3 / 15

- 재귀적 알고리즘 연습1
    - 순차 탐색 vs 이분 탐색 (정렬) → 이분 탐색 시간 복잡도 Log2^n
    - 관찰
        - Top-down vs Bottom-up
        - Recursion vs Iteration
        - 재귀적 구조
        - Base condition
        - 수렴
        - 매개변수

$$S(A,n,x) = \begin{cases} n-1 & \text{if } A[n-1]=x \\ S(A,n-1,x) & \text{if } A[n-1]!=x \\ -1 & \text{if } n=0 \end{cases}$$

$$BS(A,\ p,\ r,\ x) = \begin{cases} m & m=(p+r)/2,\ \text{if } A[m]=x \\ BS(A,m+1,r,\ x) & \text{if } A[m] <x \\ BS(A,p,m-1,\ x) & \text{if } A[m] >x \\ -1 & \text{if } p>r \end{cases}$$

- 검색
  - 컴퓨터에 저정한 자료 중에서 원하는 항목을 찾는 작업
    - 검색 성공 - 원하는 항목을 찾은 경우
    - 검색 시패 - 원하는 항목을 찾지 못한 경우
  - 탐색 키를 가진 항목을 찾는 것
    - 탐색 키(Search Key) - 자료를 구별하여 인식할 수 있는 키
  - 삽입/삭제 작업에서의 검색
    - 원소를 삽입하거나 삭제할 위치를 찾기 위해서 검색 연산 수행
- 검색 방법
  - 수행 위치에 따른 분류
    - 내부 검색 - 메모리 내의 자료에 대해서 검색 수행
    - 외부 검색 - 보조 기억 장치에 있는 자료에 대해서 검색 수행
  - 검색 방식에 따른 분류
    - 비교 검색 방식 (Comparison Search Method)
    - 계산 검색 방식 (Non-Comparison Method)

- 검색 알고리즘
  - 순차 검색
    - 일렬로 된 자료를 처음부터 마지막까지 순서대로 검색한ㄴ 방법
    - 가장 간단하고, 직접적인 검색 방법
    - 배열이나 연결 리스트로 구현된 순차 자료구조에서 원하는 항목을 찾는 방법
    - 검색 대상 자료가 많은 경우에 비효율적이지만, 알고리즘이 단순하여 구현이 용이함
  - 이진 검색
    - 이분 검색, 보간 검색 (Interpolation Search) 이라고도 함
    - 자료의 가운데에 있는 항목을 키 값과 비교하여, 다음 검색 위치를 결정
      - 찾는 키 값 > 원소의 키 값 : 오른쪽 부분에 대해서 검색 실행

      - 찾는 키 값 < 원소의 키 값 : 왼쪽 부분에 대해서 검색 실행
    - 키를 찾을 때까지 이진 검색을 순환적으로 반복 수행함으로써 검색 범위를 반으로 줄여가면서 더 빠르게 검색
    - 분할정복(Divide-And-Conquer)기법을 이용한 검색 방법
      - 검색 범위를 반으로 분할하는 작업과 검색 작업을 반복 수행
    - 정렬되어있는 자료에 대해서 수행하는 검색 방법

[정렬되어 있지 않은 자료의 경우 예]

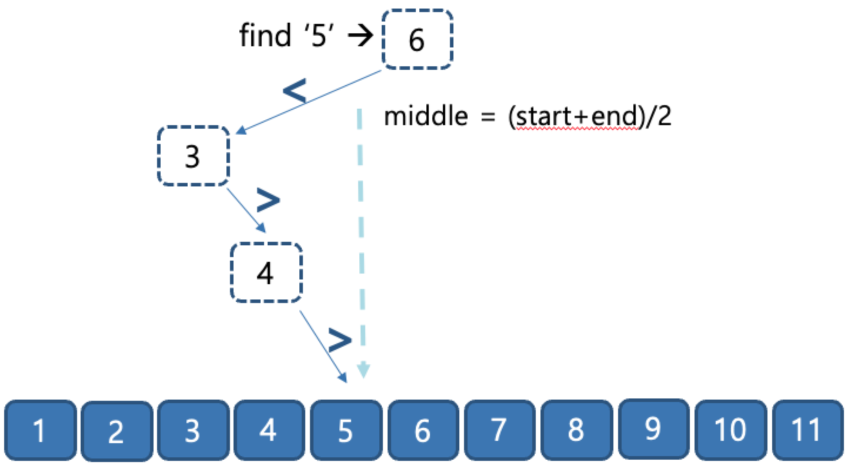
```
seqSearch1(a[], key) {
   n = a.length;
   i = 0;
   while (i<n && a[i]!=key) {
      i++;
   }
   if (i<n) return i;
   else return -1;
}
```

