알고리즘해석 강의노트 v0.7

한양대학교 전자계산학과
3학년 알고리즘해석
2000년 봄학기

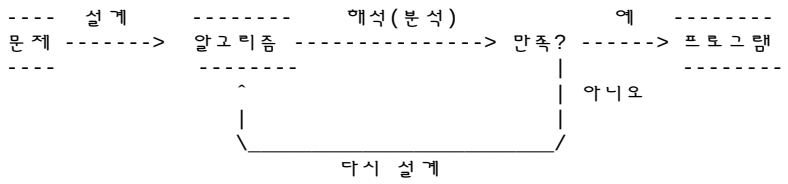
Department of Computer Science and Engineering
Hanyang University, 425-791, Korea
doh@cse.hanyang.ac.kr, hkoh@cse.hanyang.ac.kr

목 차

제 1 장

서막

- 학습 목표
  - ▷ 알고리즘의 설계(design) 방법을 배운다.
  - ▷ 알고리즘의 분석(analysis)하여 계산복잡도를 구하는 방법을 배운다.
  - ▷ 문제에 대한 계산복잡도(computational complexity)를 공부한다.
- 프로그램 설계 과정



제 1 절 알고리즘

- **정의 1.1: 알고리즘(Algorithm)** - 문제에 대한 답을 찾기 위해서 계산하는 절차를 알고리즘이라고 한다. 다시 말하면,
  - ▷ 단계별로 주의 깊게 설계된 계산 과정
  - ▷ 입력을 받아서 출력으로 전환시켜주는 일련의 계산 절차 (computational steps)
- 보기
  - ▷ 문제(problem): 전화번호부에서 "홍길동"이라는 사람의 이름 찾기
  - ▷ 알고리즘(algorithm):
    1. 순차검색(sequential search): 첫 쪽부터 홍길동이라는 이름이 나올 때까지 순서대로 찾는다.
    2. 수정된 이진검색(modified binary search): 전화번호부는 가나다 순으로 되어 있으므로 먼저 "ㅎ"이 있을 만한 곳으로 넘겨본 후 앞뒤로 뒤적여 가며 찾는다.
  - ▷ 분석(analysis): 어떤 알고리즘이 더 좋은가?

1.1 문제의 표기

- 문제를 표기하는데 필요한 사항
  - ▷ 문제(problem): 답을 찾고자 주는 질문
  - ▷ 매개변수(parameter): 문제를 설명하는 과정에서 어떤 특정한 값이 지정되어 있지 않은 변수(variable)
  - ▷ 문제의 사례(instance) = 입력(input): 문제에 주어진 매개변수에 어떤 특정 값을 지정한 것
  - ▷ 사례에 대한 해답(solution) = 출력(output): 어떤 사례에 대하여 문제에 의해서 제기된 질문에 대한 답
- 보기: 검색(Searching)
  - ▷ 문제: n개의 수(number)를 가진 리스트 S에 x라는 수가 있는지 결정하시오. 답은 x가 S에 있으면 "예", 그렇지 않으면 "아니오".
  - ▷ 매개변수: S(리스트), n(S 안에 있는 수의 개수), x(찾고자 하는 항목)
  - ▷ 입력의 예: $S = [10, 7, 11, 5, 13, 8]$, $n = 6$, $x = 5$
  - ▷ 출력의 예: "예"

1.2 알고리즘의 표기

- 자연어(영어 또는 한글)
- 프로그래밍언어: C, C++, Java, ML 등
- 의사코드(pseudo-code): 직접 실행할 수 있는 프로그래밍언어는 아니지만 거의 실제 프로그램에 가깝게 계산과정을 표현할 수 있는 언어 (알고리즘은 보통 의사코드로 표현한다) - 여기서는 C/C++에 가까운 의사코드를 사용함

1.3 보기: 검색하기

1.3.1 순차검색(Sequential Search) 알고리즘

- 문제: 크기가 n인 배열(array) S에 x가 있는가?
- 입력(매개변수): (1) 양수 n, (2) 배열 S[1..n], (3) 찾고자 하는 항목 x
- 출력: x가 S의 어디에 있는지의 위치. 만약 x가 S에 없다면 0.
- 알고리즘(자연어): x와 같은 항목을 찾을 때까지 S 배열에 있는 모든 항목을 차례로 비교한다. 만일 x와 같은 항목을 찾으면 S 배열 상의 위치를 내주고, S에 있는 모든 항목을 검사하고도 찾지 못하면 0을 내준다.
- 알고리즘(의사코드):

```
void seqsearch(int n,                    // 입력 (1)
               const keytype S[],        //      (2)
               keytype x,                //      (3)
               index& location)          // 출력
{
   location = 1;
   while (location <= n && S[location] != x)      // (A)
      location++;
   if (location > n)                              // (B)
      location = 0
}
```

- 조건문의 의미
  - (A): 아직 검사할 항목이 있고, x를 찾지 못했나?
  - (B): 모두 검사하였으나 x를 찾지 못했나?
- 관찰사항: 이 알고리즘으로 어떤 항목(key)을 찾기 위해서 배열 S에 있는 항목을 몇 개나 검색해야 하는가? 이는 찾고자 하는 항목이 어디에 위치하고 있는가에 달려 있겠지만, 최악의 경우, 즉, 찾고자 하는 항목이 S[n]에 위치하고 있거나, 아예 없는 경우에는 최소한 n개는 검색해야 한다.
- 좀 더 빠르게 찾을 수는 없을까? 사실 더 이상 빨리 찾을 수 있는 알고리즘은 존재하지 않는다. 왜냐하면 배열 S에 들어있는 항목들에 대한 정보가 전혀 없는 상황에서는 모든 항목을 검색하지 않고 임의의 항목 x를 항상 찾을 수 있다는 보장은 없기 때문이다. 그러나 다음 절에서와 같이 만약 배열 S가 이미 오름차순으로 정렬(sorting) 되어 있는 경우에는 좀 더 빠른 시간 안에 검색할 수가 있다.

1.3.2 이분검색(Binary Search) 알고리즘

- 문제: 크기가 n인 정렬된 배열(array) S에 x가 있는가?
- 입력(매개변수): (1) 양수 n, (2) 배열 S[1..n], (3) 찾고자 하는 항목 x
- 출력: x가 S의 어디에 있는지의 위치. 만약 x가 S에 없다면 0.
- 알고리즘:

```
void binsearch (int n,                  // 입력 (1)
                const keytype S[],      //      (2)
                keytype x,              //      (3)
                index& location)        // 출력
{
   index low, high, mid;

   low = 1; high = n;
   location = 0;
   while (low <= high && location == 0) {       // (A)
      mid = (low + high) / 2;
            // 정수 나눗셈 (나머지 버림)
      if (x == S[mid])
         location = mid;
      else if (x < S[mid])
         high = mid - 1;
      else
         low = mid + 1;
   }
}
```

- 관찰사항: 조건문의 의미
  - `(A)`: 아직 검사할 항목이 있고, `x`를 찾지 못했나?
- 관찰사항: 이 알고리즘으로 어떤 항목(key)을 찾기 위해서 배열 `S`에 있는 항목을 몇 개나 검색해야 하는가? 이 경우 `while`문을 수행할 때마다 검색 대상의 총 크기가 반 씩 감소하기 때문에 최악의 경우라도 $\lg n + 1$개만 검색하면 된다.

1.3.3 비교: 순차검색 대 이분검색

- 아래 표에서 볼 수 있듯이, 배열의 크기가 커질수록 두 알고리즘의 검색 횟수의 차이는 크게 벌어진다.

배열의 크기 n	순차검색 n	이분검색 lg n + 1	최악의 경우
128	128	8	
1,024	1,024	11	
1,048,576	1,048,576	21	
4,294,967,296	4,294,967,296	33	

1.4 보기: n번째 피보나찌 수 구하기

1.4.1 피보나찌(Fibonacci) 수열의 정의

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

예: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1264, \cdots$

1.4.2 되부름(recursive) 방법

- 문제: n번째 피보나찌 수를 구하시오.
- 입력: 양수 `n`
- 출력: n번째 피보나찌 수.

```
int fib (int n)
{
   if (n <= 1)
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```

- 관찰사항: 위의 알고리즘은 수행속도가 매우 느리다(inefficient). 왜냐하면, 같은 피보나찌 수를 중복하여 계산하기 때문이다. 예를 들면, `fib(5)`를 계산할 때 `fib(2)`를 3번 중복하여 계산한다. `fib(5)`를 계산하는데 `fib` 함수를 부르는 횟수를 계산하기 위해서는 교재 14쪽의 그림 1.2와 같이 되부름 나무(recursion tree)를 그려서 그 마디(node)의 개수를 세어보면 된다. $T(n)$을 `fib(n)`을 계산하기 위해서 `fib` 함수를 부르는 횟수, 즉, 되부름 나무 상의 마디의 수라고 하면, 다음과 같이 계산할 수 있다.

$$\begin{aligned} T(0) &= 1; \\ T(1) &= 1; \\ T(n) &= T(n-1) + T(n-2) + 1 && \text{for } n \geq 2 \\ &> 2 \times T(n-2) && \text{since } T(n-1) > T(n-2) \\ &> 2^2 \times T(n-4) \\ &> 2^3 \times T(n-6) \\ &\cdots \\ &> 2^{n/2} \times T(0) \\ &= 2^{n/2} \end{aligned}$$

- **정리 1.1:** 위의 알고리즘으로 구성한 되부름나무의 마디의 수를 $T(n)$이라고 하면, $n \geq 2$인 모든 n에 대해서 $T(n) > 2^{n/2}$이다.

  증명: n에 대해서 수학적 귀납법(mathematical induction)으로 증명

  귀납출발점(induction base):

  $$T(2) = T(1) + T(0) + 1 = 3 > 2 = 2^{2/2}$$

  $$T(3) = T(2) + T(1) + 1 = 5 > 2.83 \approx 2^{3/2}$$

  귀납가정(induction hypothesis): $2 \leq m < n$인 모든 m에 대해서 $T(m) > 2^{m/2}$라 가정.

귀납절차(induction step): $T(n) > 2^{n/2}$임을 보여야 한다.

$$\begin{aligned} T(n) &= T(n-1)+T(n-2)+1 \\ &> 2^{(n-1)/2}+2^{(n-2)/2}+1 \qquad \text{귀납가정에 의하여} \\ &> 2^{(n-2)/2}+2^{(n-2)/2} \\ &= 2\times 2^{(n/2)-1} \\ &= 2^{n/2} \end{aligned}$$

1.4.3 반복적(iteration) 방법

- 문제: n번째 피보나찌 수를 구하시오.
- 입력: 양수 n
- 출력: n번째 피보나찌 수.

```
int fib2 (int n)
{
   index i;
   int f[0..n];

   f[0] = 0;
   if (n >0) {
      f[1] = 1;
      for (i = 2; i <= n; i++)
         f[i] = f[i-1] + f[i-2];
   }
   return f[n];
}
```

- 관찰사항: 반복적 알고리즘은 수행속도가 빠르다(efficient). 왜냐하면, 되부름 알고리즘과는 달리 중복하여 계산하는 경우가 없기 때문이다. 계산하는 항의 총 갯수는 $T(n) = n+1$이 된다. 즉, `fib2(n)`을 계산하기 위해서는 `f[0]`부터 `f[n]`까지 한번씩만 계산하면 된다.
- 관찰사항: 교재 16쪽의 표 1.2에 위의 두 알고리즘의 수행시간을 비교해 놓았다.

제 2 절 알고리즘 분석: 시간복잡도 분석

- 알고리즘 분석(analysis of algorithms)이란? - 입력 크기에 따라서 기본동작이 몇번 수행되는지를 결정하는 절차 [= 시간복잡도(time complexity) 분석]

2.1 시간복잡도를 표현하는데 필요한 척도

- 기본 동작(basic operation): 비교문, 지정문(assignment statement) 등
- 입력 크기(input size, parameter): 배열의 크기, 리스트의 길이, 행렬에서 행(row)와 열(column)의 크기, 나무구조(tree)에서 마디(vertex)의 수와 가지(edge)의 수 등

2.2 분석 방법의 종류

- 모든 경우를 고려한 분석(every-case analysis)
  - ▷ 모든 경우를 고려하여 분석한 복잡도(complexity)는 다음의 성질을 가진다. "입력의 크기와는 관련이 있고(dependent), 입력의 값과는 관련이 없다(independent)."
  - ▷ 기본적인 동작이 수행되는 횟수는 입력의 값에 상관없이 항상 일정하다.
- 최악의 경우를 고려한 분석(worst-case analysis)
  - ▷ 최악의 경우를 고려하여 분석한 복잡도는 다음의 성질을 가진다. "입력의 크기와도 관련이 있고(dependent), 입력의 값과도 관련이 있다(dependent)."
  - ▷ 기본적인 동작이 수행되는 횟수가 최대인 경우를 택한다.
- 평균적인 경우를 고려한 분석(average-case analysis)
  - ▷ 평균의 경우를 고려하여 분석한 복잡도는 "모든 입력에 대해서 알고리즘이 기본 동작을 수행하는 횟수의 평균(기대치)이다."
  - ▷ 각 입력에 대해서 확률을 할당할 수도 있다.
  - ▷ 최악의 경우를 고려한 시간복잡도 분석보다 대체로 계산하기가 복잡하다.
- 최선의 경우를 고려한 분석(best-case analysis)
  - ▷ 최선의 경우를 고려하여 분석한 복잡도는 모든 입력 중에서 알고리즘이 최소로 기본동작을 수행하는 횟수이다.

2.3 보기: 배열(Array) 덧셈

2.3.1 알고리즘

- 문제: 크기가 n인 배열 S의 모든 수를 더하시오.
- 입력: 양수 n, 배열 S[1..n]
- 출력: 배열 S의 모든 수의 합

```
number sum (int n, const number S[])
{
   index i;
   number result;

   result = 0;
   for (i = 1; i <= n; i++)
      result = result + S[i];
   return result;
}
```

2.3.2 시간복잡도 분석 I

- 기본동작: 덧셈
- 입력의 크기: 배열의 크기 n
- 모든 경우를 고려한 시간복잡도 분석: 배열에 어떤 수가 있는지에 상관없이 for-맴돌이(loop)가 n번 반복된다. 그리고 각 맴돌이 마다 덧셈이 1회 수행된다. 따라서 n에 대해서 덧셈이 수행되는 총 횟수는 $T(n) = n$이다.

2.3.3 시간복잡도 분석 II

- 기본동작: 지정문 - for-맴돌이의 첨자 대입문 포함
- 입력의 크기: 배열의 크기 n
- 모든 경우를 고려한 시간복잡도 분석: 배열에 어떤 수가 있는지에 상관없이 for-맴돌이가 n번 반복된다. 따라서 지정문이 $T(n) = n+n+1$번 수행된다.

2.3.4 관찰사항

- 위에서 기본동작을 다르게 봄으로서 시간복잡도가 다르게 나왔다. 그러나 사실은 둘 다 같은 복잡도 카테고리에 속한다고 봐도 된다. 자세한 사항은 뒤에 차수(order)를 배울때 다루게 된다.

2.4 보기: 교환정렬(Exchange Sort)

2.4.1 알고리즘

- 문제: 비내림차순(nondecreasing order)으로 n개의 키(key)를 정렬
- 입력: 양수 n, 배열 S[1..n]
- 출력: 비내림차순으로 정렬된 배열 S

