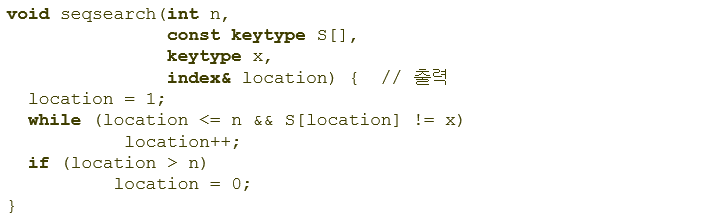
**<Chapter 1>**

* **알고리즘의 정의**
* 문제를 해결할 수 있는 잘 정의된 유한 시간 내에 종료되는 계산적인 절차
* Method는 시간에 구애 받지 않지만, 알고리즘은 유한 시간 내에 종료되어야 한다.
* **Parameter:** 문제에서 특정 값이 주어지지 않은 변수 (매개변수)
* **Instance:** Parameter에 특정 값을 지정한 것 (입력)
* **Solution:** 주어진 사례에 관한 질문에 대한 답 (출력)

ex) n 개의 수로 구성된 리스트 S를 비내림차순으로 정렬하라. 해답은 S를 비내림차순으로 정렬한리스트 이다.

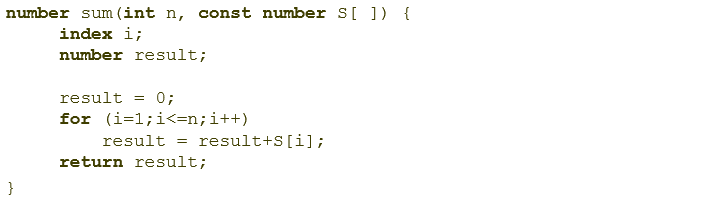
* Parameter: S, n
* Instance: S = [10, 7, 11, 5, 13, 8], n = 6
* Solution: [5, 7, 8, 10, 11, 13]
* **자연어:** 문제를 정확하게 기술하는데 어려움이 있으며 상대적으로 긴 문장이 필요하다. 또한 해석하는 사람에 따라서 다르게 해석할 수도 있다.
* **프로그래밍 언어:** 너무나 구체적으로 기술해야하기 때문에 알고리즘을 이해하는데 어렵다.
* **의사코드:** 실제 프로그램과 가깝게 계산과정을 표현할 수 있는 언어이며 간결하면서도 정확한 의미를 표현 가능하다.
* **의사코드 vs C++**
* 의사코드는 배열 인덱스에 제한이 없는 반면 프로그래밍 언어는 무조건 0으로 시작한다.
* 프로시저의 parameter에 2차원 배열 크기의 가변성을 허용한다. ex) void pname(A[][]) {…}
* 지역 배열에 변수 인덱스를 허용한다. ex) keytype S[low..high];
* 수학적 표현식 허용한다. ex) 10 <= a <= 20;
* C++ 에는 없는 타입 사용 가능 ex) index: 첨자로 사용되는 정수, number: 정수 또는 실수
* 순차 검색 알고리즘 – worst-case analysis: W(n) = n

문제: n개의 키로 구성된 배열 S에 키 x가 있는가  
입력: 양의 정수 n, 1에서 n까지의 첨자를 가진 배열 S, 그리고 x  
출력: S안에 x의 위치를 가리키는 location (S 안에 x가 없으면 0)



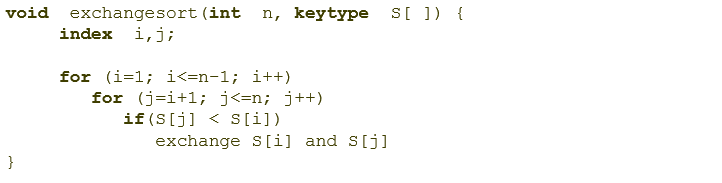
* 배열의 수 더하기 – every-case analysis: T(n) = n

문제: n개의 수로 된 배열 S에 있는 모든 수를 더하라  
입력: 양의 정수 n, 수의 배열 S(첨자는 1부터 n까지)  
출력: S에 있는 수의 합



