 다음 반복에서는 $i>q$ 이고 list[0]에서 list[q]까지는 변하지 않는다.
  - 따라서 바깥 for 루프를 마지막으로 수행하면(즉, i=n-2),

    list[0] ≤ list[1]≤... ≤ list[n-1]가 된다.

예제: 이진탐색(Binary Search) (1)

- 정수 searchnum이 배열 list에 있는지 검사
  - list[0] ≤ list[1] ≤ ... ≤ list[n-1]
  - 주어진 정수 searchnum에 대해 list[i]=searchnum인 인덱스 i 반환
  - 없는 경우는 -1 반환
  - 초기값 : left=0, right=n-1

  - list의 중간 위치 : middle=(left+right)/2
  - list[middle]와 searchnum을 비교
    1) searchnum < list[middle] :
       list[0] ≤ searchnum ≤ list[middle-1]
       right = middle-1
    2) searchnum = list[middle] : middle을 반환
    3) searchnum > list[middle] :
       list[middle+1] ≤ searchnum ≤ list[n-1]
       left = middle+1



예제: 이진탐색(Binary Search) (2)

```
while (there are more integers to check) {
        middle = (left + right) / 2;
        if(searchnum < list[middle])
                right = middle-1;
        else if (searchnum == list[middle])
                return middle;
        else left = middle +1;
}
```

< 정렬된 리스트 탐색 >

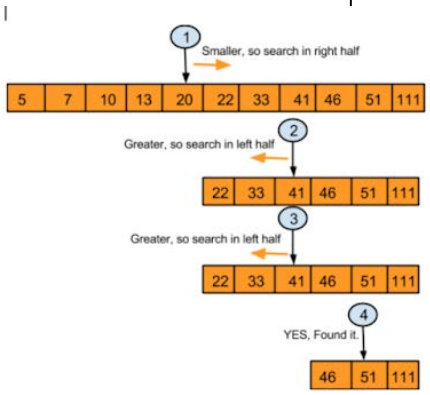
```
int compare(int x, int y)
{   /* compare x and y, return -1 for less than,
       0 for equal,  1 for greater */
      if (x < y) return -1;
      else if (x == y) return 0;
             else return 1;
}
```

< 두 정수의 비교 >

예제: 이진탐색(Binary Search) (3)

- **비교 연산 : 함수, 매크로**
  - `#define COMPARE(x, y) (((x) < (y)) ? -1 : ((x) == (y)) ? 0 : 1)`
  - 첫 번째 수치 값이 두 번째 수치의 값보다 작을 때 음수 반환
  - 두 수치 값이 같은 경우 0을 반환
  - 첫 번째 수치 값이 두 번째 수치 값보다 큰 경우 양수 반환

```
int binsearch(int list[], int searchnum, int left, int right)
{  */ search list [0] <= list[1] <= … <= list[n-1] for searchnum.
      Return its position if found. Otherwise return -1 */
     int middle;
     while (left <=right) {
         middle = (left + right) / 2;
         switch (COMPARE(list[middle], searchnum)) {
             case -1: left = middle + 1
                      break;
             case 0 : return middle;
             case 1 : right = middle -1;
         }
      }
      return -1;
}
```



순환 알고리즘(Recursive Algorithms)(1)

Recursive Call
= 재귀호출
= 순환호출

- **수행이 완료되기 전에 <u>자기 자신을 다시 호출</u>**
  - 직접 순환 : direct recursion (func_a → func_a)
  - 간접 순환 : indirect recursion (func_a → func_b → func_a)

- **Fibonacci 수, factorial, 이항 계수**

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

나 자신(함수)을 호출

*n*개의 원소 중 *m*개의 원소를
뽑아내는 경우의 수

=

*n-1*개의 원소 중 *m*개를
뽑아내는 경우의 수

+

*n-1*개의 원소 중 *m-1*개를
뽑아내는 경우의 수 의 합

순환 알고리즘(Recursive Algorithms)(2)

- **예제 1.3: 이진 탐색**

Direct recursive call

```
int binsearch(int list[], int searchnum, int left, int right)
{  */ search list [0] <= list[1] <= … <= list[n-1] for searchnum.
        Return its position if found. Otherwise return -1 */

     int middle;
     if (left <=right) {
            middle = (left + right) / 2;
            switch (COMPARE(list[middle], searchnum)) {
                case -1: return
                        binsearch(list, searchnum, middle+1, right);
                case 0 : return middle;
                case 1 : return
                        binsearch(list, searchnum, left, middle-1);
            }
      }
      return -1;
}
```

< 이원탐색에 대한 순환 구현 >

순환 알고리즘(Recursive Algorithms)(3)