```
void exchangesort (int n, keytype S[])
{
   index i,j;

   for (i = 1; i <= n-1; i++)
      for (j = i+1; j <= n; j++)
         if (S[j] < S[i])
            exchange S[i] and S[j];
}
```

2.4.2 시간복잡도 분석 I

- 기본동작: 조건문 - S[j]와 S[i]를 비교하는 동작
- 입력의 크기: 정렬할 항목의 수 n
- 모든 경우를 고려한 시간복잡도 분석: j-맴돌이가 수행될 때마다 어떠한 경우에도 조건문이 1번씩 수행된다. 따라서 조건문이 수행되는 총 횟수는 다음과 같이 구할 수 있다.

$i = 1$ 일때, j-맴돌이 $n-1$번 수행
$i = 2$ 일때, j-맴돌이 $n-2$번 수행
$i = 3$ 일때, j-맴돌이 $n-3$번 수행
$\cdots$
$i = n-1$ 일때, j-맴돌이 1번 수행
따라서 $T(n) = (n-1)+(n-2)+(n-3)+\cdots+1 = (n-1)n/2$

2.4.3 시간복잡도 분석 II

- 기본동작: 교환하는 동작 - `exchange S[i] and S[j]`
- 입력의 크기: 정렬할 항목의 수 n
- 최악의 경우를 고려한 시간복잡도 분석: 조건문의 결과에 따라서 교환하는 동작이 수행될 수도 있고, 그렇지 않을 수도 있다. 따라서 최악의 경우, 즉, 조건문이 항상 참값을 가지는 경우(입력 배열이 역순으로 정렬되어 있는 경우)를 고려해 보면 위의 분석과 마찬가지로 $T(n) = (n-1)n/2$이 된다.

2.5 보기: 순차검색

- 순차검색(sequential search) 알고리즘의 경우 입력 배열의 값에 따라서 검색하는 횟수가 다르므로, 모든 경우를 고려한 해석은 불가능하다.

2.5.1 시간복잡도 분석 I

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 최악의 경우를 고려한 시간복잡도 분석: x가 배열의 마지막 항목이거나 배열 안에 없을 경우에는 기본동작이 n번 수행된다. 따라서 $W(n) = n$

2.5.2 시간복잡도 분석 II

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 평균의 경우를 고려한 시간복잡도 분석: 배열의 항목이 모두 다르다고 가정하고, 다음과 같이 시간복잡도를 계산한다.
  - ▷ 경우 1: (x가 배열 S 안에 있는 경우)
    - ⋆ $1 \le k \le n$에 대해서, x가 배열의 k번째 있을 확률 $= 1/n$
    - ⋆ 만약 x가 배열의 k번째 있다면, k를 찾기 위해서 수행하는 기본 동작의 횟수 $= k$
    - ⋆ 따라서,

$$\begin{aligned} A(n) &= \sum_{k=1}^{n} (k \times \frac{1}{n}) \\ &= \frac{1}{n} \times \sum_{k=1}^{n} k \\ &= \frac{1}{n} \times \frac{n(n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

  - ▷ 경우 2: (x가 배열 S 안에 없는 경우)
    - ⋆ x가 배열 S 안에 있을 확률을 p라고 하면,
      - · x가 배열의 k번째 있을 확률 $= p/n$
      - · x가 배열의 k번째 없을 확률 $= 1-p$
    - ⋆ 따라서,

$$\begin{aligned} A(n) &= \sum_{k=1}^{n} (k \times \frac{p}{n}) + n(1-p) \\ &= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) \\ &= n(1 - \frac{p}{2}) + \frac{p}{2} \end{aligned}$$

2.5.3 시간복잡도 분석 III

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 최선의 경우를 고려한 시간복잡도 분석: x가 $S[1]$일때, 입력의 크기에 상관없이 기본동작이 한번만 수행된다. 따라서 $B(n) = 1$

2.5.4 관찰사항

- 최악의 경우보다 평균의 경우의 분석이 직관적으로 더 이치에 맞아보인다. 그러나 위의 경우 두 분석의 결과가 모두 같은 복잡도 카테고리 $\Theta(n)$에 속한다. 사실 일반적으로 거의 대부분의 경우 두 분석 방법에 따른 결과의 차이는 없다. (물론 예외는 있다.) 따라서 계산하기가 훨씬 간단한 최악의 경우를 이용하는 것이 일반적이다. 그리고 최선의 경우를 선택하는 것은 비현실적이다.

2.6 정확도 분석(Analysis of Correctness)

1. 알고리즘이 원래 의도한 대로 실제로 수행이 되는지를 증명하는 절차
2. 정확한 알고리즘이란 무었인가? - "어떠한 입력에 대해서도 맞는 답을 출력하면서 멈추는 알고리즘"
3. 정확하지 않은 알고리즘이란? - "어떤 입력에 대해서 멈추지 않거나 또는 틀린 답을 출력하면서 멈추는 알고리즘"

제 3 절 차수

- 차수(order)는 알고리즘의 복잡도(complexity)를 표시하기 위하여 쓰는 일종의 표기법이다.
- 예를 들면, $\Theta(n^2)$은 2차함수(quadratic function)로 분류되는 모든 복잡도 함수의 집합을 나타낸다. 다시말해서, $\Theta(n^2)$에 속해있는 모든 복잡도 함수는 차수가 n^2이라고 대표해서 부를 수 있다. 따라서 $5n^2$, $5n^2+100$, $0.1n^2+n+100$ 모두 차수가 n^2이다. (교재 27쪽의 표 1.3 참조)
- 이때 낮은 차수의 항(low-order term)은 무시해도 상관없다. 왜냐하면 가장 높은 차수의 항이 전체 항의 성질을 지배하기 때문이다.

3.1 복잡도 카테고리

- $\Theta(\lg n)$
- $\Theta(n)$: 선형(linear)
- $\Theta(n \lg n)$
- $\Theta(n^2)$: 2차(quadratic)
- $\Theta(n^3)$: 3차(cubic)
- $\Theta(2^n)$: 지수형(exponential)
- $\Theta(n!)$

1. 교재 28쪽의 그림 1.3과 29쪽의 표 1.4를 보시오.
2. 같은 카테고리에 속한 어떤 함수도 사실은 그 카테고리를 대표할 수 있다. 그러나 편의상 가장 간단히 표시할 수 있는 함수로 그 카테고리를 표현하는 것이 통례이다.

3.2 큰(Big)O 표기법

- **정의 2: 점근적 상한(Asymptotic Upper Bound)**
  주어진 복잡도 함수 $f(n)$에 대해서 $g(n) \in O(f(n))$ 이면 다음을 만족한다: $n \geq N$인 모든 정수 n에 대해서 $g(n) \leq c \times f(n)$이 성립하는 실수 $c > 0$와 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(a)를 보시오.)
- $g(n) \in O(f(n))$은 "$g(n)$은 $f(n)$의 큰 오(big O)"라고 부른다.
- 어떤 함수 $g(n)$이 $O(n^2)$에 속한다는 말은, 그 함수는 궁극에 가서는 (즉, 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차함수 cn^2의 값보다는 **작은** 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 **아래**에 위치) 다시말해서, 그 함수 $g(n)$은 어떤 2차함수 cn^2 보다는 궁극적으로 **좋다**고 (기울기가 **낮다**고) 말할 수 있다.
- 어떤 알고리즘의 시간복잡도가 $O(f(n))$이라면, 입력의 크기 n에 대해서 이 알고리즘의 수행시간은 **아무리 늦어도** $f(n)$은 된다. ($f(n)$이 **상한**이다.) 다시말하면, 이 알고리즘의 수행시간은 $f(n)$보다 절대로 더 느릴 수는 없다는 말이다.
- 보기 1.3: $n^2+10n \in O(n^2)$임을 보이시오.
  (1) $n \geq 10$인 모든 정수 n에 대해서 $n^2+10n \leq 2n^2$이 성립한다. 그러므로, $c=2$와 $N=10$을 선택하면, "큰 O"의 정의에 의해서 $n^2+10n \in O(n^2)$이라고 결론지을 수 있다. (교재 30쪽의 그림 1.5를 보시오.)
  (2) $n \geq 1$인 모든 정수 n에 대해서 $n^2+10n \leq n^2+10n^2 = 11n^2$이 성립한다. 그러므로, $c=11$와 $N=1$을 선택하면, "큰 O"의 정의에 의해서 $n^2+10n \in O(n^2)$이라고 결론지을 수 있다.
- 보기 1.4: $5n^2 \in O(n^2)$임을 보이시오.
  $c=5$와 $N=0$을 선택하면, $n \geq 0$인 모든 정수 n에 대해서 $5n^2 \leq 5n^2$이 성립한다.
- 보기 1.5: $T(n) = \frac{n(n-1)}{2}$은 어떻게 될까?
  $n \geq 0$인 모든 정수 n에 대해서 $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$이 성립한다. 그러므로, $c=\frac{1}{2}$와 $N=0$을 선택하면, $T(n) \in O(n^2)$이라고 결론지을 수 있다.
- 보기 1.6: $n^2 \in O(n^2+10n)$임을 보이시오.
  $n \geq 0$인 모든 정수 n에 대해서, $n^2 \leq 1 \times (n^2+10n)$이 성립한다. 그러므로, $c=1$와 $N=0$을 선택하면, $n^2 \in O(n^2+10n)$이라고 결론지을 수 있다.
- 보기 1.7: $n \in O(n^2)$임을 보이시오.
  $n \geq 1$인 모든 정수 n에 대해서, $n \leq 1 \times n^2$이 성립한다. 그러므로, $c=1$와 $N=1$을 선택하면, $n \in O(n^2)$이라고 결론지을 수 있다.
- 보기 1.8: $n^3 \in O(n^2)$이 아님을 보이시오.
  $n \geq N$인 모든 n에 대해서 $n^3 \leq c \times n^2$이 성립하는 c와 N 값은 존재하지 않는다. 즉, 양변을 n^2으로 나누면, $n \leq c$가 되는데 c를 아무리 크게 잡더라도 그 보다 더 큰 n이 존재한다.
- 참고: 교재 32쪽의 그림 1.6(a)를 보시오.

3.3 Ω 표기법

- **정의 1.3: 점근적 하한(Asymptotic Lower Bound)**
  주어진 복잡도 함수 $f(n)$에 대해서 $g(n) \in \Omega(f(n))$이면 다음을 만족한다: $n \geq N$인 모든 정수 n에 대해서 $g(n) \geq c \times f(n)$이 성립하는 실수 $c > 0$와 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(b)를 보시오.)
- $g(n) \in \Omega(f(n))$은 "$g(n)$은 $f(n)$의 오메가(omega)"라고 부른다.
- 어떤 함수 $g(n)$이 $\Omega(n^2)$에 속한다는 말은, 그 함수는 궁극에 가서는 (즉, 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차함수 cn^2의 값보다는 **큰** 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 **위**에 위치) 다시말해서, 그 함수 $g(n)$은 어떤 2차함수 cn^2 보다는 궁극적으로 **나쁘다**고 (기울기가 **높다**고) 말할 수 있다.
- 어떤 알고리즘의 시간복잡도가 $\Omega(f(n))$이라면, 입력의 크기 n에 대해서 이 알고리즘의 수행시간은 **아무리 빨라도** $f(n)$밖에 되지 않는다. ($f(n)$이 **하한**이다.) 다시말하면, 이 알고리즘의 수행시간은 $f(n)$보다 절대로 더 빠를 수는 없다는 말이다.
- 보기 1.9: $n^2 + 10n \in \Omega(n^2)$임을 보이시오.
  $n \geq 0$인 모든 정수 n에 대해서 $n^2 + 10n \geq n^2$이 성립한다. 그러므로, $c = 1$와 $N = 0$을 선택하면, $n^2 + 10n \in \Omega(n^2)$이라고 결론지을 수 있다.
- 보기 1.10: $5n^2 \in \Omega(n^2)$임을 보이시오.
  $n \geq 0$인 모든 정수 n에 대해서, $5n^2 \geq 1 \times n^2$이 성립한다. 그러므로, $c = 1$와 $N = 0$을 선택하면, $5n^2 \in \Omega(n^2)$이라고 결론지을 수 있다.
- 보기 1.11: $T(n) = \frac{n(n-1)}{2}$은 어떻게 될까?
  $n \geq 2$인 모든 n에 대해서 $n - 1 \geq \frac{n}{2}$이 성립한다. 그러므로, $n \geq 2$인 모든 n에 대해서 $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$이 성립한다. 따라서 $c = \frac{1}{4}$과 $N = 2$를 선택하면, $T(n) \in \Omega(n^2)$이라고 결론지을 수 있다.
- 보기 1.12: $n^3 \in \Omega(n^2)$임을 보이시오.
  $n \geq 1$인 모든 정수 n에 대해서, $n^3 \geq 1 \times n^2$이 성립한다. 그러므로, $c = 1$와 $N = 1$을 선택하면, $n^3 \in \Omega(n^2)$이라고 결론지을 수 있다.
- 보기 1.13: n이 $\Omega(n^2)$이 아님을 보이시오.
  모순유도에 의한 증명(Proof by contradiction): $n \in \Omega(n^2)$이라고 가정. 그러면 $n \geq N$인 모든 정수 n에 대해서, $n \geq c \times n^2$이 성립하는 실수 $c > 0$, 그리고 음이 아닌 정수 N이 존재한다. 위의 부등식의 양변을 cn으로 나누면 $\frac{1}{c} \geq n$가 된다. 그러나 이 부등식은 절대로 성립할 수 없다. 따라서 위의 가정은 모순이다.
- 참고: 교재 32쪽의 그림 1.6(b)를 보시오.

3.4 Θ 표기법

- **정의 1.4: (Asymptotic Tight Bound)**
  주어진 복잡도 함수 $f(n)$에 대해서 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$. 다시말하면, $\Theta(f(n))$은 다음을 만족하는 복잡도함수 $g(n)$의 집합이다: $n \geq N$인 모든 정수 n에 대해서 $c \times f(n) \leq g(n) \leq d \times f(n)$이 성립하는 실수 $c > 0$와 $d > 0$, 그리고 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(c)를 보시오.)
- 교재 32쪽의 그림 1.6(c)를 보시오.
- 참고: $g(n) \in \Theta(f(n))$은 "$g(n)$은 $f(n)$의 차수(order)"라고 부른다.
- 보기 1.14: $T(n) = \frac{n(n-1)}{2}$은 $O(n^2)$이면서 $\Omega(n^2)$이다. 따라서 $T(n) = \Theta(n^2)$

3.5 작은(Small) o 표기법

- 작은 o는 복잡도 함수 끼리의 관계를 나타내기 위한 표기법이다.
- **정의 1.5: 작은 o**
  주어진 복잡도 함수 $f(n)$에 대해서 $o(f(n))$은 다음을 만족하는 모든 복잡도 함수 $g(n)$의 집합이다: 모든 실수 $c > 0$에 대해서 $g(n) \leq c \times f(n)$ (여기서 $n \geq N$인 모든 n에 대해서)이 성립하는 음이 아닌 정수 N이 존재한다.
- 참고: $g(n) \in o(f(n))$은 "$g(n)$은 $f(n)$의 작은 오(o)"라고 부른다.
- 참고: 큰 O와의 차이점
  - ▷ 큰 O - 실수 $c > 0$ 중에서 하나만 성립하어도 됨
  - ▷ 작은 o - 모든 실수 $c > 0$에 대해서 성립하여야 함
- 참고: $g(n) \in o(f(n))$은 쉽게 설명하자면 $g(n)$이 궁극적으로 $f(n)$보다 훨씬 낫다(좋다)는 의미이다. 실례를 보자.
- 보기 1.15: $n \in o(n^2)$임을 보이시오.
  증명: $c > 0$이라고 하자. $n \geq N$인 모든 n에 대해서 $n \leq cn^2$이 성립하는 N을 찾아야 한다. 이 부등식의 양변을 cn으로 나누면 $\frac{1}{c} \leq n$을 얻는다. 따라서 $N \geq \frac{1}{c}$가 되는 어떤 N을 찾으면 된다. 여기서 N의 값은 c에 의해 좌우된다. 예를 들어 만약 $c = 0.0001$이라고 하면, N의 값은 최소한 10,000이 되어야 한다. 즉, $n \geq 10,000$인 모든 n에 대해서 $n \leq 0.0001n^2$이 성립한다.
- 보기 1.16: n이 $o(5n)$이 아님을 보이시오.
  모순 유도에 의한 증명: $c = \frac{1}{6}$이라고 하자. $n \in o(5n)$이라고 가정하면, $n \geq N$인 모든 정수 n에 대해서, $n \leq \frac{1}{6} \times 5n = \frac{5}{6}n$이 성립하는 음이 아닌 정수 N이 존재해야 한다. 그러나 그런 N은 절대로 있을 수 없다. 따라서 위의 가정은 모순이다.

- 보기 1.17: n^2이 $o(n)$이 아님을 보이시오.
  모순 유도에 의한 증명: (숙제)
- **정리 1.2: 작은 o와 다른 표기법과의 관계**
  $g(n) \in o(f(n))$이라면, $g(n) \in O(f(n)) - \Omega(f(n))$이 성립한다. 즉, $g(n)$은 $O(f(n))$이기는 하지만, $\Omega(f(n))$은 아니다.
  - ▷ $g(n)$은 $O(f(n))$임을 증명(직접증명)
    $g(n) \in o(f(n))$이므로 모든 실수 $c > 0$에 대해서 $g(n) \le c \times f(n)$이 $n \ge N$인 모든 n에 대해서 성립하는 N이 존재한다. 여기서 임의의 c를 선택하여도 이 부등식은 성립하므로 $g(n)$은 당연히 $O(f(n))$이다.
  - ▷ $g(n)$은 $\Omega(f(n))$이 아님을 증명(모순유도에 의한 증명)
    $g(n) \in \Omega(f(n))$이라고 가정하자. 그러면 $n \ge N_1$인 모든 n에 대해서 $g(n) \ge c \times f(n)$을 만족시키는 실수 $c > 0$과 음이 아닌 정수 N_1이 반드시 존재한다. 그러나 $g(n) \in o(f(n))$이기 때문에, $n \ge N_2$인 모든 n에 대해서 $g(n) \le \frac{c}{2} \times f(n)$을 만족시키는 N_2가 존재한다. 따라서,
    $$cf(n) \le g(n) \le \frac{c}{2} f(n)$$
    각 항을 $f(n)$으로 나누면,
    $$c \le \frac{g(n)}{f(n)} \le \frac{c}{2}$$
    이 두 부등식에 의하면 N_1과 N_2보다 큰 모든 n에 대해서 성립해야 하는데, 이는 실제로 불가능하다. 모순유도 성공!
- 참고: 일반적으로 $O(f(n)) - \Omega(f(n))$인 복잡도 함수는 $o(f(n))$이기도 하다. 그러나 항상 그렇지는 않다. (자세한 사항은 교재 36쪽 참조)

3.6 차수의 주요 성질

1. $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$
3. $b > 1$이고 $a > 1$이면, $\log_a n \in \Theta(\log_b n)$은 항상 성립. 다시 말하면 로그(logarithm) 복잡도 함수는 모두 같은 카테고리에 속한다. 따라서 통상 $\Theta(\lg n)$으로 표시한다.
4. $b > a > 0$이면, $a^n \in o(b^n)$. 다시말하면 지수형(exponential) 복잡도 함수가 모두 같은 카테고리 안에 있는 것은 아니다.
5. $a > 0$인 모든 a에 대해서, $a^n \in o(n!)$. 다시말하면, $n!$은 어떤 지수형 복잡도 함수보다도 나쁘다.
6. 복잡도 함수를 다음 순으로 나열해 보자.
   $$\Theta(\lg n), \Theta(n), \Theta(n \lg n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$
   여기서 $k > j > 2$이고 $b > a > 1$이다. 복잡도함수 $g(n)$이 $f(n)$을 포함한 카테고리의 왼쪽에 위치한다고 하면, $g(n) \in o(f(n))$.
7. $c \ge 0$, $d \ge 0$, $g(n) \in O(f(n))$, 그리고 $h(n) \in \Theta(f(n))$이면, $c \times g(n) + d \times h(n) \in \Theta(f(n))$.

3.7 극한(limit)를 이용하여 차수를 구하는 방법

- **정리 1.3:**
  $$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \begin{cases} c > 0 & \text{이면} \quad g(n) \in \Theta(f(n)) \\ 0 & \text{이면} \quad g(n) \in o(f(n)) \\ \infty & \text{이면} \quad f(n) \in o(g(n)) \end{cases}$$
- 보기 1.16: 다음이 성립함을 정리 3을 이용하여 보이시오.
  - ▷ $\frac{n^2}{2} \in o(n^3)$
    $$\lim_{n\to\infty} \frac{n^2/2}{n^3} = \lim_{n\to\infty} \frac{1}{2n} = 0$$
  - ▷ $b > a > 0$일때, $a^n \in o(b^n)$
    $$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n = 0 \text{ 왜냐하면, } 0 < \frac{a}{b} < 1$$
- **정리 1.4: 로피탈(L'Hopital)의 법칙**
  $$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty \text{ 이면}$$
  $$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{g'(n)}{f'(n)} \text{ 이다.}$$
- 보기 1.17: 다음이 성립함을 정리 3과 4를 이용하여 보이시오.
  - ▷ $\lg n \in o(n)$
    $$\lim_{n\to\infty} \frac{\lg n}{n} = \lim_{n\to\infty} \frac{\frac{1}{n \ln 2}}{1} = 0$$
  - ▷ $\log_a n \in \Theta(\log_b n)$
    $$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \frac{\frac{1}{n \ln a}}{\frac{1}{n \ln b}} = \frac{\log b}{\log a} > 0$$

제 2 장

분할정복

분할정복(Divide-and-Conquer)식 설계 전략

- 분할(Divide): 해결하기 쉽도록 문제를 여러 개의 작은 부분으로 나눈다.
- 정복(Conquer): 나눈 작은 문제를 각각 해결한다.
- 통합(Combine): (필요하다면) 해결된 해답을 모은다.

이러한 문제 해결 방법을 **하향식(top-down)** 접근방법이라고 한다.

제 1 절 이분검색(Binary Search)

1.1 알고리즘: 되부름(recursive) 방법

- 문제: 크기가 n인 정렬된 배열 S에 x가 있는지를 결정하시오.
- 입력: 자연수 n, 비내림차순으로 정렬된 배열 S[1..n], 찾고자 하는 항목 x
- 출력: locationout - x가 S의 어디에 있는지의 위치. 만약 x가 S에 없다면 0
- 설계전략:
  - x가 배열의 중간에 위치하고 있는 항목과 같으면, "빙고", 찾았다! 그렇지 않으면:
  - 분할: 배열을 반으로 나누어서 x가 중앙에 위치한 항목보다 작으면 왼쪽에 위치한 배열 반쪽을 선택하고, 그렇지 않으면 오른쪽에 위치한 배열 반쪽을 선택한다.
  - 정복: 선택된 반쪽 배열에서 x를 찾는다.
  - 통합: (필요없음)
- 보기: 10 12 13 14 18 20 25 27 30 45 47에서 18을 찾으시오.
- 알고리즘:

```
index location (index low, index high)
{
   index mid;

   if (low > high)
      return 0;                          // 찾지 못했음
   else {
      mid = (low + high) / 2;            // 정수 나눗셈 (나머지 버림)
      if (x == S[mid])
         return mid;                     // 찾았음
      else if (x < S[mid])
         return location(low,mid-1);     // 왼쪽 배열 반쪽을 선택함
      else
         return location(mid+1,high);    // 오른쪽 배열 반쪽을 선택함
   }
}

...

locationout = location(1,n);
...
```

1.2 관찰사항

- 왜 지역함수 `location`을 정의하여 알고리즘을 둘로 나누었을까?
  입력 매개변수인 `n, S, x`는 알고리즘 수행 중 변하지 않는 값이다. 따라서 함수를 되부를 때 마다 이러한 변하지 않는 매개변수를 가지고 다니는 것은 극심한 낭비이다.
- 되부름 알고리즘(recursive algorithm)에서 모든 되부름이 알고리즘의 마지막(꼬리) 부분에서 이루어질 때 – 꼬리 되부름(tail recursion)이라고 함 – 그 알고리즘은 맴돌이 알고리즘(iterative algorithm)으로 변환하기가 수월하다. 일반적으로 되부름 알고리즘은 되부를 때마다 그 당시의 상태를 활성레코드(activation records) 스택에 저장해 놓아야 하는 반면, 맴돌이 알고리즘은 그럴 필요가 없기 때문에 일반적으로 더 효율적이다(빠르다). 그렇다고 맴돌이 알고리즘의 계산복잡도가 되부름 알고리즘보다 좋다는 의미는 아니다. 맴돌이 알고리즘이 상수적(constant factor)으로만 좋다(빠르다)는 말이다. 최근에 설계, 구현된 ML이나 Scheme 같은 함수형 언어는 컴파일러가 자동으로 되부름 프로그램을 맴돌이 프로그램으로 바꾸어준다.

1.3 최악의 경우 시간복잡도 분석

- 기본동작: `x`와 `S[mid]`의 비교
- 입력의 크기: 배열의 크기 `n` (= `high` - `low` + 1)
- 알고리즘을 살펴보면 기본동작으로 설정한 조건문을 `while` 맴돌이 내부에서 2번 수행하지만, 사실상 비교는 한번 이루어진다고 봐도 된다. 그 이유는: (1) 어셈블리 언어로는 하나의 조건 명령으로 충분히 구현할 수 있기 때문이기도 하고; (2) `x`를 찾기 전까지는 항상 2개의 조건문을 수행하므로 하나로 묶어서 한 단위로 취급을 해도 되기 때문이기도 하다. 이와같이 기본동작은 최대한으로 효율적으로(빠르게) 구현된다고 일반적으로 가정하여, 1단위로 취급을 해도 된다.

1.3.1 경우 1: 검색하게 될 반쪽 배열의 크기가 항상 정확하게 $\frac{n}{2}$이 되는 경우

- 시간복잡도를 나타내 주는 재현방정식(recurrence)은 다음과 같다.

$$\begin{array}{lcll} W(n) & = & W\left(\frac{n}{2}\right)+1 & n>1 \text{ 이고, } n=2^k(k\geq 1) \\ W(1) & = & 1 & \end{array}$$

  이 식의 해는 다음과 같이 구할 수 있다.

$$\begin{array}{rcl} W(1) & = & 1 \\ W(2) & = & W(1)+1 = 2 \\ W(4) & = & W(2)+1 = 3 \\ W(8) & = & W(4)+1 = 4 \\ W(16) & = & W(8)+1 = 5 \\ & \cdots & \\ W(2^k) & = & k+1 \\ & \cdots & \\ W(n) & = & \lg n+1 \end{array}$$

- 이 해가 과연 맞는지 확인하기 위하여 증명해 보자.
- 증명: 수학적귀납법
  귀납출발점: $n=1$이면, $W(1)=1=\lg 1+1$.
  귀납가정: 2의 거듭제곱(power)인 양의 정수 n에 대해서, $W(n)=\lg n+1$라고 가정한다.
  귀납단계: $W(2n)=\lg(2n)+1$임을 보이면 된다. 회귀식을 사용하면,

$$\begin{array}{lcll} W(2n) & = & W(n)+1 & \text{재현방정식에 의해서} \\ & = & \lg n+1+1 & \text{귀납가정에 의해서} \\ & = & \lg n+\lg 2+1 & \\ & = & \lg(2n)+1 & \end{array}$$

1.3.2 경우 2: 일반적인 경우 - 반쪽 배열의 크기는 $\lfloor \frac{n}{2} \rfloor$이 됨

- $\lfloor y \rfloor$란 y보다 작거나 같은 최대 정수를 나타낸다고 할때, n에 대해서 가운데 첨자는 $mid = \lfloor \frac{1+n}{2} \rfloor$이 되는데, 이 때 각 부분배열의 크기는 다음과 같다.

n	왼쪽 부분배열의 크기	mid	오른쪽 부분배열의 크기
짝수	$n/2-1$	1	$n/2$
홀수	$(n-1)/2$	1	$(n-1)/2$

  위의 표에 의하면 알고리즘이 다음 단계에 찾아야 할 항목의 갯수는 기껏해야 $\lfloor \frac{n}{2} \rfloor$개가 된다. 따라서 다음과 같은 재현방정식으로 표현할 수 있다.

$$\begin{array}{lcll} W(n) & = & 1+W\left(\lfloor \frac{n}{2} \rfloor\right) & n>1\text{일때} \\ W(1) & = & 1 & \end{array}$$

- 이 재현방정식의 해가 $W(n)=\lfloor \lg n \rfloor+1$가 됨을 n에 대한 수학적귀납법으로 증명한다.

- 증명: 수학적귀납법
  귀납출발점: $n = 1$이면, 다음이 성립한다.

$$\lfloor \lg n \rfloor + 1 = \lfloor \lg 1 \rfloor + 1 = 0 + 1 = 1 = W(1)$$

  귀납가정: $n > 1$이고, $1 \le k < n$인 모든 k에 대해서, $W(k) = \lfloor \lg k \rfloor + 1$가 성립한다고 가정한다.
  귀납단계: n이 짝수이면 (즉, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$),

$$\begin{array}{rcll} W(n) & = & 1 + W\left(\lfloor \frac{n}{2} \rfloor\right) & \text{재현방정식에 의해서} \\ & = & 1 + \lfloor \lg \lfloor \frac{n}{2} \rfloor \rfloor + 1 & \text{귀납가정에 의해서} \\ & = & 2 + \lfloor \lg \lfloor \frac{n}{2} \rfloor \rfloor & \\ & = & 2 + \lfloor \lg \frac{n}{2} \rfloor & n\text{이 짝수이므로} \\ & = & 2 + \lfloor \lg n - 1 \rfloor & \\ & = & 2 + \lfloor \lg n \rfloor - 1 & \\ & = & 1 + \lfloor \lg n \rfloor & \end{array}$$

  n이 홀수이면 (즉, $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$),

$$\begin{array}{rcll} W(n) & = & 1 + W\left(\lfloor \frac{n}{2} \rfloor\right) & \text{재현방정식에 의해서} \\ & = & 1 + \lfloor \lg \lfloor \frac{n}{2} \rfloor \rfloor + 1 & \text{귀납가정에 의해서} \\ & = & 2 + \lfloor \lg \lfloor \frac{n}{2} \rfloor \rfloor & \\ & = & 2 + \lfloor \lg \frac{n-1}{2} \rfloor & n\text{이 홀수이므로} \\ & = & 2 + \lfloor \lg (n-1) - 1 \rfloor & \\ & = & 2 + \lfloor \lg (n-1) \rfloor - 1 & \\ & = & 1 + \lfloor \lg (n-1) \rfloor & \\ & = & 1 + \lfloor \lg n \rfloor & n\text{이 홀수이므로} \end{array}$$

  따라서, $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$.

제 2 절 합병정렬

2.1 알고리즘

2.1.1 합병정렬(Mergesort)

- 문제: n개의 정수를 비내림차순으로 정렬하시오.
- 입력: 정수 n, 크기가 n인 배열 S[1..n]
- 출력: 비내림차순으로 정렬된 배열 S[1..n]
- 보기: 27, 10, 12, 20, 25, 13, 15, 22
- 알고리즘:

```
void mergesort (int n, keytype S[])
{
   const int h = n / 2, m = n - h;
   keytype U[1..h], V[1..m];

   if (n > 1) {
      copy S[1] through S[h] to U[1] through U[h];
      copy S[h+1] through S[n] to V[1] through V[m];
      mergesort(h,U);
      mergesort(m,V);
      merge(h,m,U,V,S);
   }
}
```

2.1.2 합병(Merge)

- 문제: 두개의 정렬된 배열을 하나의 정렬된 배열로 합병하시오.
- 입력: (1) 양의 정수 h, m, (2) 정렬된 배열 U[1..h], V[1..m]
- 출력: U와 V에 있는 키들을 하나의 배열에 정렬한 S[1..h+m]
- 알고리즘:

```
void merge(int h,
           int m,
           const keytype U[],
           const keytype V[],
           keytype S[])
```

```
{
   index i,j,k;

   i = 1; j = 1; k = 1;
   while (i <= h && j <= m) {
      if (U[i] < V[j]) {
         S[k] = U[i];
         i++;
      }
      else {
         S[k] = V[j];
         j++;
      }
      k++;
   }
   if (i > h)
      copy V[j] through V[m] to S[k] through S[h+m];
   else
      copy U[i] through U[h] to S[k] through S[h+m];
}
```

2.2 시간복잡도 분석

2.2.1 합병 알고리즘의 최악의 경우 시간복잡도 분석

- 기본동작: `U[i]`와 `V[j]`의 비교
- 입력의 크기: 2개의 입력 배열에 각각 들어있는 항목의 갯수: h와 m
- 분석: $i = h$이고, $j = m-1$인 상태로 맴돌이(loop)에서 빠져나가는 때가 최악의 경우로서(V에 있는 처음 $m-1$개의 항목이 S의 앞부분에 위치하고, U에 있는 h개의 모든 항목이 그 뒤에 위치하는 경우), 이 때 기본동작의 실행 횟수는 $h+m-1$이다. 따라서, 따라서 최악의 경우 합병하는 시간복잡도는 $W(h,m) = h+m-1$.

2.2.2 합병정렬 알고리즘의 최악의 경우 시간복잡도 분석

- 기본동작: 합병 알고리즘 `merge`에서 발생하는 비교
- 입력의 크기: 배열 S에 들어있는 항목의 갯수 n
- 분석: 최악의 경우 수행시간은 $W(h,m) = W(h)+W(m)+h+m-1$이 된다. 여기서 $W(h)$는 U를 정렬하는데 걸리는 시간, $W(m)$은 V를 정렬하는데 걸리는 시간, 그리고 $h+m-1$은 합병하는데 걸리는 시간이다. 정수 n을 $2^k (k \geq 1)$이라고 가정하면, $h = \frac{n}{2}, m = \frac{n}{2}$이 된다. 따라서 최악의 경우 재현방정식은:

$$\begin{array}{lcll} W(n) & = & 2W\left(\frac{n}{2}\right)+n-1 & n>1 \text{ 이고, } n=2^k (k \geq 1) \\ W(1) & = & 0 & \text{왜냐하면 합병이 전혀 이루어지지 않으므로} \end{array}$$

  이 회귀식의 해는 아래의 도사정리의 2번을 적용하면, $W(n) = \Theta(n \lg n)$이 된다.

- n이 2의 거듭제곱(power)의 형태가 아닌 경우의 재현 방정식은 다음과 같이 된다.

$$\begin{array}{lcll} W(n) & = & W\left(\lfloor \frac{n}{2} \rfloor\right)+W\left(\lceil \frac{n}{2} \rceil\right)+n-1 & n>1 \text{ 일때} \\ W(1) & = & 0 & \end{array}$$

  그러나 이 재현방정식의 정확한 해를 구하기는 복잡하다. 그러나, 앞의 이분검색 알고리즘의 분석에서도 보았듯이, $n = 2^k$라고 가정해서 해를 구하면, 이 재현방정식의 해와 같은 카테고리의 시간복잡도를 얻게 된다. 따라서 앞으로 이와 비슷한 재현방정식의 해를 구할때, $n = 2^k$라고 가정해서 구해도 점근적으로는 같은 해를 얻게 된다.

2.3 공간복잡도 분석

- 입력을 저장하는데 필요한 만큼 이상의 저장장소를 사용하지 않고 정렬하는 알고리즘을 **제자리정렬(in-place sort) 알고리즘**이라고 한다. 위에서 본 합병정렬 알고리즘은 제자리정렬 알고리즘이 아니다. 왜냐하면 입력인 배열 S이외에 U와 V를 추가로 만들어서 사용하기 때문이다.
- 그러면 얼마만큼의 추가적인 저정장소가 필요할까? `mergesort`를 되부를 때마다 크기가 S의 반이 되는 U와 V가 추가적으로 필요하다. `merge` 알고리즘에서는 U와 V가 주소로 전달이 되어 그냥 사용되므로 추가적인 저장장소를 만들지 않는다. 따라서 `mergesort`를 되부를 때마다 얼마만큼의 추가적인 저장장소가 만들어져야 하는지를 계산해 보면 된다. 처음 S의 크기가 n이면, 추가적으로 필요한 U와 V의 저장장소 크기의 합은 n이 된다. 다음 되부름 호출에는 $\frac{n}{2}$ 의 추가적인 저장장소가 필요하고, 다음에는 $\frac{n}{4}$가 필요하고, 등등 이런식으로 계속된다. 그러므로 추가적으로 필요한 총 저장장소의 크기는 $n+\frac{n}{2}+\frac{n}{4}+\cdots = 2n$이다. 결론적으로 이 알고리즘의 공간복잡도는 $2n \in \Theta(n)$이라고 할 수 있다.
- 추가적으로 필요한 저장장소가 n이 되도록, 즉, 공간복잡도가 n이 되도록 알고리즘을 향상시킬 수 있다 (다음 절의 알고리즘). 그러나 합병정렬 알고리즘이 제자리정렬 알고리즘이 될 수는 없다.

2.4 공간복잡도가 향상된 알고리즘

2.4.1 합병정렬(Mergesort)

- 문제: n개의 정수를 비내림차순으로 정렬하시오.
- 입력: 정수 n, 크기가 n인 배열 S[1..n]
- 출력: 비내림차순으로 정렬된 배열 S[1..n]
- 알고리즘:

```
void mergesort2 (index low, index high)
{
   index mid;

   if (low < high) {
      mid = (low + high) /2;
      mergesort2(low,mid);
      mergesort2(mid+1,high);
      merge2(low,mid,high);
   }
}

...
mergesort2(1,n);
...
```

2.4.2 합병(Merge)

- 문제: 두개의 정렬된 배열을 하나의 정렬된 배열로 합병하시오.
- 입력: (1) 첨자 low, mid, high, (2) 부분 배열 S[low..high], 여기서 S[low..mid]와 S[mid+1..high]는 이미 각각 정렬이 완료되어 있음.
- 출력: 정렬이 완료된 부분배열 S[1..high]
- 알고리즘:

```
void merge2(index low, index mid, index high)
{
   index i,j,k;
   keytype U[low..high];           // 합병하는데 필요한 지역 배열

   i = low; j = mid + 1; k = low;
   while (i <= mid && j <= high) {
      if (S[i] < S[j]) {
         U[k] = S[i];
         i++;
      }
      else {
         U[k] = S[j];
         j++;
      }
      k++;
   }
   if (i > mid)
      copy S[j] through S[high] to U[k] through U[high];
   else
      copy S[i] through S[mid] to U[k] through U[high];
   copy U[low] through U[high] to S[low] through S[high];
}
```

제 3 절 도사 정리

3.1 도사정리(The Master Theorem)

(Cormen 등의 Introduction to Algorithms(MIT Press, 1990)에서 발췌)

- a와 b를 1보다 큰 상수하고 하고, $f(n)$을 어떤 함수라고 하고, 음이 아닌 정수 n에 대해서 정의된 재현방정식 $T(n)$이 다음의 형태를 이룬다고 하자.

$$T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$$

그러면 $T(n)$은 다음과 같이 점근적인 한계점(asymptotic bound)을 가질 수 있다.

1. 어떤 상수 $\varepsilon > 0$에 대해서, 만약 $f(n) = O(n^{\log_b a - \varepsilon})$이면, $T(n) = \Theta(n^{\log_b a})$.
2. $f(n) = \Theta(n^{\log_b a})$이면, $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. 어떤 상수 $\varepsilon > 0$에 대해서, 만약 $f(n) = \Omega(n^{\log_b a + \varepsilon})$이고, 어떤 상수 $c < 1$과 충분히 큰 모든 n에 대해서, $a \times f(\frac{n}{b}) \leq c \times f(n)$이면, $T(n) = \Theta(f(n))$.

   여기서 $\frac{n}{b}$은 $\lfloor \frac{n}{b} \rfloor$로 여겨도 되고, $\lceil \frac{n}{b} \rceil$으로 여겨도 된다.

- 이 정리의 증명은 상당히 복잡하기 때문에 이 강의에서는 다루지 않는다. 어떻게 증명하는지 모르고는 밥이 넘어가지 않는 학생은 Cormen의 책을 참조하기 바란다.

3.2 도사정리 적용의 예

- 예제 1:
$$T(n) = 9T\left(\frac{n}{3}\right) + n$$
여기서 $a = 9, b = 3, f(n) = n$이고, $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ 이므로, $\varepsilon = 1$일때, $f(n) = O(n^{\log_3 9 - \varepsilon})$ 이라고 할 수 있다. 도사정리 1번을 적용하면, $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$이 된다.
- 예제 2:
$$T(n) = T\left(\frac{2n}{3}\right) + 1$$
여기서 $a = 1, b = \frac{3}{2}, f(n) = 1$ 이고, $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = \Theta(1)$ 이므로, $f(n) = \Theta(1)$ 이라고 할 수 있다. 도사정리 2번을 적용하면, $T(n) = \Theta(1 \lg n) = \Theta(\lg n)$이 된다.
- 예제 3:
$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$
여기서 $a = 3, b = 4, f(n) = n \lg n$ 이고, $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ 이므로, $\varepsilon \approx 0.2$일때, $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$이라고 할 수 있다. 도사정리 3번을 적용할 수 있는지 보기 위해서, 충분히 큰 n에 대해서, $3f(\frac{n}{4}) \leq c \times f(n)$이 성립하는 1보다 작은 c가 존재하는가를 보아야 한다. 여기서, $c = \frac{3}{4}$이면, $3\frac{n}{4}\lg(\frac{n}{4}) \leq \frac{3}{4} n \lg n$은 충분히 큰 n에 대해서 항상 성립한다. 따라서 $T(n) = \Theta(n \lg n)$이 된다.
- 예제 4:
$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$
여기서 $a = 2, b = 2$ $f(n) = n \lg n$ 이고, $n^{\log_b a} = n^{\log_2 2} = \Theta(n)$ 이므로, $f(n) = \Omega(n^{\log_2 2 + \varepsilon})$이라고 할 수 있다. 여기서 도사정리 3번을 적용할 수 있는지 보기 위해서, 충분히 큰 n에 대해서, $2f(\frac{n}{2}) \leq c \times f(n)$이 성립하는 1보다 작은 c가 존재하는가를 보아야 한다. 그러나, $2\frac{n}{2}\lg(\frac{n}{2}) \leq cn \lg n$에서 충분히 큰 n에 대해서 항상 성립하는 c는 없다. 왜냐하면, 위의 식을 정리하면 $\frac{\lg n - 1}{\lg n} \leq c$가 되고, 어떠한 c를 선택하더라도 이 부등식은 성립할 수 없다. 따라서 도사정리를 이용하여 해를 구할 수 없다. 이런 경우는 다음의 도사보조정리를 이용하여 해를 구할 수 있다.

3.3 도사보조정리

- 다음 형태의 재현방정식
$$T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$$
에서, $k \geq 0$인 어떤 k에 대해서 $f(n)$이 $\Theta(n^{\log_b a} \lg^k n)$이면 $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$이 된다. (증명 생략)
- 도사보조정리의 적용의 예
  위의 보기 4번의 해는 $f(n) = \Theta(n \lg n)$이므로 $T(n) = \Theta(n \lg^2 n)$이 된다.

이 절에서 공부한 도사정리는 재현방정식을 푸는데 상당히 유용하게 쓰인다. 따라서 보지 않고도 재현방정식을 푸는데 적용할 수 있을 만큼 이 정리를 완전히 이해하기를 권장한다.

제 4 절 빠른정렬

- 1962년에 영국의 호아(C.A.R. Hoare)의 의해서 고안
- 빠른정렬(Quicksort)란 이름이 오해의 여지가 있음. 왜냐하면 사실 가장 빠른 정렬 알고리즘이 아니기 때문이다. 차라리 "분할교환정렬(partition exchange sort)"라고 부르는게 더 정확함
- 보기: 15 22 13 27 12 10 20 25

4.1 빠른정렬 알고리즘

- 문제: n개의 정수를 비내림차순으로 정렬
- 입력: 정수 $n > 0$, 크기가 n인 배열 `S[1..n]`
- 출력: 비내림차순으로 정렬된 배열 `S[1..n]`

- 알고리즘:

```
void quicksort (index low, index high)
{
   index pivotpoint;

   if (high > low) {
      partition(low,high,pivotpoint);
      quicksort(low,pivotpoint-1);
      quicksort(pivotpoint+1,high);
   }
}
```

4.2 분할 알고리즘

- 문제: 빠른정렬을 하기 위해서 배열 S를 둘로 쪼갠다.
- 입력: (1) 첨자 low, high, (2) 첨자 low에서 high까지의 S의 부분배열
- 출력: 첨자 low에서 high까지의 S의 부분배열의 기준점(pivot point), pivotpoint
- 알고리즘:

```
void partition (index low, index high,
                index& pivotpoint)
{
   index i,j;
   keytype pivotitem;

   pivotitem = S[low];
               // pivotitem을 위한 첫번째 항목을 고른다
   j = low;
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem) {
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint];
               // pivotitem 값을 pivotpoint에 넣는다
}
```

4.3 분석

4.3.1 분할 알고리즘의 모든 경우를 고려한 시간복잡도 분석

- 기본동작: S[i]와 key와의 비교
- 입력의 크기: 부분배열이 가지고 있는 항목의 수, $n = high - low + 1$
- 분석: 배열의 첫번째 항목만 제외하고 모든 항목을 한번씩 비교하므로, $T(n) = n - 1$이다.

4.3.2 빠른정렬 알고리즘의 최악의 경우를 고려한 시간복잡도 분석

- 기본동작: 분할알고리즘의 S[i]와 key와의 비교
- 입력의 크기: 배열이 S가 가지고 있는 항목의 수, n
- 분석: 이미 비내림차순으로 정렬이 되어있는 배열을 정렬하려는 경우가 최악의 경우가 된다. 왜 그럴까? 비내림차순으로 정렬되어 있으면 첫번째(기준점) 항목보다 작은 항목으 없으므로, 크기가 n인 배열은 크기가 0인 부분배열은 왼쪽에 오고, 크기가 $n-1$인 부분배열은 오른쪽에 오도록 하여 계속 쪼개진다. 따라서,

$$T(n) = T(0) + T(n-1) + n - 1$$

그런데, $T(0) = 0$이므로, 재현방정식은 다음과 같이 된다.

$$\begin{aligned} T(n) &= T(n-1) + n - 1, \quad n > 0 \text{ 이면} \\ T(0) &= 0 \end{aligned}$$

이 재현방정식을 풀면,

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ T(n-1) &= T(n-2) + n - 2 \\ T(n-2) &= T(n-3) + n - 3 \\ &\cdots \\ T(2) &= T(1) + 1 \\ T(1) &= T(0) + 0 \\ T(0) &= 0 \end{aligned}$$

$$T(n) = 1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$$

가 되므로, 이미 정렬이 되어 있는 경우 빠른정렬 알고리즘의 시간복잡도는 $\frac{n(n-1)}{2}$이 된다는 사실을 알았다. 그러면 시간이 더 많이 걸리는 경우는 있을까? 이 경우가 최악의 경우이며, 따라서 이 보다 더 많은 시간이 걸릴 수가 없다는 사실을 수학적으로 엄밀하게 증명해보자.

- 모든 정수 n에 대해서, $W(n) \le \frac{n(n-1)}{2}$ 임을 증명하시오.
- 증명: 수학적귀납법

  귀납출발점: $n=0$일때, $W(0) = 0 \le \frac{0(0-1)}{2}$

  귀납가정: $0 \le k < n$인 모든 k에 대해서, $W(k) \le \frac{k(k-1)}{2}$

  귀납단계: $W(n) \le \frac{n(n-1)}{2}$ 임을 보이면 된다.

$$\begin{aligned} W(n) &\le W(p-1)+W(n-p)+n-1 \quad \texttt{pivotpoint} \text{ 값이 } p\text{인 경우 재현방정식에 의해서} \\ &\le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1 \quad \text{귀납가정에 의하여} \\ &= \frac{p^2-3p+2+(n-p)^2-n+p+2n-2}{2} \\ &= \frac{p^2+(n-p)^2+n-2p}{2} \end{aligned}$$

여기서 p가 1이나 $n-1$일때 최대값을 가진다. 따라서

$$max_{1\le p\le n-1}(p^2+(n-p)^2) = 1^2+(n-1)^2 = n^2-2n+2$$

가 되고, 결과적으로

$$W(n) \le \frac{p^2+(n-p)^2+n-2p}{2} \le \frac{n^2-2n+2+n-2}{2} = \frac{n^2-n}{2} = \frac{n(n-1)}{2}$$

가 된다. 따라서 최악의 경우 시간복잡도는

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

4.3.3 빠른정렬 알고리즘의 평균 시간복잡도 분석

- 기본동작: 분할알고리즘의 `S[i]`와 `key`와의 비교
- 입력의 크기: 배열이 `S`가 가지고 있는 항목의 수, n
- 분석: 배열 안에 있는 항목이 어떤 특정 순으로 정렬이 되어 있는 경우는 사실 별로 없다. 그러므로 분할 알고리즘이 주는 기준점 값은 1부터 n사이의 어떤 값도 될 수가 있고, 그 확률은 모두 같다고 봐도 된다. 따라서, 평균의 경우를 고려한 시간복잡도 분석을 해도 된다. 기준점이 p가 될 확률은 $\frac{1}{n}$이고, 기준점이 p일때 두 부분배열을 정렬하는데 걸리는 평균기간은 $[A(p-1)+A(n-p)]$이고, 분할하는데 걸리는 시간은 $n-1$이므로, 평균적인 시간복잡도는 다음과 같이 된다.

$$\begin{aligned} A(n) &= \sum_{p=1}^{n} \frac{1}{n}[A(p-1)+A(n-p)] + n - 1 \\ &= \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1 \end{aligned}$$

양변을 n으로 곱하면,

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1) \quad \text{---- (1)}$$

n대신 $n-1$을 대입하면,

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \quad \text{---- (2)}$$

(1)에서 (2)를 빼면,

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

간단히 정리하면,

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

여기서,

$$a_n = \frac{A(n)}{n+1}$$

라고 하면, 다음과 같은 재현방정식을 얻을 수가 있다.

$$\begin{aligned} a_n &= a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad n > 0 \text{ 이면} \\ a_0 &= 0 \end{aligned}$$

그러면,

$$\begin{aligned} a_n &= a_{n-1} + \frac{2(n-1)}{n(n+1)} \\ a_{n-1} &= a_{n-2} + \frac{2(n-2)}{(n-1)n} \\ &\cdots \\ a_2 &= a_1 + \frac{1}{3} \\ a_1 &= a_0 + 0 \end{aligned}$$

따라서, 해는

$$\begin{aligned} a_n &= \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \\ &= 2\left(\sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)} \right) \end{aligned}$$

여기에서 오른쪽 항은 무시해도 될 만큼 작으므로 무시한다. $\ln n = \log_e n$이고,

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n$$

이므로, 해는 $a_n \approx 2 \ln n$. 따라서,

$$\begin{aligned} A(n) &\approx (n+1)2\ln n \\ &= (n+1)2(\ln 2)(\lg n) \\ &\approx 1.38(n+1)\lg n \\ &\in \Theta(n \lg n) \end{aligned}$$

제 5 절 행렬 곱셈

5.1 단순한 행렬곱셈(matrix multiplication) 알고리즘

- 문제: $n \times n$ 크기의 행렬의 곱을 구하시오.
- 입력: 양수 n, $n \times n$ 크기의 행렬 A와 B
- 출력: 행렬 A와 B의 곱인 C
- 알고리즘:

```
void matrixmult (int n,
                 const number A[][],
                 const number B[][],
                 number C[][])
{
   index i,j,k;

   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++) {
         C[i][j] = 0;
         for (k = 1; k <= n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```

- 시간복잡도 분석 I:
  - ▷ 기본동작: 가장 안쪽의 맴돌이(loop)에 있는 곱셈하는 동작
  - ▷ 입력의 크기: 행과 열의 수, n
  - ▷ 모든 경우 시간복잡도 분석: 총 곱셈의 횟수는 $T(n) = n \times n \times n = n^3 \in \Theta(n^3)$이다.
- 시간복잡도 분석 II:
  - ▷ 기본동작: 가장 안쪽의 맴돌이(loop)에 있는 덧셈하는 동작
  - ▷ 입력의 크기: 행과 열의 수, n
  - ▷ 모든 경우 시간복잡도 분석: 총 덧셈의 횟수는 $T(n) = (n-1) \times n \times n = n^3 - n^2 \in \Theta(n^3)$이다.

5.2 2×2 행렬곱셈: 쉬트라쎈의 방법

- 문제: 두 2×2 행렬 A와 B의 곱(product) C,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

- 쉬트라쎈(Strassen)의 해:

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

여기서

$$\begin{aligned} m_1 &= (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22}) \times b_{11} \\ m_3 &= a_{11} \times (b_{12} - b_{22}) \\ m_4 &= a_{22} \times (b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12}) \times b_{22} \\ m_6 &= (a_{21} - a_{11}) \times (b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned}$$

- 시간복잡도 분석: 단순한 방법은 8번의 곱셈과 4번의 덧셈이 필요한 반면, 쉬트라쎈의 방법은 7번의 곱셈과 18번의 덧셈/뺄셈을 필요로 한다. 언뜻 봐서는 전혀 좋아지지 않았다! 그러나 행렬의 크기가 커지면 쉬트라쎈의 방법의 가치가 들어난다.

5.3 $n \times n$ 행렬곱셈: 쉬트라쎈의 방법

- 문제: n이 2의 거듭제곱이고, 각 행렬을 4개의 부분행렬(submatrix)로 나눈다고 가정하자. 두 $n \times n$ 행렬 A와 B의 곱 C:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- 쉬트라쎈(Strassen)의 해:

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

여기서

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} + B_{22}) \\ M_4 &= A_{22} \times (B_{21} + B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} \\ M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

5.4 쉬트라쎈의 알고리즘

- 문제: n이 2의 거듭제곱일때, $n \times n$ 크기의 두 행렬의 곱을 구하시오.
- 입력: 정수 n, $n \times n$ 크기의 행렬 A와 B
- 출력: 행렬 A와 B의 곱인 C
- 알고리즘:

```
void strassen (int n,
               n*n_matrix A,
               n*n_matrix B,
               n*n_matrix& C)
{
   if (n <= 분기점)
      단순한 알고리즘을 사용하여 C = A * B를 계산;
   else {
      A를 4개의 부분행렬 A11,A12,A21,A22로 분할;
      B를 4개의 부분행렬 B11,B12,B21,B22로 분할;
      쉬트라쎈의 방법을 사용하여 C = A * B를 계산;
      // 되부르는 모습의 예: strassen(n/2,A11+A22,B11+B22,M1)
   }
}
```

- 용어: 분기점(threshold)이란? 단순한 알고리즘보다 쉬트라쎈의 알고리즘을 사용하는 편이 더 좋을 것이라고 예상되는 지점.

5.5 분석

5.5.1 시간복잡도 분석 I

- 기본동작: 곱셈하는 동작
- 입력의 크기: 행과 열의 수, n

- 모든 경우 시간복잡도 분석: 분기점의 값을 1이라고 하자. (분기점의 값은 차수에 전혀 영향을 미치지 않는다.) 재현방정식은

$$\begin{array}{rcl} T(n) & = & 7T\left(\frac{n}{2}\right) \quad n>1 \text{ 이고, } n=2^k(k\geq 1) \\ T(1) & = & 1 \end{array}$$

  이 식을 전개해 보면,

$$\begin{array}{rcl} T(n) & = & 7\times 7\times\cdots\times 7 \quad (k\text{번}) \\ & = & 7^k \\ & = & 7^{\lg n} \\ & = & n^{\lg 7} \\ & = & n^{2.81} \\ & \in & \Theta(n^{2.81}) \end{array}$$

  이 결과는 수학적귀납법에 의해서 증명이 가능하다. 증명을 해보라. 사실 위의 재현방정식은 도사정리 3가지 중에서 1번을 이용하면 간단히 해를 구할 수 있다.

5.5.2 시간복잡도 분석 II

- 기본동작: 뎃셈/뺄셈하는 동작
- 입력의 크기: 행과 열의 수, n
- 모든 경우 시간복잡도 분석: 위에서와 마찬가지로 분기점의 값을 1이라고 하자. 재현방정식은

$$\begin{array}{rcl} T(n) & = & 7T(\frac{n}{2})+18(\frac{n}{2})^2 \quad n>1 \text{ 이고, } n=2^k(k\geq 1) \\ T(1) & = & 0 \end{array}$$

  도사정리의 3가지 중에서 1번을 이용하면 간단히 해를 구할 수 있다.

$$T(n)=\Theta(n^{\lg_2 7})=\Theta(n^{2.81})$$

5.6 사색

- 두 개의 행렬을 곱하기 위한 문제에 대해서 시간복잡도가 $\Theta(n^2)$이 되는 알고리즘을 만들어 낸 사람은 아무도 없다.
- 게다가 그러한 알고리즘을 만들 수 없다고 증명한 사람도 아무도 없다.

제 6 절 분할정복법을 사용하지 말아야 하는 경우

- 크기가 n인 입력이 2개 이상의 조각으로 분할되며, 분할된 부분들의 크기가 거의 n에 가깝게 되는 경우 $\Rightarrow$ 시간복잡도: 지수(exponential) 시간
- 크기가 n인 입력이 거의 n개의 조각으로 분할되며, 분할된 부분의 크기가 n/c인 경우. 여기서 c는 상수이다. $\Rightarrow$ 시간복잡도: $\Theta(n^{\lg n})$

제 3 장

동적계획

- 분할정복식 알고리즘 설계법은 하향식 해결법으로서, 나누어진 부분들 사이에 서로 상관관계가 없는 문제를 해결하는데 적합하다. (II장에서 배운 합병정렬과 빠른정렬 알고리즘을 연상해 보라)
- 피보나찌 알고리즘의 경우에는 나누어진 부분들이 서로 연관이 있다. 즉, 분할정복식 방법을 적용하여 알고리즘을 설계하게 되면 같은 항을 한 번 이상 계산하는 결과를 초래하게 되므로 효율적이지가 않다. 따라서 이 경우에는 분할정복식 방법은 적합하지 않다.
- 동적계획법(Dynamic programming)은 상향식 해결법(bottom-up approach)을 사용하여 알고리즘을 설계하는 방법이다. 이 방법은 분할정복식 방법과 마찬가지로 문제를 나눈 후에 나누어진 부분들을 먼저 푼다. 그러나 이미 풀어서 답을 알고있는 부분의 결과가 다시 필요한 경우에는 반복하여 계산하는 대신에 이미 계산된 결과를 그냥 사용한다. (I장에서 배운 반복적 방법을 이용한 피보나찌 알고리즘을 연상해 보라)

제 1 절 이항계수 구하기

- 이항계수 구하는 공식

$$\left[\begin{array}{c} n \\ k \end{array} \right] = \frac{n!}{k!(n-k)!} \quad \text{for } 0 \le k \le n$$

- 계산량이 많은 $n!$ 이나 $k!$을 계산하지 않고 이항계수(binomial coefficient)를 구하기 위해서 통상 다음식을 사용한다.

$$\left[\begin{array}{c} n \\ k \end{array} \right] = \begin{cases} \left[\begin{array}{c} n-1 \\ k-1 \end{array} \right] + \left[\begin{array}{c} n-1 \\ k \end{array} \right] & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

1.1 알고리즘: 분할정복식 접근방법

- 문제: 이항계수를 계산한다
- 입력: 음수가 아닌 정수 n 과 k, 여기서 $k \le n$
- 출력: `bin`, $\left[\begin{array}{c} n \\ k \end{array} \right]$
- 알고리즘:

```
int bin(int n,int k)
{
   if (k == 0 || n == k)
      return 1;
   else
      return bin(n-1,k-1) + bin(n-1,k);
}
```

- 시간복잡도 분석: 위의 알고리즘은 작성하기는 간단하지만, 효율적이지 않다. 왜냐하면 알고리즘을 되부를때(recursive call) 같은 계산을 반복해서 수행하기 때문이다. 예를들면, `bin(n-1,k-1)`과 `bin(n-1,k)`는 둘다 `bin(n-2,k-1)`의 결과가 필요한데, 따로 반복하여 계산한다.
  $\left[\begin{array}{c} n \\ k \end{array} \right]$을 구하기 위해서 이 알고리즘이 계산하는 항(term)의 갯수는 $2\left[\begin{array}{c} n \\ k \end{array} \right] - 1$이다.
- 증명: (n대한 수학적귀납법으로 증명)
  귀납출발점: 항의 갯수 n이 1일때 $2\left[\begin{array}{c} 1 \\ k \end{array} \right] - 1 = 2 \times 1 - 1 = 1$ 이 됨을 보이면 된다. $\left[\begin{array}{c} 1 \\ k \end{array} \right]$는 $k = 0$이나 1 일때 1 이므로 항의 갯수는 항상 1이다.
  귀납가정: $\left[\begin{array}{c} n \\ k \end{array} \right]$을 계산하기 위한 항의 갯수는 $2\left[\begin{array}{c} n \\ k \end{array} \right] - 1$ 이라고 가정한다.