[정렬되어 있는 자료의 경우 예]

```
seqSearch2(a[], key) {
   n = a.length;
   i = 0;
   while (i<n && a[i]<key) {
        i++;
   }
   if (a[i]==n) return i;
   else return -1;
}
```

  - 비교횟수 → 찾고자 하는 원소의 위치에 따라 결정
  - 정렬되지 않은 경우 순차 검색의 평균 비교 횟수
    - 찾는 원소가 첫 번째 원소라면 비교횟수는 1번, 두 번째 원소라면 비교횟수는 2번, 세 번째 원소라면 비교횟수는 3번, 찾는 원소가 i번째 원소이면 i번, …
    - ⇒ 1 / n (1 + 2 + 3 + 4 + … + n) ⇒ (n + 1) / 2 → 평균 시간 복잡도 : O(N)
  - 정렬되어있는 경우 순차 검색의 평균 비교 횟수
    - ⇒ 1 / n (1 + 2 + 3 + 4 + … + n) * 1 / 2 ⇒ (n + 1) / 4 → 평균 시간 복잡도 : O(N)
  - 재귀는 ?

- 이진 탐색
    - 정렬이 되어 있는 경우, 비교 횟수를 줄이자!
        - 중간의 값을 임의로 선택하여, 그 값과 찾고자 하는 값의 크기를 비교하여, 다음 검색의 범위를 빠르게 좁혀간다.



    - 이진 탐색의 구현
        - Iteration
            - 장점
            - 단점
        - Recursion
            - 장점
                - 관계중심으로 문제를 간명하게 볼 수 있다.
                    - 명확한 의미 전달
                    - 간결한 프로그램
            - 단점
                - 재귀적 해법을 이용하면, 심한 중복 호출이 발생한다. but 메모이제이션 기법을 이용해서 해결이 가능하다.
        - Base case?
        - 수렴?

- 명시적 파라메터?

[Iteration]

```
binarySearchIteration(a[], key, start, end) {
    if (start>end) return -1;
    while (start<=end) {
      mid=(start+end)/2;
      if (a[mid]==key) return mid;
      else if (a[mid]<key)
         start=mid+1;
      else
         end = mid-1;
  }
  return -1;
}
```

[Recursion]

```
binarySearchRecursion(a[], key, start, end) {
  if (start>end) return -1;
  mid=(start+end)/2;
  if (a[mid]==key) return mid;
    else if(a[mid]<key)
          return binarySearchRecursion(a, key, mid+1, end);
    else
          return binarySearchRecursion(a, key, start, mid-1);
}
```

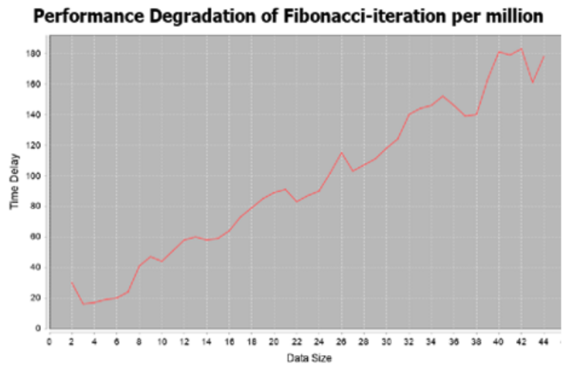
- 재귀적 알고리즘 연습
  - N * N에서 경로상 숫자들을 합할 때 최대값을 구하라
  - 오른쪽, 아래로만 이동이 가능하다.