- **예제 1.4: 순열(permutation)**

  n≥1개의 원소를 가진 집합 모든 가능한 순열

  – {a, b, c} : {(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)}

  – n원소 : $n!$개의 상이한 순열

  – {a, b, c, d}

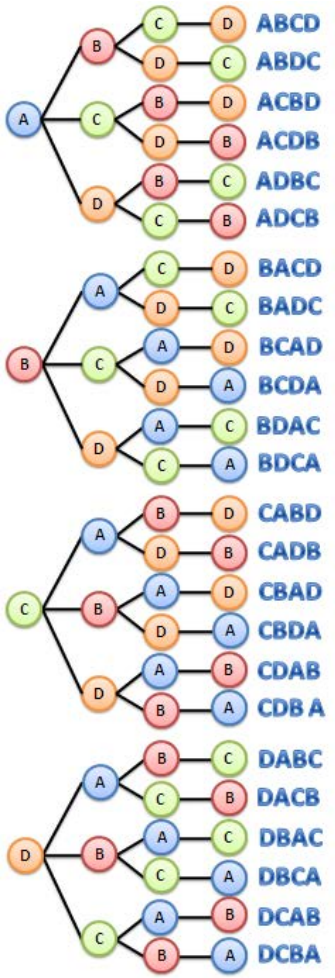
  1) a로 시작하는 {b,c,d}의 모든 순열

  2) b로 시작하는 {a,c,d}의 모든 순열

  3) c로 시작하는 {a,b,d}의 모든 순열

  4) d로 시작하는 {a,b,c}의 모든 순열

$$n! = n \times (n-1)! = n \times (n-1) \times (n-2)! = \cdots$$



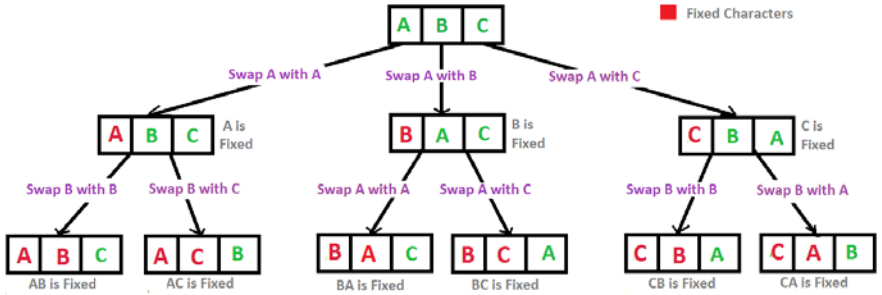
순환 알고리즘(Recursive Algorithms)(4)

- 초기 함수 호출 : perm(list, 0, n-1)

```
void perm(char *list, int i, int n)
{*/ generate all the permutations of list[i] to list[n] */
      int j, temp;
      if(i == n) {
         for(j = 0; j <= n; j++)
             printf("%c", list[j]);
         printf("        ");
      }
      else {
         /* list[i] to list[n] has more than
            one permutation, generate these recursively*/
         for(j = i, j <= n; j++){
             SWAP(list[i], list[j], temp);
             perm(list, i+1, n);
             SWAP(list[i], list[j], temp);
         }
      }
}
```

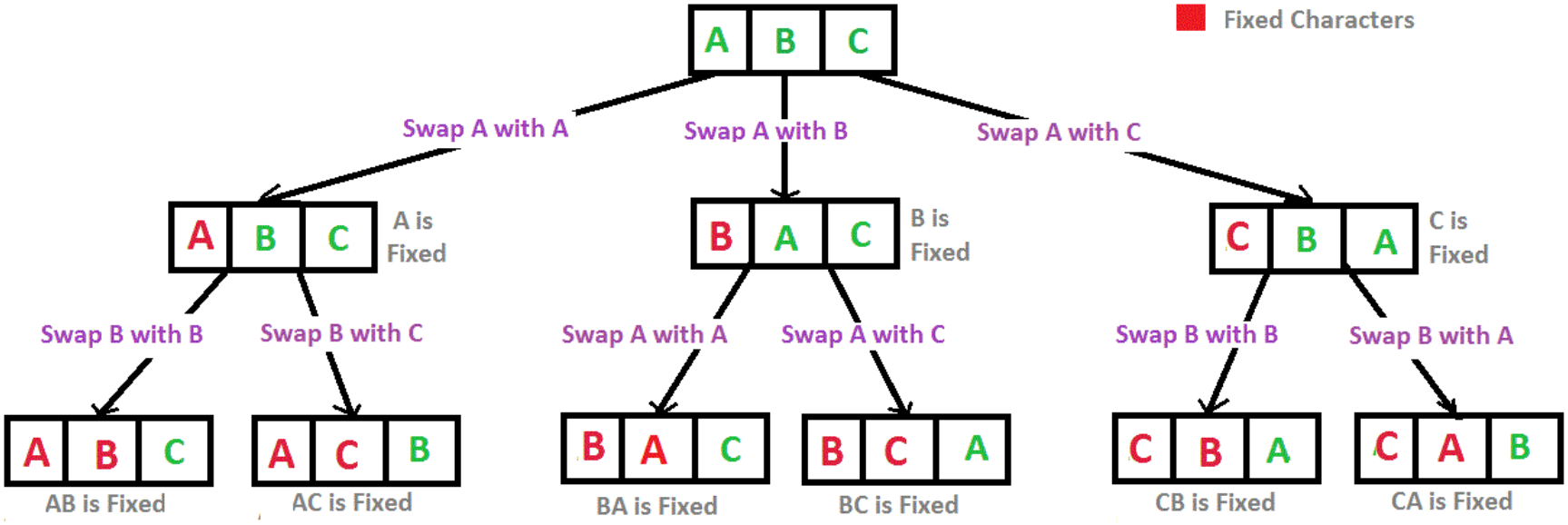
< 순환 순열 생성기 >

http://www.eandbsoftware.org/print-all-permutations-of-a-given-string/



Recursion Tree for Permutations of String "ABC"

순환 알고리즘(Recursive Algorithms)(5)



Recursion Tree for Permutations of String "ABC"

http://www.eandbsoftware.org/print-all-permutations-of-a-given-string/

데이터 추상화
(Data Abstraction)

데이터 추상화(Data Abstraction)(1)

- **C언어의 기본 자료형 : char, int, float, double**
  - 키워드 (short, long, unsigned)에 의해 변경될 수 있음

- **자료의 그룹화하는 2가지 방법**
  - 배열(array)
    - int list[5] : 정수형 배열, 동일한 기본 데이터 타입에 속하는 원소의 집합
  - 구조(structure)

```
struct student {
        char lastName;
        int  studentId;
        char grade;
}
```

```
typedef struct {
        char lastName;
        int  studentId;
        char grade;
} student;
```

- **포인터 자료형 : 정수형, 실수형, 문자형, float형 포인터**
  - int i, *pi;

Predefined data type

- **사용자 정의(user-defined) 자료형**

데이터 타입(Data Type)

- **정의 : 데이터타입(data type) =**

  객체(object) + 객체 위에 동작 하는 연산(operation)들의 집합

  - `int` : {0, +1, -1, +2, -2, … , INT_MAX, INT_MIN}
  - 정수 연산 : +, -, *, /, %, 테스트 연산자, 치환문 …
    - atoi와 같은 전위(prefix) 연산자 → atoi("283")
    - +와 같은 중위(infix) 연산자 → a + b
  - 연산(operation):
    - 이름(ex: 함수명), 매개 변수, 결과에 대한 정의가 명확해야 한다 (함수를 생각)