귀납절차: $\left[\begin{smallmatrix} n+1 \\ k \end{smallmatrix}\right]$ 을 계산하기 위한 항의 갯수가 $2\left[\begin{smallmatrix} n+1 \\ k \end{smallmatrix}\right]-1$ 임을 보이면 된다. 알고리즘에 의해서 $\left[\begin{smallmatrix} n+1 \\ k \end{smallmatrix}\right] = \left[\begin{smallmatrix} n \\ k-1 \end{smallmatrix}\right] + \left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$ 이므로, $\left[\begin{smallmatrix} n+1 \\ k \end{smallmatrix}\right]$ 를 계산하기 위한 항의 총 갯수는 $\left[\begin{smallmatrix} n \\ k-1 \end{smallmatrix}\right]$ 을 계산하기 위한 항의 총 갯수와 $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$ 를 계산하기 위한 항의 총 갯수에다가 이 둘을 더하기 위한 항 1을 더한 수가 된다. 그런데 $\left[\begin{smallmatrix} n \\ k-1 \end{smallmatrix}\right]$ 을 계산하기 위한 항의 갯수는 가정 의해서 $2\left[\begin{smallmatrix} n \\ k-1 \end{smallmatrix}\right]-1$ 이고, $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$ 를 계산하기 위한 항의 갯수는 가정 의해서 $2\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]-1$ 이다. 따라서 항의 총 갯수는

$$
\begin{aligned}
& 2\left[\begin{smallmatrix} n \\ k-1 \end{smallmatrix}\right]-1+2\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]-1+1 \\
&= 2\left(\frac{n!}{(k-1)!(n-k+1)!}+\frac{n!}{k!(n-k)!}\right)-1 \\
&= 2\left(\frac{n!(k+n+1-k)}{k!(n+1-k)!}\right)-1 \\
&= 2\left(\frac{n!(n+1)}{k!(n+1-k)!}\right)-1 \\
&= 2\left(\frac{(n+1)!}{k!(n+1-k)!}\right)-1 \\
&= 2\left[\begin{smallmatrix} n+1 \\ k \end{smallmatrix}\right]-1
\end{aligned}
$$

증명 끝.

1.2 알고리즘: 동적계획식 접근방법

- 설계전략

  1. 되부름 관계식(recursive property)을 정립:
     2차원 배열 B를 만들고, 각 $B[i][j]$에는 $\left[\begin{smallmatrix} i \\ j \end{smallmatrix}\right]$ 값을 저장하도록 하면, 그 값은 다음과 같은 되부름 관계식으로 계산할 수 있다.

$$
B[i][j] = \begin{cases} B[i-1][j-1]+B[i-1][j] & \text{if } 0 < j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \end{cases}
$$

  2. $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$ 를 구하기 위해서는 다음과 같이 $B[0][0]$부터 시작하여 위에서 아래로 되부름 관계식을 적용하여 배열을 채워나가면 된다. 결국 $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$ 값은 $B[n][k]$에 저장된다.

```
B[0][0]
B[1][0]   B[1][1]
B[2][0]   B[2][1]     B[2][2]
.............................
.......................................
B[k][0]   B[k][1]   ..................... B[k][k]
B[k+1][0] B[k+1][1] ..................... B[k+1][k]
.............................................
B[n][0]   B[n][1]   ..................... B[n][k]
```

     위의 배열의 각 항은 다음과 같이 계산이 된다.

```
B[i-1][j-1] +  B[i-1][j]
             \\
               B[i][j]
```

     `B[4][2]`를 계산해 보시오.

- **문제:** 이항계수를 계산한다
- **입력:** 음수가 아닌 정수 n 과 k, 여기서 $k \le n$
- **출력:** `bin`, $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]$
- **알고리즘:**

```
int bin2(int n,int k)
{
   index i,j;
   int B[0..n][0..k];

   for (i = 0; i <= n; i++)
      for (j = 0; j <= minimum(i,k); j++)
```

```
            if (j == 0 || j == i)
               B[i][j] = 1;
            else  B[i][j] = B[i-1][j-1] + B[i-1][j];
      return B[n][k];
   }
```

- 분석:
  - ▷ 기본동작: for-j 맴돌이(loop)를 수행하는 횟수
  - ▷ 입력의 크기: n, k

$$\begin{array}{ll} i=0\text{일 때 j 맴돌이 수행 횟수} & : \quad 1 \\ i=1\text{일 때 j 맴돌이 수행 횟수} & : \quad 2 \\ i=2\text{일 때 j 맴돌이 수행 횟수} & : \quad 3 \\ \cdots\cdots\cdots\cdots & \\ i=k-1\text{일 때 j 맴돌이 수행 횟수} & : \quad k \\ i=k\text{일 때 j 맴돌이 수행 횟수} & : \quad k+1 \\ i=k+1\text{일 때 j 맴돌이 수행 횟수} & : \quad k+1 \\ \cdots\cdots\cdots\cdots & \\ i=n\text{일 때 j 맴돌이 수행 횟수} & : \quad k+1 \end{array}$$

따라서 총 수행횟수는:

$$\begin{aligned} 1+2+3+\cdots+k+\overbrace{(k+1)+\cdots+(k+1)}^{n-k+1 \text{ times}} &= \frac{k(k+1)}{2}+(n-k+1)(k+1) \\ &= \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk) \end{aligned}$$

제 2 절 최단경로찾기: Floyd의 알고리즘

- 최단경로(Shortest paths) 찾기 문제의 보기: 한 도시에서 다른 도시로 직항로가 없는 경우 가장 빨리 갈 수 있는 항로를 찾는 문제
- 그래프 이론(Graph theory) 용어: (95쪽의 Figure 3.2를 참조)
  정점(vertex, node), 이음선(edge, arc), 방향성 그래프(directed graph, digraph), 가중치(weight), 가중치가 포함된 그래프(weighted graph), 경로(path), 단순경로(simple path) - 같은 정점을 두번 지나지 않음, 순환(cycle) - 한 정점에서 다시 그 정점으로 돌아오는 경로, 순환적 그래프(cyclic graph), 비순환적 그래프(acyclic graph), 길이(length)
- 문제: 가중치가 포함된, 방향성 그래프(weighted, directed graph)에서 최단경로 찾기. (95쪽의 Figure 3.2를 참조)
- 주어진 문제에 대하여 하나 이상의 많은 해답이 존재할때, 이 가운데에서 가장 최적인 해답(optimal solution)을 찾아야 하는 문제를 최적화문제(optimization problem)라고한다. 최단경로찾기문제(shortest paths problem)는 최적화문제에 속한다.
- 주먹구구식 알고리즘: 한 정점에서 다른 정점으로의 모든 경로의 길이를 구한 뒤, 그들 중에서 최소길이를 찾는다.
  분석: 그래프가 n개의 정점을 가지고 있고, 모든 정점들 사이에 이음선이 존재한다고 가정하자. 그러면 한 정점 v_i에서 어떤 다른 정점 v_j로 가는 경로들을 다 모아 보면, 그 경로들 중에서 나머지 모든 정점을 한번 씩은 꼭 거쳐서 가는 경로들도 포함되어 있는데, 그 경로들의 수 만 우선 계산해 보자. v_i에서 출발하여 처음에 도착할 수 있는 정점의 가지수는 $n-2$개 이고, 그 중에 하나를 선택하면, 그 다음에 도착할 수 있는 정점의 가지수는 $n-3$개 이고, 이렇게 계속하여 계산해 보면, 총 경로의 갯수는 $(n-2)(n-3)\cdots 1=(n-2)!$이 된다. 이 경로의 갯수 만 보아도 지수보다 훨씬 크므로, 이 알고리즘은 절대적으로 비효율적이다!

2.1 동적계획식 설계전략 - 자료구조

- 그래프의 인접행렬(adjacent matrix)식 표현: W

$$W[i][j]=\begin{cases} \text{이음선의 가중치} & v_i \text{ 에서 } v_j \text{ 로의 이음선이 있다면} \\ \infty & v_i \text{ 에서 } v_j \text{ 로의 이음선이 없다면} \\ 0 & i=j \text{ 이면} \end{cases}$$

- 그래프에서 최단경로의 길이의 표현: $0 \le k \le n$ 인, $D^{(k)}$

$$D^{(k)}[i][j]=\{v_1, v_2, \ldots, v_k\} \text{ 의 정점들 만을 통해서 } v_i \text{ 에서 } v_j \text{ 로 가는 최단경로의 길이}$$

- 보기: 95쪽의 Figure 3.2에 있는 그래프는 다음과 같이 W로 표현할 수 있으며, 각 정점들 사이의 최단거리는 계산하여 다음과 같이 D로 표현한다.

$W[i][j]$	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D[i][j]$	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

여기서, $0 \le k \le 5$ 일때 , $D^{(k)}[2][5]$를 구해보라.

- $D^{(0)}=W$이고, $D^{(n)}=D$ 임은 분명하다. 따라서 D를 구하기 위해서는 $D^{(0)}$를 가지고 $D^{(n)}$을 구할 수 있는 방법을 고안해 내어야 한다.

2.2 동적계획식 설계절차

1. $D^{(k-1)}$을 가지고 $D^{(k)}$를 계산할 수 있는 되부름 관계식 (recursive property)을 정립

$$D^{(k)}[i][j] = \min(\underbrace{D^{(k-1)}[i][j]}_{\text{경우 1}}, \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{경우 2}})$$

   경우 1: $\{v_1, v_2, \ldots, v_k\}$ 의 정점들 만을 통해서 v_i에서 v_j로 가는 최단경로가 v_k를 거치지 않는 경우. 보기: $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$

   경우 2: $\{v_1, v_2, \ldots, v_k\}$ 의 정점들 만을 통해서 v_i에서 v_j로 가는 최단경로가 v_k를 거치는 경우. 보기: $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$

   보기: $D^{(2)}[5][4]$

2. 상향식 해결법으로 $k = 1$ 부터 n 까지 다음과 같이 이 과정을 반복하여 해를 구한다.

$$D^{(0)}, D^{(1)}, \ldots, D^{(n)}$$

2.3 알고리즘

2.3.1 Floyd의 알고리즘 I

- 문제: 가중치가 포함된 그래프의 각 정점에서 다른 모든 정점 까지의 최단거리를 계산하라.
- 입력: 가중치가 포함된, 방향성 그래프 W와 그 그래프에서의 정점의 수 n.
- 출력: 최단거리의 길이가 포함된 배열 D
- 알고리즘:

```
void floyd(int n,
         const number W[][],
               number D[][])
{
   int i,j,k;

   D = W;
   for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            D[i][j] = minimum(D[i][j],D[i][k]+D[k][j]);
}
```

- 모든 경우를 고려한 분석:
  - ▷ 기본동작: for-j 맴돌이(loop)안의 대입문
  - ▷ 입력의 크기: 그래프에서의 정점의 수 n

$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$

2.3.2 Floyd의 알고리즘 II

- 문제: 가중치가 포함된 그래프의 각 정점에서 다른 모든 정점까지의 최단거리를 계산하고, 각각의 최단경로를 구하라.
- 입력: 가중치가 포함된, 방향성 그래프 W와 그 그래프에서의 정점의 수 n.
- 출력: 최단거리의 길이가 포함된 배열 D, 그리고 다음을 만족하는 배열 P:

$$P[i][j] = \begin{cases} v_i \text{ 에서 } v_j \text{ 까지 가는 최단경로의 중간에 놓여있는 정점이 최소한 하나는 있는 경우} \rightarrow \\ \qquad \text{그 놓여있는 정점 중에서 가장 큰 인덱스} \\ \text{최단경로의 중간에 놓여있는 정점이 없는 경우} \rightarrow 0 \end{cases}$$

- 알고리즘:

```
void floyd2(int n,
   const number W[][],
         number D[][],
          index P[][])
{
   index i,j,k;

   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
```

```
            P[i][j] = 0;
      D = W;
      for (k = 1; k <= n; k++)
         for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
               if (D[i][k] + D[k][j] < D[i][j]) {
                  P[i][j] = k;
                  D[i][j] = D[i][k] + D[k][j];
               }
   }
```

- 95쪽의 Figure 3.2를 가지고 `D`와 `P`를 구해 보시오.

$P[i][j]$	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

2.3.3 최단경로의 출력

- 문제: 최단경로 상에 놓여있는 정점을 출력하라
- 알고리즘:

```
void path(index q,r)
{
   if (P[q][r] != 0) {
      path(q,P[q][r]);
      cout << "v" << P[q][r];
      path(P[q][r],r);
   }
}
```

- 위의 `P`를 가지고 `path(5,3)`을 구해 보시오.

```
path(5,3) = 4
  path(5,4) = 1
    path(5,1) = 0
    v1
    path(1,4) = 0
  v4
  path(4,3) = 0
```

결과: `v1 v4`. 즉, v_5에서 v_3으로 가는 최단경로는 v_5, v_1, v_4, v_3이다.

제 3 절 동적계획법과 최적화 문제

3.1 동적계획법에 의한 설계 절차

1. 문제의 입력에 대해서 최적(optimal)의 해답을 주는 되부름 관계식(recursive property)을 설정
2. 상향적인 해결방법으로 최적의 해답을 계산한다.
3. 상향적인 해결방법으로 최적의 해답을 구축한다.