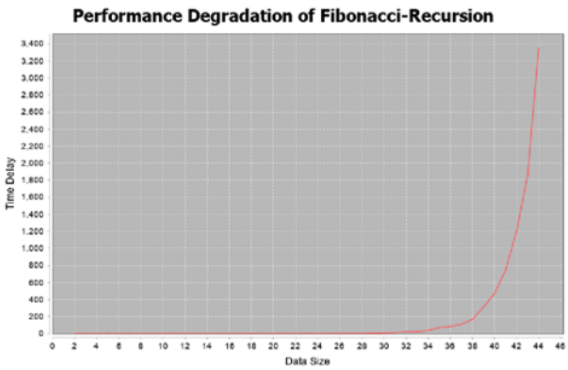
1	2	1	5	8	4
4	1	9	4	2	3
8	5	4	3	8	2
1	5	3	5	7	3
4	7	7	9	2	8
2	4	6	3	1	4

$$
\mathrm{Sum}(i,j) = \begin{cases} \mathrm{maze}[i][j] & i=0,\ j=0 \\ \mathrm{maze}[i][j] + \mathrm{Sum}(i,\ j-1) & i=0 \\ \mathrm{maze}[i][j] + \mathrm{Sum}(i-1,\ j) & j=0 \\ \mathrm{maze}[i][j] + \mathrm{Max}(\mathrm{Sum}(i,\ j-1),\ \mathrm{Sum}(i-1,\ j)) & \text{else} \end{cases}
$$

- 재귀는 스택 영역안에서 담겨, Base Case까지 돌아오게 된다. 스택이므로, 제일 처음에 들어온 Basecase가 제일 마지막에 빠져나오게 되며, 그외의 값은 가지친 연산을 통해 값이 도출되게 된다.

- 성능 비교





- 알고리즘 분석 목적

- 무결성 확인
- 자원 사용의 효율성 파악
  - 시간
  - 메모리, 통신대 역, …
- 알고리즘 효율성 분석 방법
  - 공간 복잡도
    - 알고리즘을 프로그램으로 실행하여, 완료하기까지의 소요시간
    - 소요 시간
      - 컴파일 시간 : 프로그램마다 거의 고정적인 시간 소요
      - 실행 시간 : 컴퓨터의 성능에 따라 달라질 수 있음 → 실제 실행시간보다 명령문의 실행 빈도수에 따라 계산
    - 알고리즘의 수행 시간을 좌우하는 기준은 다양하게 잡을 수 있다.
      - for의 반복 횟수, 특정한 행이 수행되는 횟수, 함수의 호출 횟수, …
    - 크기가 작은 문제
      - 알고리즘의 효율성이 중요하지 않다.
      - 비효율적인 알고리즘도 무방
    - 크기가 충분히 큰 문제
      - 알고리즘의 효율성이 중요하다
      - 비효율적인 알고리즘은 치명적
    - 입력의 크기가 충분히 큰 경우에 대한 분석을 점근적 분석이라한다.

$$\lim_{n\to\infty} f(n)$$

– 예 : $\lim_{n\to\infty}\left(\frac{2n^2+999999n+1000000000000}{n^2}\right) = ?$

```
sample1(A[ ], n)
{
        k = 2n -1;
        return A[k];
}
```

→ 상수 시간 (n 에 무관)

```
sample2(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                sum← sum+ A[i] ;
        return sum ;
}
```

→ $\propto n$

```
sample3(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n
                        sum← sum+ A[i]*A[j] ;
        return sum ;
}
```

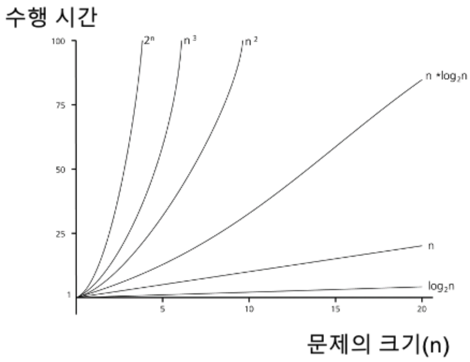
→ $\propto n^2$

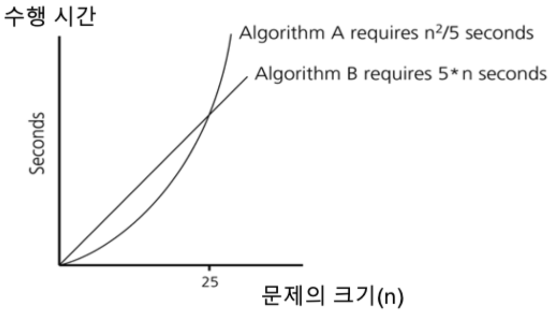
```
sample4(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n {
                        k ← A[1 ... n]에서 임의로 ⌊n/2⌋개를 뽑을 때 이들 중 최댓값 ;
                        sum ← sum + k ;
                }
        return sum ;
}
```

→ $\propto n^3$

알고리즘 수행시간

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

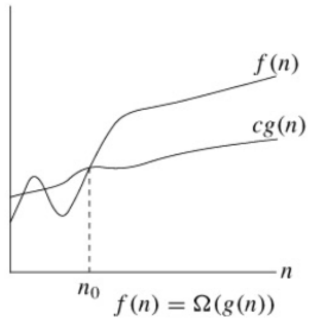




- Big-O
  - 점근적 표기법으로 각 위의 시간복잡도에 O(N), O(N^2), O(log2^n) 등으로 할 수 있다.