- **자료형의 객체 표현**
  - char형: 1바이트 비트 열
  - int형: 2 또는 4바이트
  - 구체적인 내용을 사용자가 모르도록 하는 것이 좋은 방법
    - 객체 구현 내용에 대한 사용자 프로그램의 독립성

제공된 함수를 통해서만 객체를 처리

추상 자료형

Programming Level ↔ Abstract Level

- **정의 : 추상 자료형(ADT: abstract data type)**
  - 객체의 명세(specification)와 그 연산의 명세(specification)가 **객체의 표현과 연산의 구현으로부터 분리된 자료형**

- **ADT 연산의 명세**
  - 함수 이름, 매개 변수형, 결과형, 함수가 수행하는 기능에 대한 기술
  - 내부적 표현이나 구현에 대한 자세한 설명은 필요 없음
    - ADT는 구현에 독립

데이터 타입 함수

- **생성자(creator)/구성자(constructor) :**
  - 새 인스턴스 생성
- **변환자(transformer) :**
  - 한 개 이상의 서로 다른 인스턴스를 이용하여 지정된 형의 인스턴스를 만듦
  - `Add(x, y), Subtract(x, y)`
- **관찰자(observers)/보고자(reporter) :**
  - 자료형의 인스턴스에 대한 정보를 제공

예제 1.5: 추상 데이터 타입 NaturalNumber

- **객체(objects)** :
  - 0 ~ 컴퓨터상의 최대 정수 값(INT_MAX)까지 순서화된 정수의 부분 범위

- **함수(functions)** :
  - x, y ∈ NaturalNumber
  - TRUE, FALSE ∈ Boolean
  - +, -, <, ==는 일반적인 정수 연산자

```
functions:
    NaturalNumber Zero()          ::= 0
    Boolean IsZero(x)             ::= if(x) return FALSE
                                      else return TRUE
    Boolean Equal(x, y)           ::= if(x==y) return TRUE
                                      else return FALSE
    NaturalNumber Successor(x)    ::= if(x == INT_MAX) return x
                                      else return x+1
    NaturalNumber Add(x, y)       ::= if((x+y) <= INT_MAX) return x+y
                                      else return INT_MAX
    NaturalNumber Subtract(x, y) ::= if(x < y) return 0
                                      else return x-y
end NaturalNumber
```

성능분석
(Performance Measurement)

성능 분석 (Performance Measurement)

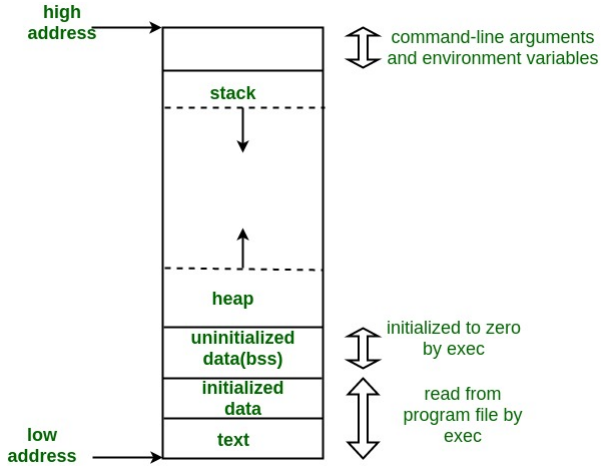
- **성능 평가(performance evaluation)**
  - 성능분석(performance analysis)
    - 시간과 공간의 추산 (예측)
    - 복잡도 이론(complexity theory)
  - 성능측정(performance measurement)
    - 컴퓨터 의존적 실행 시간, 실측성능

- **정의(definition of complexity)**
  - 공간 복잡도(space complexity)
    - 프로그램을 실행시켜 완료하는데 필요한 공간의 양 (ex: Memory)
  - 시간 복잡도(time complexity)
    - 프로그램이 완료되는데 필요한 컴퓨터 시간의 양 (ex: CPU time)

공간 복잡도(Space Complexity)(1)

- **프로그램에 필요한 공간**
  - 고정 공간 요구 — Compile Time에 정해지는 공간
    - 프로그램 입출력의 횟수나 크기와 관계없는 공간 요구
    - 명령어 공간, 단순 변수, 고정 크기의 구조화 변수, 상수
  - 가변 공간 요구
    - 문제의 인스턴스 I 에 의존하는 공간
    - 가변 공간

malloc()



- **공간 요구량 : $S(P) = c + S_p(I)$**

- **예제 1.6: abc**
  - 고정 공간 요구만을 가짐 → $S_{abc}(I) = 0$ — 변수들은 stack에 위치

```
float abc(float a, float b, float c)
{
        return a+b+b*c + (a+b-c) / (a+b) + 4.00;
}
```

공간 복잡도(Space Complexity)(2)

```
float sum(float list[], int n)
{
        float tempsum = 0;
        int i;
        for (i=0; i<n; i++)
                tempsum += list[i];
        return tempsum;
}
```

- **예제 1.7: sum**
  - 가변 공간 요구: 배열(크기 n)
    - 함수에 대한 배열의 전달 방식
    - Pascal : 값 호출(**call by value**)
      - $S_{sum}(I) = S_{sum}(n) = n$: 배열 전체가 임시기억장소에 복사
    - C : 배열의 첫 번째 요소의 주소 전달 (**call by reference**)
      - $S_{sum}(n) = 0$

공간 복잡도(Space Complexity)(3)

- **예제 1.8: rsum**
  - 함수가 실행될 때마다 매개 변수, 지역 변수, 매 순환 호출 시에 복귀 주소를 저장

```
float rsum(float list[], int n)                    Recursive call
{
        if (n) return rsum(list, n-1) + list[n-1];
        return 0;
}
```

  - 하나의 순환 호출을 위해 요구되는 공간
    - 두 개의 매개 변수, 복귀 주소를 위한 바이트 수 정수와 포인터가 각각 4 Bytes로 가정

      = sizeof(n) + sizeof(list[]의 주소) + sizeof(복귀주소) = 12
    - N = MAX_SIZE

      → $S_{rsum}(MAX\_SIZE) = 12 * MAX\_SIZE$

MAX_SIZE=1000
→ 12x1000 = 12,000Bytes

순환함수는 반복함수보다 큰 오버헤드가 필요
실무에서 recursive function 사용??