3.2 최적의 원칙

- 어떤 문제의 입력에 대한 최적해가 그 입력을 나누어 쪼갠 여러 부분에 대한 최적해를 항상 포함하고 있으면, 그 문제는 **최적의 원칙(the principle of optimality)**이 적용된다 라고 한다.
- 보기 1: 최단경로를 구하는 문제에서, v_k를 v_i에서 v_j로 가는 최적 경로 상의 정점이라고 하면, v_i에서 v_k로 가는 부분경로와 v_k에서 v_j로 가는 부분경로도 반드시 최적이어야 한다. 이렇게 되면 최적이 원칙을 준수하게 되므로 동적계획법을 사용하여 이 문제를 풀수있다!
- 보기 2: 최장경로(longest path)를 구하는 문제에서는, 최적의 원칙이 적용되지 않는다. 104쪽의 Example 3.4를 보시오. v_1에서 v_4로의 최장경로는 $[v_1, v_3, v_2, v_4]$가 된다. 그러나, 이 경로의 부분경로인 v_1에서 v_3으로의 최장경로는 $[v_1, v_3]$이 아니고, $[v_1, v_2, v_3]$이다. 따라서 최적의 원칙이 적용되지 않는다. (주의: 여기서는 단순경로(simple path), 즉 순환(cycle)이 없는 경로만 고려한다.)

제 4 절 연쇄행렬곱셈(Matrix-chain Multiplication)

- $i \times j$ 행렬과 $j \times k$행렬을 곱하기 위해서는 일반적으로 $i \times j \times k$번 만큼의 기본적인 곱셈이 필요하다.
- 연쇄적으로 행렬을 곱할 때, 어떤 행렬곱셈을 먼저 수행하느냐에 따라서 필요한 기본적인 곱셈의 횟수가 달라지게 된다. 예를 들어서, 다음 연쇄행렬곱셈을 생각해보자: $A_1 \times A_2 \times A_3$. 여기서 A_1의 크기는 10×100이고, A_2의 크기는 100×5이고, A_3의 크기는 5×50라고 하자. 만약 $A_1 \times A_2$를 먼저 계산한다면, 기본적인 곱셈의 총 횟수는 7,500회가 된다. 그러나 만약 $A_2 \times A_3$를 먼저 수행한다면, 총 횟수는 75,000회가 된다. 따라서, 연쇄적으로 행렬을 곱할때 기본적인 곱셈의 횟수가 가장 적게 되는 최적의 순서를 결정하는 알고리즘을 개발하는 것이 본 절의 목표이다.
- 주먹구구식 알고리즘: 가능한 모든 순서를 모두 고려해 보고, 그 가운데에서 가장 최소를 택한다. 그러나 이 알고리즘은 최소한 지수(exponential-time)적인 시간복잡도를 가진다.

  증명: n개의 행렬$(A_1, A_2, \ldots, A_n)$을 곱할 수 있는 모든 순서의 가지수를 t_n이라고 하자. 만약 A_1이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_2, \ldots, A_n$을 곱하는 데는 t_{n-1}개의 가지수가 있을 것이다. A_n이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_1, \ldots, A_{n-1}$을 곱하는 데는 또한 t_{n-1}개의 가지수가 있을 것이다. 그러면, $t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$이고 $t_2 = 1$이라는 사실은 쉽게 알 수 있다. 따라서 $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq 2^3 t_{n-3} \geq \cdots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$

4.1 동적계획식 설계전략

- d_k를 행렬 A_k의 열(column)의 수라고 하자. 그러면 자연히 A_k의 행(row)의 수는 d_{k-1}가 된다. 그리고 A_1의 행의 수는 d_0라고 하자. 그러면, 다음과 같이 되부름 관계식을 구축할 수 있다. $1 \leq i \leq j \leq n$ 일때

$$\begin{array}{lcl} M[i][j] & = & i < j \text{ 일때 } A_i\text{부터 } A_j\text{까지의 행렬을 곱하는데 필요한 기본적인 곱셈의 최소 횟수} \\ & = & \min_{i \leq k \leq j-1}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) \text{ if } i < j \\ M[i][i] & = & 0 \end{array}$$

- 보기:

A_1	A_2	A_3	A_4	A_5	A_6
5×2	2×3	3×4	4×6	6×7	7×8

$$\begin{array}{rcl} M[4][6] & = & \min_{4 \leq k \leq 5}(M[4][4] + M[5][6] + 4 \times 6 \times 8, M[4][5] + M[6][6] + 4 \times 7 \times 8) \\ & = & \min(0 + 6 \times 7 \times 8 + 4 \times 6 \times 8, 4 \times 6 \times 7 + 0 + 4 \times 7 \times 8) \\ & = & \min(528, 392) = 392 \end{array}$$

$M[i][j]$	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

- 최적 순서는 얻기 위해서는 $M[i][j]$를 계산할 때 최소값을 주는 k 값을 $P[i][j]$에 기억한다. 예를 들어서, $P[2][5] = 4$ 인 경우의 최적 순서는 $(A_2 A_3 A_4) A_5$이다. 위의 보기를 가지고 P를 구축하면 다음과 같이 된다.

$P[i][j]$	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

  위의 배열 P가 주어졌을 때, 최적 분해는 $(A_1((((A_2 A_3) A_4) A_5) A_6))$.
- 최적의 원칙이 이 알고리즘에 적용이 되는가? 생각해 보시오.

4.2 알고리즘

4.2.1 최소곱셈(Minimum Multiplication): Godbole(1972)

- 문제: n개의 행렬을 곱하는데 필요한 기본적인 곱셈의 횟수의 최소치를 결정하고, 그 최소치를 구하는 순서를 결정하라.
- 입력: 행렬의 수 `n`, 배열 `d[1..n]`, 여기서 `d[i-1]` × `d[i]`는 `i`번째 행렬의 규모를 나타낸다.
- 출력: 기본적인 곱셈의 횟수의 최소치를 나타내는 `minmult`; 최적의 순서를 얻을 수 있는 배열 `P`, 여기서 `P[i][j]`는 행렬 `i`부터 `j`까지가 최적의 순서로 갈라지는 기점을 나타낸다.
- **알고리즘:**

```
int minmult(int n,
      const int d[],
          index P[][])
{
   index i,j,k,diagonal;
   int M[1..n,1..n];
```

```
   for (i = 1; i <= n; i++)
      M[i][i] = 0;
   for (diagonal = 1; diagonal <= n-1; diagonal++)
      for (i = 1; i <= n-diagonal; i++) {
         j = i + diagonal;
         M[i][j] = minimum(M[i][k]+M[k+1][j]+d[i-1]*d[k]*d[j]);
                   where i <= k <= j-1
         P[i][j] = 최소치를 주는 k의 값
      }
   return M[1][n];
}
```

4.2.2 최소곱셈 알고리즘의 모든 경우를 고려한 분석

- 기본동작: 각 k 값에 대하여 실행된 명령문(instruction), 여기서 최소값인 지를 알아보는 비교문도 포함한다.
- 입력의 크기: 곱할 행렬의 수 n
- 해석: $j = i + diagonal$이므로, k-맴돌이(loop)를 수행하는 횟수는

$$(j-1) - i + 1 = i + diagonal - 1 - i + 1 = diagonal$$

  이 되고, for-i-맴돌이를 수행하는 횟수는 $n - diagonal$이 된다. 따라서

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

4.2.3 최적의 해를 주는 순서의 출력

- 문제: n개의 행렬을 곱하는 최적의 순서를 출력하시오.
- 입력: n과 P.
- 출력: 최적의 순서
- 알고리즘:

```
void order(index i, index j)
{
   if (i == j)
      cout << "A" << i;
   else {
      k = P[i][j];
      cout << "(";
      order(i,k);
      order(k+1,j);
      cout << ")";
   }
}
```

- order(i,j)의 의미: $A_i \times \cdots \times A_j$의 계산을 수행하는데 기본적인 곱셈의 수가 가장 적게 드는 순서대로 괄호를 쳐서 출력하시오.
- 분석: $T(n) \in \Theta(n)$. 어떻게?

4.3 다른 알고리즘

- Yao(1982) - $\Theta(n^2)$
- Hu and Shing(1982,1984) - $\Theta(n \lg n)$

제 4 장

탐욕적인 접근방법

- 탐욕적인 알고리즘(Greedy algorithm)은 결정을 해야할 때마다 그 순간에 가장 좋다고 생각되는 것을 해답으로 선택하므로서 최종적인 해답에 도달한다.
- 그 순간의 선택은 그 당시(local)에는 최적이다. 그러나 최적이라고 생각했던 해답들을 모아서 최종적인(global) 해답을 만들었다고 해서, 그 해답이 궁극적으로 최적이라는 보장이 없다.
- 따라서 탐욕적인 알고리즘은 *항상 최적의 해답을 주는지를 반드시 검증해야 한다.*
- 일반적으로 이와같은 탐욕적인 접근방법은 다음과 같은 절차로 진행이 된다.
  1. **선정과정(selection procedure)** - 현재 상태에서 가장 좋으리라고 생각되는(greedy) 해답을 찾아서 해답모음(solution set)에 포함시킨다.
  2. **적정성점검(feasibility check)** - 새로 얻은 해답모음이 적절한지를 결정한다.
  3. **해답점검(solution check)** - 새로 얻은 해답모음이 최적의 해인지를 결정한다.
- 보기: 거스름돈 문제
  - ▷ 문제: 동전의 갯수가 최소가 되도록 거스름 돈을 주는 문제
  - ▷ 탐욕적인 알고리즘: 거스름돈을 x라 하자. 먼저, 가치가 가장 높은 동전을 x가 초과되지 않도록 계속 내어 준다. 이 과정을 가치가 높은 동전 부터 내림 순으로 총액이 정확히 x가 될때까지 계속한다.
  - ▷ 현재 우리나라에서 유통되고 있는 동전 만을 가지고, 이 알고리즘을 적용하여 거스름돈을 주면, 항상 동전의 갯수는 최소가 된다. 따라서 이 알고리즘은 최적(optimal)!
  - ▷ 그러나, 만약 12원 짜리 동전을 새로 발행할 경우에는 이 알고리즘을 적용하여 거스름돈을 주면, 항상 동전의 갯수는 최소가 된다는 보장이 없다. 예를 들어, 거슬러 주어야할 돈의 액수가 16원이고, 이 알고리즘을 적용하면: 12원 $\times$ 1, 1원 $\times$ 4 와 같이 거슬러 주어야 할 동전의 갯수가 5개인 것으로 해답이 나오지만, 이는 최적(optimal)이 아니다. 최적의 해는 10원 $\times$ 1, 5원 $\times$ 1, 1원 $\times$ 1 이 되어 동전의 갯수는 3개가 된다.

제 1 절 최소비용 신장나무

- 그래프이론(graph theory) 용어: 비방향성 그래프 (undirected graph) $G = (V, E)$, 여기서 V는 마디(vertex)의 집합이고 E는 이음선(edge)의 집합이다. 경로(path). 연결된 그래프 (connected graph) - 어떤 두 마디 사이에도 경로가 존재한다. 부분그래프(subgraph), 가중치가 포함된 그래프(weighted graph). 순환경로(cycle), 순환적그래프(cyclic graph), 비순환적그래프(acyclic graph). 나무(tree) - 비순환적이며, 연결된, 비방향성 그래프. 뿌리 있는 나무(rooted tree) - 한 마디가 뿌리로 지정된 나무.
- 139쪽의 Figure 4.3을 참조하시오.
- 연결된, 비방향성 그래프 G에서 순환경로를 제거하면서 연결된 부분그래프가 되도록 이음선을 제거하면 신장나무(spanning tree)가 된다. 따라서 신장나무는 G 안에 있는 모든 마디를 다 포함하면서 나무가 되는 연결된 부분그래프이다.
- 신장나무가 되는 G의 부분그래프 중에서 가중치가 최소가 되는 부분그래프를 **최소비용 신장나무 (minimum spanning tree)** 라고 한다. 여기서 최소의 가중치를 가진 부분그래프는 반드시 나무가 되어야 한다. 왜냐하면, 만약 나무가 아니라면, 분명히 순환경로(cycle)가 있을 것이고, 그렇게 되면 순환경로 상의 한 이음선을 제거하면 더 작은 비용의 신장나무가 되기 때문이다.
- 관찰: 모든 신장나무가 최소비용신장나무는 아니다.
- 보기:
  - ▷ 도로건설 - 도시들을 모두 연결하면서 도로의 길이가 최소가 되도록 하는 문제
  - ▷ 통신(telecommunications) - 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성하는 문제
  - ▷ 배관(plumbing) - 파이프의 총 길이가 최소가 되도록 연결하는 문제
- 주먹구구식 알고리즘: 모든 신장나무를 다 고려해 보고, 그 중에서 최소비용이 드는 것을 고른다. 이는 최악의 경우, 지수보다도 나쁘다. 왜 그런지 생각해 보시오.

1.1 최소비용 신장나무를 구하기 위한 탐욕적인 알고리즘

- 문제: 비방향성 그래프 $G=(V,E)$가 주어졌을 때, $F \subseteq E$를 만족하면서, (V,F)가 G의 최소비용신장나무(MST)가 되는 F를 찾는 문제.
- 알고리즘:
  1. $F := \emptyset$;
  2. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복하라
     (a) **선정 절차:** 적절한 최적해 선정절차에 따라서 하나의 이음선을 선정
     (b) **적정성 점검:** 선정한 이음선을 F에 추가시켜도 순환경로가 생기지 않으면, F에 추가시킨다.
     (c) **해답 점검:** $T=(V,F)$가 신장나무이면, T가 최소비용 신장나무이다.
- 세부적인 알고리즘은 선정절차가 어떻게 되는가에 따라서 차이가 있을 수 있다. 다음에 선정절차가 다른 2가지 대표적인 알고리즘을 소개한다.

1.2 Prim의 알고리즘

1.2.1 추상적인 알고리즘

1. $F := \emptyset$;
2. $Y := \{v_1\}$;
3. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복하라
   (a) **선정 절차/적정성 점검:** $V-Y$에 속한 마디 중에서, Y에 가장 가까운 마디 하나를 선정한다
   (b) 선정한 마디를 Y에 추가한다
   (c) Y로 이어지는 이음선을 F에 추가한다
   (d) **해답 점검:** $Y=V$가 되면, $T=(V,F)$가 최소비용 신장나무 이다.