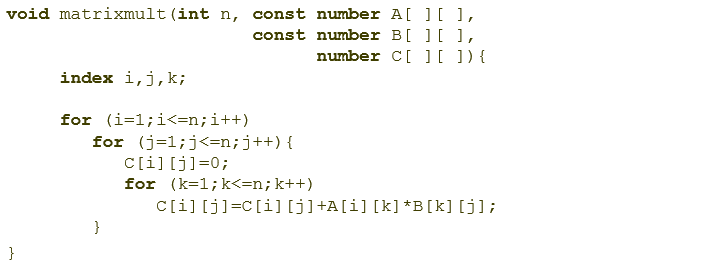
* 교환 정렬 – every-case analysis: T(n) = n(n-1)/2

문제: 비내림차순으로 n개의 키를 정렬하라  
입력: 양의 정수 n, 키의 배열 S (첨자는 1부터 n까지)  
출력: 키가 비내림차순으로 정리된 배열 S



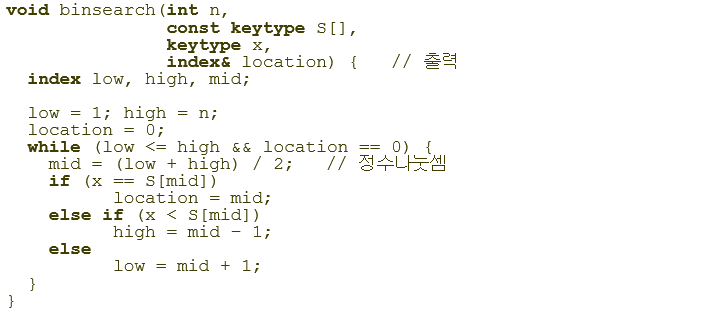
* 행렬 곱셈 – every-case analysis: T(n) = n3

문제: 두개의 n x n 행렬의 곱을 구하여라  
입력: 양의 정수 n, 2차원 배열 A, B (행렬의 행과 열은 모두 첨자가 1부터 n까지)  
출력: A와 B의 곱이 되는 2차원 배열 (행렬의 행과 열은 모두 첨자가 1부터 n까지)



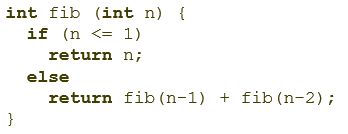
* 이분 검색 알고리즘

문제: 원소가 n개인 정렬된 배열 S에 x가 있는가?  
입력: 자연수 n, (비내림차순으로 정렬된) 배열 S (인덱스의 범위는 1부터 까지), 원소 x  
출력: location, S에서 x가 있는 위치 (만약 x가 S에 없으면 0)



* 피보나찌 수 구하기 -> 재귀를 사용할 경우 같은 값을 중복해서 계산해서 매우 비효율적

문제: 피보나치 수열에서 n번째 수를 구하시오.  
입력: 양수 n  
출력: n 번째 피보나찌 수

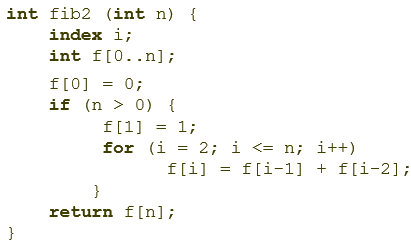


* 함수 호출 계산

T(n) = T(n – 1) + T(n – 2) + 1 (n > 1)  
T(n) >= 2*n*/2

* n번쨰 피보나찌 구하기

문제: 피보나치 수열에서 n번째 항를 구하시오.  
입력: 양수 n  
출력: n 번째 피보나찌 항



* 함수 호출 계산: T(n) = n + 1 -> 재귀보다 훨씬 빠름
* 수학적 귀납법

1. 귀납 출발점(base case): n = 0 or n = 1 일 때 주장이 사실임을 보인다.
2. 귀납 가정(inductive hypothesis): 어떤 n에 대해서 주장이 사실임을 가정한다.
3. 귀납 절차(inductive step): n + 1에 대해서 주장이 사실임을 보인다.