시간 복잡도(Time Complexity)(1)

- **프로그램 P에 의해 소요되는 시간 : T(P)**
  = 컴파일 시간 + T_p(= 실행 시간)

수치 값을 더하고 빼는 간단한 프로그램의 소요시간 $T_p(n)$

$$T_p(n) = C_aADD(n) + C_sSUB(n) + C_lLDA(n) + C_{st}STA(n)$$

- C_a, C_s, C_l, C_{st} : 각 연산을 수행하기 위해 필요한 상수 시간
- $ADD, SUB, LDA, STA(n)$: 특성 n에 대한 연산 실행 횟수

실측 성능 vs. 컴퓨터에 독립적 견적

- **정의 : 프로그램 단계(program step)**
  - 의미적으로 독립성을 갖는 프로그램의 단위 → 1 step
    - `a = 2`
    - `a = 2*b+3*c/d-e+f/g/a/b/c`
  - 한 단계 실행에 필요한 시간이 인스턴스 특성에 독립적이어야 함

시간 복잡도(Time Complexity)(2)

- **단계의 계산(방법 1)**
  - 전역 변수 count의 사용

- **예제 1.9: 리스트에 있는 수의 반복적 합산**

```
float sum(float list[], int n)
{
    float tempsum = 0; count++;             /*for assignment */
    int i;
    for(i=0; i<n; i++) {
        count++;                            /*for the for loop */
        tempsum += list[i]; count++;        /*for assignment */
    }
    count++;    /* last execution of for */
    count++;    /* for return */
    return tempsum;
}
```

```
float sum(float list[], int n)              /* 단순화된 프로그램 */
{
    float tempsum = 0;
    int i;
    for(i=0; i<n; i++)
        count += 2;
    count += 3;
    return 0;
}
```

2n+3 단계 수행

시간 복잡도(Time Complexity)(3)

- **예제 1.10: 리스트에 있는 수의 순환적 합산**

```
float rsum(float list[], int n)
{
    count++;       /*for if conditional */
    if (n)    {
       count++;    /*for return and rsum invocation */
       return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}
```

  - n = 0 → 2 (if, 마지막 return)
  - n > 0 → 2 (if, 처음 return) : n회 호출

  ∴ *2n + 2 steps*

※ *2n + 3 (iterative) > 2n + 2 (recursive)*

→ $T_{iterative} > T_{recursive}$

얼마나 많은 단계가 수행되는지 말해줄 뿐, 얼마나 많은 시간을 소요하는지 말해주는 개념이 아님

순환 함수가 적은 단계 수를 갖더라도 반복 함수보다 실행 속도가 느리다. Why?

시간 복잡도(Time Complexity)(4)

- 예제 1.11: 행렬의 덧셈 C = A+B

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                            int c[][MAX_SIZE}, int rows, int cols)
{
    int i, j;
    for(i=0; i<rows; i++)
       for(j=0; j<cols; j++)
          c[i][j] = a[i][j] + b[i][j];
}
```

< count문이 첨가된 행렬의 덧셈 >

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                            int c[][MAX_SIZE}, int rows, int cols)
{
    int i, j;
    for(i=0; i<rows; i++)  {
       count++;                          /* for i for loop */
       for(j=0; j<cols; j++)   {
          count++;                       /* for j for loop */
          c[i][j] = a[i][j] + b[i][j];
          count++; }                     /* for assignment statement */
       count++;                          /* last time of j for loop */
    }
    count++;     /*last time of i for loop */
}
```

시간 복잡도(Time Complexity)(5)