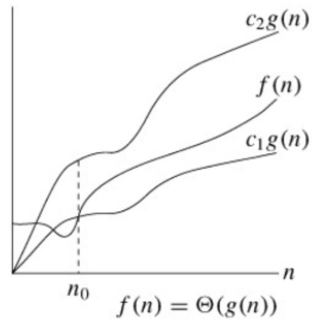
점근적 표기법 Asymptotic Notation

$\Omega(g(n))$
- 적어도 $g(n)$의 비율로 증가하는 함수
- $O(g(n))$과 대칭적



- Formal definition
  - $\Omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t.} \forall n \geq n_0, c\mathbf{g}(n) \leq \mathbf{f}(n) \}$
- 직관적 의미
  - $\mathbf{f}(n) = \Omega(\mathbf{g}(n)) \Rightarrow$ **f**는 **g**보다 느리게 증가하지 않는다

$\Theta(g(n))$
- $g(n)$의 비율로 증가하는 함수



- Formal definition
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- 직관적 의미
  - $\mathbf{f}(n) = \Theta(\mathbf{g}(n)) \Rightarrow$ **f**는 **g**와 같은 정도로 증가한다

- 병합 정렬
  - 재귀 용법을 활용한 정렬 알고리즘
    - 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눈다.
    - 각 부분 리스트를 재귀적으로 합병 정렬을 이용해 정렬한다.
    - 두 부분리스트를 다시 하나의 정렬된 리스트로 합병한다.
      - 데이터가 4개일 때 병합 정렬의 논리
        - 정렬되지 않은 배열을 끝까지 분리하는 단계
        - 분리한 데이터를 단계별로 합치는 단계
        - 1, 9, 3, 2 일때
        - [1, 9], [3, 2]로 나누고
        - [1], [9], [3], [2]로 나누어 진다. 분리단계 끝
        - [1, 9]를 정렬해서 합치고, [2, 3]을 정렬해서 합친다.
        - 이후 [1, 2, 3, 9]로 정렬해서 합친다.

        - N의 길이를 가진 배열이 있다고 했을 때, log2^n개 만큼 만들어지고, 각각의 단계는 2i * n / 2i를 가지므로 O(n)을 1단계가 시간 복잡도를 가지고, log2^n의 단계가 있으므로, 총 시간 복잡도는 O(nlogn)의 시간 복잡도를 가지게 된다.
        - 분할 정복
            - 문제를 나룰 수 없을 때 까지 나누어서 각각을 풀면서 다시 합병하여, 문제의 답을 얻는 알고리즘
            - 하양식 접근법으로, 상위의 해답을 구하기 위해, 아래로 내려가면서 하위의 해답을 구하는 방식
                - 일반적으로 재귀함수로 구현
            - 문제를 잘게 쪼갤 때, 부분 문제는 서로 중복되지 않음
                - 예 : 병합 정렬, 퀵 정렬 등
    - Split 단계와 Merge 단계로 나뉨

- 퀵 정렬
    - 정렬 알고리즘의 꽃
    - 기준점(pivot 이라고 부른다.)을 정해서, 기준점 보다 작은 데이터는 왼쪽(left), 큰 데이터는(right)으로 몽는 함수를 작성한다.
    - 각 왼쪽(left), 오른쪽(right)는 재귀 용법을 사용해서, 다시 동일 함수를 호출하여 위 작업을 반복한다.
    - 함수는 왼쪽(left) + 기준점(pivot) + 오른쪽(right)를 리턴한다.
    - 나누는 기준과 합칠 때 정렬을 하지 않을 뿐 병합과 유사하다.
    - 시간복잡도는 O(nlon)으로 되지만, 항상 모든 정렬은 최악의 경우 O(n^2)이 걸리게 된다.