* 알고리즘의 분석
* 공간복잡도 분석: 입력 크기에 따라서 작업공간(메모리)이 얼마나 필요한지 결정하는 절차
* 시간복잡도 분석: 입력 크기에 따라서 단위 연산이 몇 번 수행되는지 결정하는 절차

\*\*\* 단위 연산: 비교문, 지정문   
\*\*\* 입력 크기: 배열 원소의 개수, 리스트의 길이 등

* Every-case analysis: 입력의 값에 상관없이 항상 단위 연산의 실행 횟수가 일정하므로 입력 크기에만 영향을 받는다. -> T(n)
* Worst-case analysis: 입력의 값에 대해서 단위 연산을 실행하는 횟수가 모두 다르므로 입력 크기와 입력 값에 모두 영향을 받는다. -> W(n)
* Average-case analysis: 입력 크기 n에 대해서 그 알고리즘이 수행할 단위 연산의 평균 횟수 기대치로 입력 크기에만 종속된다. -> A(n)
* Best-case analysis: 입력 크기 n에 대해서 그 알고리즘이 실행할 단위 연산의 최소 횟수로 입력 크기와 입력 값에 모두 영향을 받는다. -> B(n)
* 차수

O( ) Big oh – 주어진 복잡도 함수 f(n)에 대해서 O(f(n))은 정수 N 이상의 모든 n에 대해서 다음 부등식이 성립하는 양의 실수 c와 음이 아닌 정수 N이 존재하는 복잡도 함수 g(n) 의 집합 -> *궁극적으로 항상 g(n)은 f(n)보다 아래에 있다 / 최소 g(n) 보다 좋다*