단순화: 단계수만 계산

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                            int c[][MAX_SIZE}, int rows, int cols)
{
        int i, j;
        for(i=0; i<rows; i++)
                for(j=0; j<cols; j++)   {
                        count += 2;
                count+=2;
         }
         count++;
}
```

배열 크기 = rows × cols

단계 수 = 2rows · cols + 2rows + 1

rows >> cols → 행과 열을 교환

시간 복잡도(Time Complexity)(6)

- **단계의 계산(방법 2)**
  - 테이블 방식(tabular method) : 단계 수 테이블
    - 문장에 대한 단계 수 : steps/execution = s/e
    - 문장이 수행되는 횟수 : 빈도수(frequency)
      - 비실행 문장의 빈도수 = 0
    - 총 단계 수 : 빈도수 × s/e

- **예제 1.12: 리스트에 있는 수를 합산하는 반복함수**

문 장	*s/e*	빈도수	총단계 수
`float sum(float list[], int n)`	0	0	0
`{`	0	0	0
`  float tempsum = 0;`	1	1	1
`  int i;`	0	0	0
`  for (i=0; i<n; i++)`	1	n+1	n+1
`  tempsum += list[i];`	1	n	n
`  return tempsum;`	1	1	1
`}`	0	0	0
합계	*2n+3*		

시간 복잡도(Time Complexity)(7)

- **예제 1.13: 리스트에 있는 수를 합산하는 순환 함수**

문 장	s/e	빈도수	총단계 수
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1) + list[n-1]	1	n	n
return list[0];	1	1	1
}	0	0	0
합 계			*2n+2*

- **예제 1.14: 행렬의 덧셈**

문 장	s/e	빈도수	총단계 수
void add(int a[][MAX_SIZE] ···)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i=0, i<rows; i++)	1	rows+1	rows+1
for(j=0, j<cols; j++)	1	rows(cols+1)	rows·cols+row
c[i][j] = a[i][j] + b[i][j]	1	rows·cols	rows·cols
}	0	0	0
합 계			*2rows·cols+2rows+1*

요약

- **프로그램의 시간복잡**
  - 프로그램의 기능을 수행하기 위한 프로그램이 취한 단계수로 표현
  - 일반적으로 중요한 특성만 선택
  - 입력수의 함수로만 계산 (ex: n)

- **단계는 독립적인 연산단위**
  - 10개의 덧셈이 한 단계로, 100개의 덧셈이 한 단계가 될 수 있다.
  - n개의 덧셈은 한 단계로 될 수 없음 (m/2, p+q개의 뺄셈도 동일)
    - 상수(constant) ↔ 변수(variable)

- **이진탐색의 경우 리스트 원소의 수 n으로 단계 수 결정은 부적절**
  - 동일한 n에 대해 searchnum의 위치에 따라 단계 수 결정

- **Best Case(Ω), Worst Case(O), Average Case(Θ)로 구분하여 단계 수를 정의할 필요가 있음**
  - 최상 단계(Best Case): 주어진 매개변수에 대해 단계수가 최소
  - 최소 단계(Worst Case): 주어지 매개변수에 대해 단계수가 최대
  - 평균 단계(Average Case): 주어진 매개변수에 대해 실행되는 평균 단계

점근 표기법(Asymptotic Notation : O,Ω,Θ)(1)

- **Worst case step count : 최대 수행 단계 수**
- **Average step count : 평균 수행 단계 수**

Asymptotic notation
→ 근사치로 표현

- **단계수를 결정하는 이유:**
  - 동일 기능의 두 프로그램의 시간 복잡도 비교
  - 인스턴스 특성의 변화에 따른 실행 시간 증가 예측

- **정확한 단계의 계산 : 무의미**

대략적인 값으로도 예측 가능

  - A의 단계 수 = 3n + 3, B의 단계 수 = 100n +10 → $T_A << T_B$
  - A ≈ 3n(or 2n), B ≈100n(or 80n, 85n) → $T_A << T_B$

점근 표기법(Asymptotic Notation : O,Ω,Θ)(2)

- **근사치 사용**
  - c_1과 c_2가 음이 아닌 상수일 때, 프로그램 P, Q의 시간복잡도

    $c_1n^2 \leq T_P(n) \leq c_2n^2$ 또는 $T_Q(n,m) = c_1n + c_2m$ 로 표현 가능

  - **균형분기점**(break even point) : $n = 98$
    - 예) $c_1=1$, $c_2=2$, $c_3=100$

      $c_1n^2 + c_2n \leq c_3n$, $n \leq 98$

      $c_1n^2+c_2n > c_3n$, $n > 98$

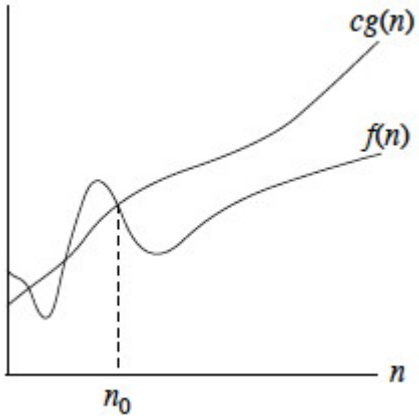
손익분기점 Break even point =
Asymptotic Notation : O, Ω, Θ의 n_0에 해당

점근 표기법(Asymptotic Notation : O,Ω,Θ)(3)

- **정의 [Big "oh"] (O)**
  - 모든 n, $n \geq n_0$에 대해 $f(n) \leq cg(n)$인 조건을 만족하는 두 양의 상수 c와 n_0가 존재하기만 하면 $f(n)=O(g(n))$이다.

n_0 = Break even point

- **예제**
  - n≥2, 3n+2≤4n → 3n+2=O(n)
  - n≥3, 3n+3≤4n → 3n+3=O(n)
  - n≥10, 100n+6≤101n → ?
  - n≥5, $10n^2+4n+2 \leq 11n^2$ → ?
  - n≥4, $6*2^n+n^2 \leq 7*2^n$ → ?
  - n≥2, $3n+3 \leq 3n^2$ → ?
  - n≥2, $10n^2+4n+2 \leq 10n^4$ → ?
  - $n \geq n_0$인 모든 n과 임의의 상수 c에 대해 3n+2≤c가 false인 경우가 존재하면 $3n+2 \neq O(1)$. $10n^2+4n+2 \neq O(n)$



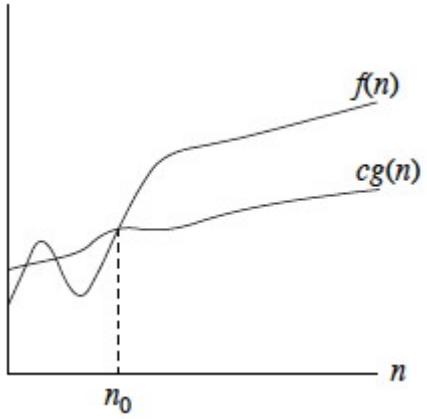
※ $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

※ $f(n) = O(g(n))$
- $n \geq n_0$인 모든 n에 대해 g(n) 값은 f(n)의 상한 값
- g(n)은 조건을 만족하는 가장 작은 함수여야 함

점근 표기법(Asymptotic Notation : O,Ω,Θ)(4)

- **정의 [Omega][f(n) = Ω(g(n))]**
  - 모든 n, $n \geq n_0$에 대해서 $f(n) \geq cg(n)$을 만족하는 두 양의 상수 c와 n_0가 존재하기만 하면 $f(n) = \Omega(g(n))$이다.

- **예제**
  - $n \geq 1, 3n+2 \geq 3n$ → $3n+2 = \Omega(n)$
  - $n \geq 1, 3n+3 \geq 3n$ → $3n+3 = \Omega(n)$
  - $n \geq 1, 100n+6 \geq 100n$ → $100n+6 = \Omega(n)$
  - $n \geq 1, 10n^2+4n+2 \geq n^2$ → $100n^2+4n+2 = \Omega(n^2)$
  - $n \geq 1, 6*2^n+n^2 \geq 2^n$ → $6*2^n+n^2 = \Omega(2^n)$

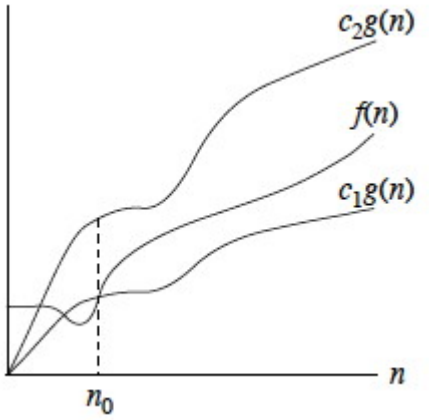


※g(n) : f(n)의 하한 값(가능한 큰 함수)

점근 표기법(Asymptotic Notation : O,Ω,Θ)(5)

- **정의[Theta][f(n) = Θ(g(n))]**
  - 모든 n, $n \geq n_0$에 대해서 $C_1 g(n) \leq f(n) \leq C_2 g(n)$을 만족하는 세 양의 상수 C_1, C_2와 n_0가 존재하기만 하면 $f(n)=\Theta(g(n))$이다.

- **예제**
  - $n \geq 2$, $3n \leq 3n+2 \leq 4n$ → $3n+2 = \Theta(n)$
    $c_1=3$, $c_2=4$, $n_0=2$
  - $3n+3 = \Theta(n)$
  - $10n^2+4n+2 = \Theta(n^2)$
  - $6*2^n+n^2 = \Theta(2^n)$
  - $10*\log n + 4 = \Theta(\log n)$



※g(n)이 f(n)에 대해 상한 값과 하한 값을 모두 가지는 경우
※g(n)의 계수는 모두 1

Θ 표기법은 빅오나 오메가 표기법보다 더 정확 → 하지만 빅오를 더 많이 사용

점근 표기법(Asymptotic Notation : O,Ω,Θ)(6)

- **예제**
  - $T_{sum} = 2n + 3 \rightarrow T_{sum}(n) = \Theta(n)$
  - $T_{rsum}(n) = 2n + 2 \rightarrow \Theta(n)$
  - $T_{add}(rows, cols) = 2row \cdot cols + 2rows + 1 = \Theta(rows \cdot cols)$

- **점근적 복잡도(asymptotic complexity: O, Ω, Θ)는**
  정확한 단계수의 계산없이 쉽게 구함

- **예제[행렬 덧셈의 복잡도]**

점근적 복잡도 = 근사치

문 장	점근적 복잡도

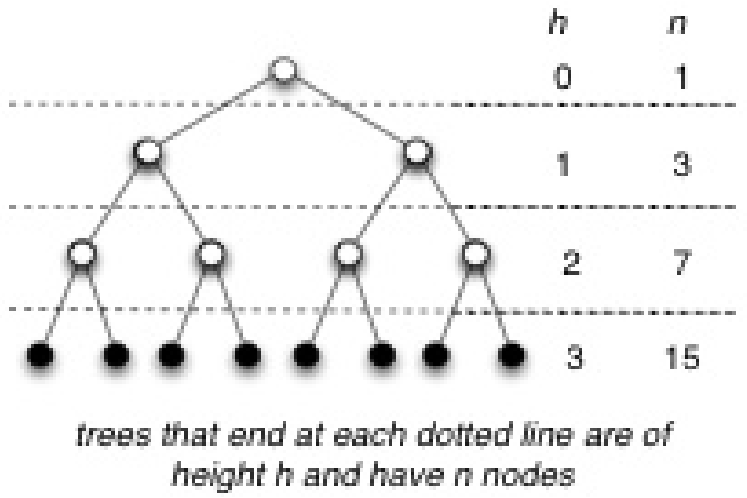
`void add(int a[][MAX_SIZE] ··· )`
`{`
`  int i, j;`
`  for (i=0, i<rows; i++)`
`  for(j=0, j<cols; j++)`
`  c[i][j] = a[i][j] + b[i][j]`
`}`
	0
	0
	0
	$\Theta(rows)$
	$\Theta(rows \cdot cols)$
	$\Theta(rows \cdot cols)$
	0

합 계	$\Theta(rows \cdot cols)$

점근 표기법(Asymptotic Notation : O,Ω,Θ)(7)

- **[예제] 이원탐색 (프로그램 1.7)**
  - 인스턴스 특성
    - 리스트 원소의 수 n
  - while 루프에서 반복시간
    - $\log_2(n+1)$번
  - list 세그먼트의 크기 → 1/2 감소
    - 최악의 경우 Θ(log n)번 반복
  - 한번 반복 실행하는데 $\Theta(1)$
    - 최악의 경우 복잡도 = Θ(log n)

```
int binsearch(int list[], int searchnum, int left, int right)
{  */ search list [0] <= list[1] <= … <= list[n-1] for searchnum.
      Return its position if found. Otherwise return -1 */
     int middle;
     while (left <=right) {
            middle = (left + right) / 2;
            switch (COMPARE(list[middle], searchnum)) {
                     case -1: left = middle + 1
                                   break;
                     case 0 : return middle;
                     case 1 : right = middle -1;}
      }
      return -1;
}
```



trees that end at each dotted line are of height h and have n nodes

Tree의 높이 = $\log_2 n$

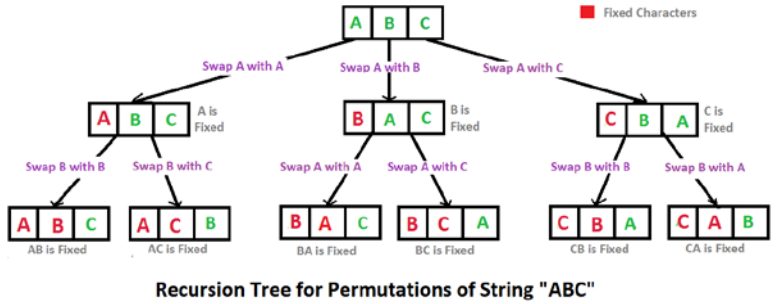
※ while 루프의 첫 번째 반복 실행에서 searchnum을 찾는 경우 → 최상의 경우 → 복잡도 Θ(1)

점근 표기법(Asymptotic Notation : O,Ω,Θ)(8)

수행단계 = $T_{perm}(i, n)$

- **[예제] 순열 (프로그램 1.9)**
  - `i = n`일 때 → $\Theta(n)$
  - `i < n`일 때 else절 수행
    - `for`루프는 `n-i+1`번 수행
    - 루프를 한번 실행할 때 마다
    - $\Theta(T_{perm}(i+1, n))$
  - 따라서, `i < n`일 때 (for loop)
    - $T_{perm}(i, n) = \Theta((n-i+1)(T_{perm}(i+1,n)))$
  - $T_{perm}(n, n) = n$
  - 이 순환을 풀면 n≥1에 대해 $T_{perm}(i, n) = \Theta(n(n!))$라는 결론

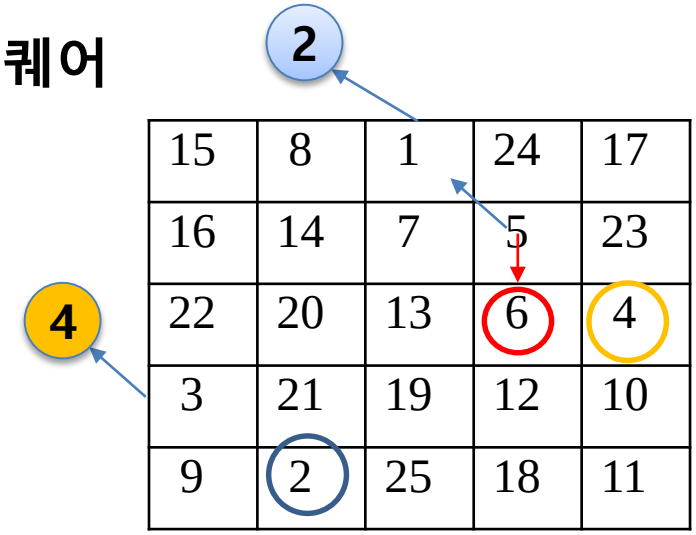
```
void perm(char *list, int i, int n)
{*/ generate all the permutations of list[i] to list[n] */
      int j, temp;
      if(i == n) {
            for(j=0; j<=n; j++)
                  printf("%c", list[j]);
            printf("        ");
      }
      else {
         /* list[i] to list[n] has more than one permutation,
            generate these recursively*/
            for(j=i, j<=n; j++){
                  SWAP(list[i], list[j], temp);
                  perm(list, i+1, n);
                  SWAP(list[i], list[j], temp);
            }
       }
}
```



Recursion Tree for Permutations of String "ABC"

점근 표기법(Asymptotic Notation : O,Ω,Θ)(9)

- **[예제] 매직 스퀘어**



< n=5인 매직 스퀘어 >

- **Algorithm [Coexter] : n이 홀수 일 때**
  - 첫 번째 행의 중앙에 1을 넣는다.
  - 왼쪽 대각선 방향으로 올라가면서 빈 자리에 1씩 큰 수를 넣는다.
  - 밖으로 벗어나면, 반대편 자리에서 계속한다.
    - 상단을 벗어나면, 같은 열 최 하단으로
    - 왼쪽에서 벗어나면 같은 행의 제일 오른쪽으로 이동
  - 만약 이동하려는 자리에 숫자가 있으면 바로 밑으로 가서 계속

점근 표기법(Asymptotic Notation : O,Ω,Θ)(10)

- [예제] 매직 스퀘어