**g(n) <= c x f(n)** (asymptotic upper bound)

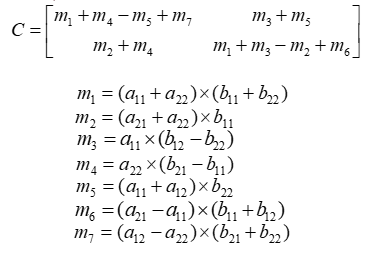
o( ) small oh –

Ω( ) Omega –

ω( ) Small omega –

Θ( ) theta -

**<Chapter 2>**

* **분할정복식 설계 전략**
* 하향식(Top-down)접근 방법
* 나누어진 부분들 사이에 서로 상관관계가 없는 문제를 해결하는데 적합  
  분할: 해결하기 쉽도록 문제를 여러 개의 작은 부분으로 나눈다.  
  정복: 나눈 작은 문제를 각각 해결한다.  
  통합: 해결된 해답을 모은다.
* **이분 검색**
* 문제: 크기가 n인 정렬된 배열 S에 x가 있는지를 결정하라.  
  입력: 자연수 n, 비내림차순으로 정렬된 배열 S[1…n], 찾고자 하는 항목 x  
  출력: location, x가 S의 어디에 있는지의 위치. 만약 x가 S에 없다면 0
* 전략: x가 배열의 중간에 위치하고 있는 항목과 같으면 x 찾음, 그렇지 않으면:  
  - 분할: 배열을 반으로 나누어서 x가 중앙에 위치한 항목보다 작으면 왼쪽에 위치한 배열 반쪽을 선택하고, 그렇지 않으면 오른쪽에 위치한 배열 반쪽을 선택  
  - 정복: 선택된 반쪽 배열에서 x를 찾음.  
  - 통합: X
* 설계  
  - 입력 파라미터 n, S, x는 알고리즘 수행 중 변하지 않으므로 전역 변수로 설정  
  - 꼬리 재귀호출: 재귀호출이 알고리즘의 마지막 부분에서 이루어지는 형태
* 최악의 경우 시간복잡도 분석  
  - 단위 연산: x와 S[mid] 비교  
  - 입력 크기: 배열의 크기 n (= high – low + 1)  
  - 단위연산으로 설정한 조건 문은 2번 수행되지만 사실상 비교는 한번 이루어진다.  
  - 경우 1: 검색하게 될 반쪽 배열의 크기가 항상 정확하게 n/2 가 되는 경우  
  - 경우 2: 일반적인 경우  
  - 분석 1: W(n) = W(n/2) + 1, n > 1, n = (k 1)  
  - 점화식 1: W(n) =   
  - 분석 2: W(n) = W() + 1  
  - 점화식 2: W(n) = + 1
* **합병 / 합병 정렬**
* 합병 정렬: 계속 나누고 마지막에 합병 알고리즘을 사용한다.   
  문제: n 개의 정수를 비내림차순으로 정렬하시오.  
  입력: 정수 n, 크기가 n인 배열 S[1…n]  
  출력: 비내림차순으로 정렬된 배열 S[1…n]  
  ***시간 복잡도 연산***  
  - 단위 연산: U[i]와 V[j]의 비교  
  - 입력 크기: 2개의 입력 배열에 각각 들어 있는 항목의 개수 – h & m  
  - 분석: i = h + 1, j = m일 때가 최악의 경우이며 이때 단위 연산의 실행 횟수는 h + m – 1  
   -> W(h, m) = h + m - 1
* 합병: 두 배열에 있는 첫 번째 값부터 비교해서 차근차근 비교해서 S에 넣어준다.  
  문제: 두 개의 정렬된 배열을 하나의 정렬된 배열로 합병하시오.  
  입력: 양의 정수 h, m / 정렬된 배열 U[1…h], V[1…m]  
  출력: U와 V에 있는 키들을 하나의 배열에 정렬한 S[1…h + m]  
  ***시간 복잡도 연산***  
  - 단위 연산: 합병 알고리즘 merge 에서 발생하는 비교  
  - 입력 크기: 배열 S에 들어 있는 항목의 개수   
  - 분석: 최악의 경우 W(h, m) = W(h) + W(m) + h + m - 1   
   -> W(n) = 2W(n/2) + n – 1, n> 1, n = <- *n = 라고 가정하는게 중요하다!*  
  - 점화식:
* 문제점: 공간 복잡도를 분석하면 합병 정렬의 경우 재귀 호출할 때마다 추가적인 저장소가 필요하기 때문에 결과적으로 저장소의 크기는 이다. 이를 n으로 줄여보자!  
  -> 기존에 추가적으로 만든 공간을 재사용해서 줄여준다.
* **빠른 정렬: 중간 중심으로 왼쪽엔 작은 것, 오른쪽엔 큰 것 반복**
* 빠른 정렬 알고리즘  
  문제: n개의 정수를 비내림차순으로 정렬  
  입력: 정수 n > 0, 크기가 n인 배열 S[1…n]  
  출력: 비내림차순으로 정렬된 배열 S[1…n]  
  ***시간 복잡도 연산***- 단위 연산: 분할알고리즘의 S[i[와 pivotitem과의 비교  
  - 입력 크기: 배열 S가 가지고 있는 항목의 수 n  
  - 분석: 최악의 경우 T(n) = T(0) + T(n – 1) + n – 1  
  - 점화식: T(n) =
* 분할 알고리즘:   
  문제: 빠른 정렬을 하기 위해 배열 S를 둘로 나눈다.  
  입력: 첨자 low, high / S의 부분 배열  
  출력: 첨자 low에서 high까지의 S의 부분 배열의 기준점 / pivotpoint  
  ***시간 복잡도 연산***- 단위 연산: S[i]와 pivotitem과의 비교  
  - 입력 크기: 부분 배열이 가지고 있는 항목의 수, n = high – low + 1  
  - 분석/점화식: T(n) = n - 1
* **행렬 곱셈**
* 문제: n x n 크기의 행렬의 곱을 구하시오.  
  입력: 양수 n / n x n 크기의 A 와 B  
  출력: 행렬 A X B 인 C  
  ***시간 복잡도 분석***단위 연산: 가장 안쪽의 루프에 있는 곱셈하는 연산  
  입력 크기: 행과 열의 수 n  
  분석/점화식: T(n) = n x n x n =
* **쉬트라쎈**
* 문제: n이 2의 거듭제곱일 때, n x n 크기의 두 행렬의 곱을 구하시오.  
  입력: 정수 n / n x n 크기의 행렬 A, B  
  출력: 행렬 A X B 인 C  
  ***시간 복잡도 분석***단위 연산: 덧셈, 뺄셈하는 연산  
  입력 크기: 행과 열의 수 n  
  분석: T(n) = , n > 1, n =   
  -> 버전 2 참고
* **큰 정수 계산법**
* 문제: 2개의 큰 정수 u와 v를 곱하라  
  입력: 큰 정수 u와 v, 크기 n  
  출력: u와 v의 곱  
  ***시간 복잡도 분석***단위 연산: 덧셈, 뺄셈, divide , mod , x   
  입력 크기: 정수의 자리 수 n  
  분석: W(n) = 4W + cn, n > s 이고 n이 2의 거듭제곱  
  -> 버전 2 참고
* **도사 정리**

<Chapter 3>

* 동적 계획 설계
* 상향식 해결법(bottom-up approach)
* 분할정복식과 마찬가지로 문제를 나눈 후에 나누어진 부분들을 먼저 풀지만 인덱스를 효과적으로 설정하여 작은 문제들의 중복 해결을 배제한다.  
  즉, 작은 문제를 해결하고 이를 큰 문제의 해결로 확산한다.