```
#include <stdio.h>
#define MAX_SIZE 15   /*maximum size of square */
void main(void)
{ /* construct a magic square, iteratively */
     int square[MAX_SIZE][MAX_SIZE];
     int i, j, row, column;                /* indexes * /          Θ(1)
     int count;                            /* counter */
     int size;                             /* square size */

     printf("Enter the size of the square: ");
     scanf("%d", &size);

     /* check for input errors */
      if (size <1 || size > MAX_SIZE +1) {                        Θ(1)
               fprintf(stderr, "Error!  Size is out of range\n")
               exit(EXIT_FAILURE);
       }
       if (!(size %2)) {
               fprintf(stderr, "Error! Size is even\n");
               exit(EXIT_FAILURE);                                Θ(1)
       }

       for (i = 0; i < size; i++)
              for(j = 0; j < size; j++)                           Θ(n^2)
                      square[i][j] = 0;
```

점근 표기법(Asymptotic Notation : O,Ω,Θ)(11)

```
    square[0][(size-1) / 2] = 1;  /* middle of first row */
    /* i and j are current position */
    i = 0;
    j = (size-1) /2;
    for(count = 2; count <= size *size; count++) {
        row = (i-1<0) ? (size - 1) : (i-1);     /*up*/
        column = (j-1<0) ? (size-1) : (j-1);    /*left*/

        if(square[row][column])       /*down*/
            i = (++i) %size;
        else {                        /*square is unoccupied */
                 i = row;
                 j = (j-1<0) ? (size -1) : --j;
         }
         square[i][j] = count;
     }
     /* output the magic square */
     printf("Magic Square of size %d : \n\n", size);
     for (i=0; i<size; j++)  {
           for (j=0; j<size; j++)
                 printf("%5d", square[i][j]);
           printf("\n")
      }
      printf("\n\n");
}
```

$\Theta(n^2)$

$\Theta(n^2)$

'매직 스퀘어'의 시간복잡도 = $\Theta(n^2)$

실용적인 복잡도(Practical Complexities)(1)

- 함수 값

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1024	32,768	4,294,967,296

$2^{40} = (2^{10})^4$
$= (1024)^4$
$= (1000 + 24)^4$
$= (10^3 + 24)^4$
$\sim 1.1*10^{12}$

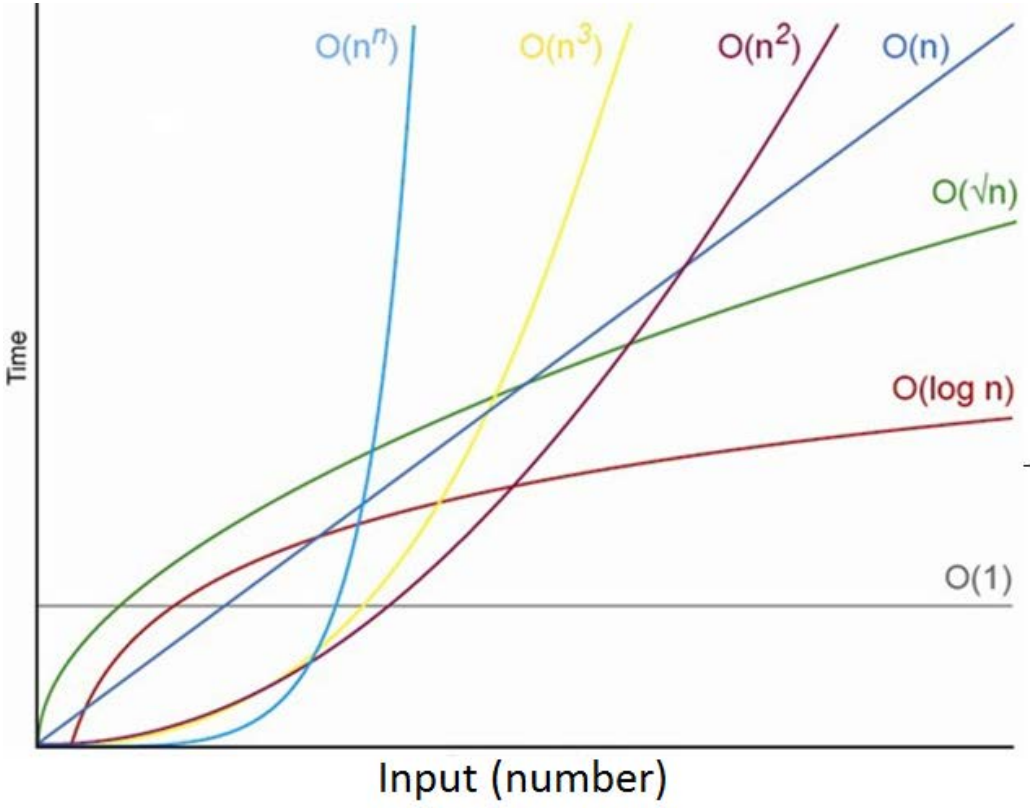
- 함수 2^n은 n이 증가함에 따라 더욱 빠르게 증가함
- 프로그램 실행을 위해 2^n단계가 필요하다면 n=40일 때
  - 요구되는 단계 수는 대략 $1.1*10^{12}$
- 1초에 10개의 단계를 수행하는 컴퓨터라면 약 18.3분 요구
- n=50 이면 13일, n=60 이면 310.56년, n=100 이면 $4*10^{13}$년
  - 따라서 유용도는 n이 아주 작을 때($n \leq 40$)로 제한됨

Fast Computer vs. Better Algorithm

Algorithmic improvement more useful than hardware improvement. E.g. 2^n to n^3

실용적인 복잡도(Practical Complexities)(2)

Algorithm = 답(해)가 있는 문제의 Optimized Solution을 찾는 것



Big-O: functions ranking

BETTER

- O(1) constant time
- O(log n) log time
- O(n) linear time
- O(n log n) log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Fast Computer vs. Better Algorithm

Algorithmic improvement more useful than hardware improvement. E.g. 2^n to n^3

성능측정
(Performance Measurement)

시간측정

- 방법 1: clock을 사용
- 방법 2: time을 사용

asymptotic complexity = 분석영역

실제 수행 시간 = 측정영역

	방법 1	방법 2
시작시간	`start = clock();`	`start = time(NULL);`
종료시간	`stop = clock();`	`stop = time(NULL)`
반환타입	`clock_t`	`time_t`
초 단위 결과	`duration = ((double)(stop-start))/ CLOCKS_PER_SEC`	`duration = (double) difftime(stop, start);`

< C 에서 시간을 재는 방법 >

```
typedef /* unspecified */ clock_t;
typedef /* unspecified */ time_t;
```

선택정렬(Selection Sort) 함수의 최악의 성능(1)

< 선택 정렬 함수의 첫 번째 시간 측정 프로그램 >

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
        int i, n, step = 10;
        int a[MAX_SIZE];
        double duration;
        clock_t start;

        /* time for n = 0, 10, …., 100, 200, …., 1000 */
        printf("     n     time\n");
        for(n = 0; n <= 1000; n += step) { /* get time for size n */
            /* initialize with worst-case data */
            for (i=0; i<n; i++)
                    a[i] = n-i;
            start = clock();
            sort(a, n);
            duration = ((double)) (clock() - start))/ CLOCK_PER_SEC;
            printf("%6d    %f\n", n, duration);
            if (n == 100) step = 100;
        }
}
```

← 데이터를 역순으로 저장

측정시간이 짧을 경우 논리적으로 정확하더라도
실행시간을 정밀하게 측정 하지 못함

선택정렬(Selection Sort) 함수의 최악의 성능(2)

< 보다 정확한 선택 정렬 함수의 시간 측정 프로그램 >

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
    int i, n, step = 10;
    int a[MAX_SIZE];
    double duration;
    /* time for n = 0, 10, …., 100, 200, …., 1000 */
    printf("     n     time\n");
    for(n=0; n <= 1000; n += step) {
        /* get time for size n */
        long repetitions = 0;
        clock_t start = clock( );
        do {
              repetitions++; /* 정렬을 몇 번 시도했는지 카운트 */
             /* initialize with worst-case data */
             for (i=0; i<n; i++) a[i] = n – i;
             sort(a, n);
       } while (clock( ) – start < 1000); /* repeat until enough time has elapsed */
       duration = ((double) (clock() - start)) / CLOCK_PER_SEC;
       duration /= repetitions; /* 해당 n에 대한 정렬 수행 시간 평균값 */
       printf("%6d %9d %f \n"), n repetitions, duration);
       if (n ==100) step = 100;
     }
}
```

1초 이상 만 측정

선택 정렬의 최악의 경우에서의 성능

n	repetitions	time	n	repetitions	time
0	8690714	0.000000	100	44058	0.000023
10	2370915	0.000000	200	12585	0.000079
20	604948	0.000002	300	5780	0.000173
30	329505	0.000003	400	3344	0.000299
40	205605	0.000005	500	2096	0.000477
50	145353	0.000007	600	1516	0.000660
60	110206	0.000009	700	1106	0.000904
70	85037	0.000012	800	852	0.001174
80	65751	0.000015	900	681	0.001468
90	54012	0.000019	1000	550	0.001818

< 선택 정렬의 최악의 경우에서의 성능(단위는 초) >