1.2.2 세부적인 알고리즘

- 그래프의 인접행렬식 표현

$$W[i][j] = \begin{cases} \text{이음선의 가중치} & v_i \text{ 에서 } v_j \text{ 로의 이음선이 있다면} \\ \infty & v_i \text{ 에서 } v_j \text{ 로의 이음선이 없다면} \\ 0 & i=j \text{ 이면} \end{cases}$$

- 추가적으로 `nearest[1..n]`과 `distance[1..n]` 배열 유지
  `nearest[i]` $=Y$에 속한 마디 중에서 v_i에서 가장 가까운 마디의 인덱스
  `distance[1..n]` $=v_i$와 `nearest[i]`를 잇는 이음선의 가중치

```
void prim(int n,              // 입력: 마디의 수
          const number W[][], // 입력: 그래프의 인접행렬식 표현
          set_of_edges& F)    // 출력: 그래프의 MST에 속한 이음선의 집합
{
   index i, vnear;
   number min;
   edge e;
   index nearest[2..n];
   number distance[2..n];

   F = empty_set;
   for (i = 2; i <= n; i++) {  // 초기화
      nearest[i] = 1;          // vi에서 가장 가까운 마디를 v1으로 초기화
      distance[i] = W[1][i];   // vi과 v1을 잇는 이음선의 가중치로 초기화
   }
   repeat (n-1 times) {        // n-1개의 마디를 Y에 추가한다
      min = "infinite";
      for (i = 2; i <= n; i++)            // 각 마디에 대해서
         if (0 <= distance[i] <= min) {   // distance[i]를 검사하여
            min = distance[i];            // 가장 가까이 있는 마디(vnear)를
            vnear = i;                    // 찾는다.
         }
      e = vnear와 nearest[vnear]를 잇는 이음선;
      e를 F에 추가;
      distance[vnear] = -1;               // 찾은 마디를 Y에 추가한다.
      for (i = 2; i <= n; i++)            //
         if (W[i][vnear] < distance[i]) { // Y에 없는 각 마디에 대해서
            distance[i] = W[i][vnear];    // distance[i]를 갱신한다.
```

```
                nearest[i] = vnear;
            }
        }
    }
```

1.2.3 분석

- 기본동작: repeat 맴돌이 안에 있는 두 개의 for 맴돌이 내부에 있는 명령문
- 입력크기: 마디의 개수, n
- 분석: repeat 맴돌이가 $n-1$번 반복되므로 $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

1.2.4 최적여부의 검증(Optimality Proof)

- Prim의 알고리즘이 찾아낸 신장나무가 최소비용(minimal)인지를 검증해야 한다. 다시 말하면, Prim의 알고리즘이 최적(optimal)인지를 보여야 한다.
- **정의 4.1:** 비방향성 그래프 $G = (V,E)$가 주어지고, 만약 E의 부분집합 F에 MST가 되도록 이음선을 추가할 수 있으면, F는 유망하다(promising)라고 한다.
- **레마 4.1:** $G = (V,E)$는 연결되고, 가중치를 포함한, 비방향성 그래프라고 하고, F는 E의 유망한 부분집합이라고 하고, Y는 F 안에 있는 이음선들에 의해서 연결이 되어있는 마디의 집합이라고 하자. 이때, Y에 있는 어떤 마디와 $V-Y$에 있는 어떤 마디를 잇는 이음선 중에서 가중치가 가장 작은 이음선을 e라고 하면, $F \cup \{e\}$는 유망하다.

  증명: F가 유망하기 때문에 $F \subseteq F'$ 이면서 (V,F')가 최소비용 신장나무(MST)가 되는 이음선 F'가 반드시 존재한다.

  - ▷ 경우 1: 만일 $e \in F'$라면, $F \cup \{e\} \subseteq F'$가 되고, 따라서 $F \cup \{e\}$도 유망하다.
  - ▷ 경우 2: 만일 $e \notin F'$라면, (V,F')는 신장나무이기 때문에, $F' \cup \{e\}$는 반드시 순환경로를 하나 포함하게 되고, e는 반드시 그 순환경로 가운데 한 이음선이 된다. 그러면 Y에 있는 한 마디에서 $V-Y$에 있는 한 마디를 연결하는 어떤 다른 이음선 $e' \in F'$가 그 순환경로 안에 반드시 존재하게 된다. 여기서 만약 $F' \cup \{e\}$에서 e'를 제거하면, 그 순환경로는 없어지게 되며, 다시 신장나무가 된다. 그런데 e는 Y에 있는 한 마디에서 $V-Y$에 있는 한 마디를 연결하는 최소의 가중치(weight)를 가진 이음선이기 때문에, e의 가중치는 반드시 e'의 가중치 보다 작거나 같아야 한다. (실제로는 반드시 같게 된다.) 그러면 $F' \cup \{e\} - \{e'\}$는 최소비용 신장나무(MST) 이다. 결론적으로 e'는 F 안에 절대로 속할 수 없으므로 (F 안에 있는 이음선들은 Y안에 있는 마디들 만을 연결함을 기억하라), $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$가 되고, 따라서 $F \cup \{e\}$는 유망하다.

- Prim의 알고리즘은 항상 최소비용 신장나무를 만들어 낸다.

  증명(수학적귀납법): 매번 반복이 수행된 후에 집합 F가 유망하다는 것을 보이면 된다

  - ▷ 출발점: 공집합은 당연히 유망하다
  - ▷ 귀납가정: 어떤 주어진 반복이 이루어진 후, 그때까지 선정하였던 이음선의 집합인 F가 유망하다고 가정한다
  - ▷ 귀납절차: 집합 $F \cup \{e\}$가 유망하다는 것을 보이면 된다. 여기서 e는 다음 단계의 반복 수행시 선정된 이음선이다. 그런데, 위의 레마 1에 의하여 $F \cup \{e\}$은 유망하다고 할 수 있다. 왜냐하면 이음선 e는 Y에 있는 어떤 마디를 $V-Y$에 있는 어떤 마디로 잇는 이음선 중에서 최소의 가중치를 가지고 있기 때문이다

  증명 끝.

1.3 Kruskal의 알고리즘

1.3.1 추상적인 알고리즘

- 알고리즘

  1. $F := \emptyset$;
  2. 서로소(disjoint)가 되는 V의 부분집합들을 만드는데, 각 부분집합마다 하나의 마디만 가지도록 한다
  3. E 안에 있는 이음선을 가중치의 비내림차순으로 정렬한다
  4. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복하라
     (a) **선정절차:** 다음 이음선을 선정한다. (최소의 가중치를 가진 이음선을 선정)
     (b) **적정성 점검:** 만약 선정된 이음선이 두 개의 서로소인 마디를 잇는다면, 먼저 그 부분집합을 하나의 집합으로 합하고, 그 다음에 그 이음선을 F에 추가한다
     (c) **해답점검:** 만약 모든 부분집합이 하나의 집합으로 합하여지면, 그때 $T = (V,F)$가 최소비용 신장나무이다.

1.3.2 세부적인 알고리즘

- 서로소 집합 추상데이타타입(disjoint set abstract daya type): 교재 부록 C 참조

```
    index i;
    set_pointer p, q;

    initial(n): n개의 서로소 부분집합을 초기화
                (하나의 부분집합에 1에서 n사이의 인덱스가 정확히 하나 포함됨)
    p = find(i): 인덱스 i가 포함된 집합의 포인터 p를 넘겨줌
    merge(p,q): 두 개의 집합을 가리키는 p와 q를 합병
    equal(p,q): p와 q가 같은 집합을 가리키면 true를 넘겨줌

void kruskal(int n, int m,       // 입력: 마디의 수 n, 이음선의 수 m
             set_of_edges E,     // 입력: 가중치를 포함한 이음선의 집합
             set_of_edges& F)    // 출력: MST를 이루는 이음선의 집합
{
   index i,j;
   set_pointer p,q;
   edge e;

   E에 속한 m개의 이음선을 가중치의 비내림차순으로 정렬;
   F = emptyset;
   initial(n);
   while (F에 속한 이음선의 개수가 n-1보다 작다) {
      e = 아직 점검하지 않은 최소의 가중치를 가진 이음선;
      i, j = e를 이루는 양쪽 마디의 인덱스;
      p = find(i);
      q = find(j);
      if (!euqal(p,q)) {
         merge(p,q);
         e를 F에 추가;
      }
   }
}
```

1.3.3 분석

- 기본동작: 비교문
- 입력크기: 마디의 수 n과 이음선의 수 m

  1. 이음선들을 정렬하는데 걸리는 시간: $\Theta(m \lg m)$
  2. 반복문 안에서 걸리는 시간: 맴돌이를 m번 수행한다. 서로소인 집합 자료구조(disjoint set data structure)를 사용하여 구현하고, find, equal, merge 같은 동작을 호출하는 횟수가 상수이면, m번 반복에 대한 시간복잡도는 $\Theta(m \lg m)$ 이다.
  3. n개의 서로소인 집합(disjoint set)을 초기화하는데 걸리는 시간: $\Theta(n)$

- 그런데 여기서 $m \geq n-1$이기 때문에, 위의 1과 2는 3을 지배하게 되므로.

$$W(m,n) = \Theta(m \lg m)$$

  가 된다.

- 그러나, 최악의 경우에는, 모든 마디가 다른 모든 마디와 연결이 될 수 있기 때문에, $m = \frac{n(n-1)}{2} \in \Theta(n^2)$ 가 된다. 그러므로, 최악의 경우의 시간복잡도는

$$W(m,n) \in \Theta(n^2 \lg n^2) = \Theta(2n^2 \lg n) = \Theta(n^2 \lg n)$$

1.3.4 최적여부의 검증(Optimality Proof)

- Prim의 알고리즘의 경우와 비슷함. (교재 151-152쪽 참조)

1.4 두 알고리즘의 비교

	$W(m,n)$	sparse graph	dense graph
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Kruskal	$\Theta(m \lg m)$ and $\Theta(n^2 \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2 \lg n)$

- 연결된 그래프에서의 m은 $n-1 \leq m \leq \dfrac{n(n-1)}{2}$의 범위를 갖는다.

1.5 토론사항

- 알고리즘의 시간복잡도는 그 알고리즘을 구현하는데 사용하는 자료구조에 좌우되는 경우도 있다.

Prim의 알고리즘	$W(m,n)$	sparse graph	dense graph
Heap	$\Theta(m\lg n)$	$\Theta(n\lg n)$	$\Theta(n^2\lg n)$
Fibonacci heap	$\Theta(m+n\lg n)$	$\Theta(n\lg n)$	$\Theta(n^2)$

제 2 절 단일출발점 최단경로 문제: Dijkstra의 알고리즘

- 가중치가 있는 방향성 그래프에서 한 특정 마디에서 다른 모든 마디로 가는 최단거리 구하는 문제.

2.1 알고리즘

- 알고리즘

  1. $F := \emptyset$;
  2. $Y := \{v_1\}$;
  3. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복하라
     (a) **선정절차/적정성 점검:** $V-Y$에 속한 정점 중에서, v_1에서 Y에 속한 정점 만을 거쳐서 최단거리가 되는 정점 v를 선정한다
     (b) 그 정점 v를 Y에 추가한다
     (c) v에서 F로 이어지는 최단거리 상의 이음선을 F에 추가한다
     (d) **해답점검:** $Y = V$가 되면, $T = (V,F)$가 최단거리를 나타내는 그래프 이다.

2.2 분석

- $T(n) \in \Theta(n^2)$. 힙(heap)으로 구현하면 $\Theta(m\lg n)$ 이고, 피보나찌 힙으로 구현하면 $\Theta(m+n\lg n)$ 이다.

2.3 최적여부의 검증(Optimality Proof)

- Prim의 알고리즘의 경우와 비슷함.

제 3 절 배낭 채우기 문제

- "도선생"의 배낭 채우기 문제(the knapsack problem)

3.1 탐욕적인 접근방법과 동적계획법의 비교

탐욕적인 접근방법	동적계획법
최적화 문제를 푸는데 적합	최적화 문제를 푸는데 적합
알고리즘이 존재할 경우 보통 더 효율적	때로는 불필요하게 복잡
알고리즘이 최적인지를 증명해야 함	최적화 원칙이 적용되는지를 점검해 보기만 하면 됨
단일출발점 최단거리 문제: $\Theta(n^2)$	단일출발점 최단거리 문제: $\Theta(n^3)$
배낭 빈틈없이 채우기 문제는 풀지만, 0-1 배낭 채우기 문제는 풀지 못함	0-1 배낭 채우기 문제를 푼다

3.2 0-1 배낭 채우기 문제의 탐욕적인 접근방법

- 문제:

$$S = \{item_1, item_2, \ldots, item_n\}$$
$$\omega_i = item_i \text{ 의 무게}$$
$$p_i = item_i \text{ 의 가치}$$
$$W = \text{배낭에 넣을 수 있는 총 무게}$$

  라고 할때, $\sum_{item_i \in A} \omega_i \le W$를 만족하면서 $\sum_{item_i \in A} p_i$가 최대가 되도록 $A \subseteq S$가 되는 A를 결정하는 문제이다.

- 주먹구구식 알고리즘:
  - ▷ n개의 물건에 대해서 모든 부분집합을 다 고려한다
  - ▷ 그러나 불행하게도 크기가 n인 집합의 부분집합의 수는 2^n개 이다.
- 탐욕적인 알고리즘:
  - ▷ 가장 비싼 물건 부터 우선적으로 채운다.

▷ 애석하게도 이 알고리즘은 최적이 아니다!

▷ 왜 아닌지 보기: $W = 30kg$

품목	무게	값
$item_1$	$25kg$	10만원
$item_2$	$10kg$	9만원
$item_3$	$10kg$	9만원

⋆ 탐욕적인 전법: $item_1 \Rightarrow 25kg \Rightarrow$ 10만원

⋆ 최적인 해답: $item_2 + item_3 \Rightarrow 20kg \Rightarrow$ 18만원

- 좀 더 세련된 탐욕적인 알고리즘:

  ▷ 무게당 가치가 가장 높은 물건 부터 우선적으로 채운다.

  ▷ 그래도 최적이 아니다!

  ▷ 왜 아닌지 보기: $W = 30kg$

품목	무게	값	값어치
$item_1$	$5kg$	50만원	10만원/kg
$item_2$	$10kg$	60만원	6만원/kg
$item_3$	$20kg$	140만원	7만원/kg

⋆ 탐욕적인 전법: $item_1 + item_3 \Rightarrow 25kg \Rightarrow$ 190만원

⋆ 최적인 해답: $item_2 + item_3 \Rightarrow 30kg \Rightarrow$ 200만원

3.3 배낭 빈틈없이 채우기 문제(The Fractional Knapsack Problem)

- 물건의 일부분을 잘라서 담을 수 있다
- 탐욕적인 접근방법으로 최적해를 구하는 알고리즘을 만들 수 있다.
- $item_1 + item_3 + item_2 \times \frac{1}{2} \Rightarrow 30kg \Rightarrow$ 220만원
- 최적이다! 증명해 보라. 숙제?

3.4 0-1 배낭채우기 문제의 동적계획적인 접근방법

- $i > 0$ 이고 $\omega > 0$일때, 전체 무게가 ω가 넘지 않도록 i번째 까지의 항목 중에서 얻어진 최고의 이익(optimal profit)을 $P[i][\omega]$라고 하면,

$$P[i][\omega] = \begin{cases} \max(P[i-1][\omega], p_i + P[i-1][\omega - \omega_i]) & \text{if } \omega_i \le \omega \\ P[i-1][\omega] & \text{if } \omega_i > \omega \end{cases}$$

  여기서 $P[i-1][w]$는 i번째 항목을 포함시키지 않는 경우의 최고 이익이고, $p_i + P[i-1][w-w_i]$는 i번째 항목을 포함시키는 경우의 최고 이익이다. 위의 되부름 관계식이 최적화 원칙을 만족하는지는 쉽게 알수있다.

- 그러면 어떻게 $P[n][W]$ 값을 구할 수 있을까? 다음과 같은 2차원 배열을 만든 후, 각 항을 계산하여 채워 넣으면 된다: `int P[0..n][0..W]`. 여기서 $P[0][w] = 0$, $P[i][0] = 0$으로 놓으면 되므로, 계산해야 할 항목의 수는 $nW \in \Theta(nW)$.
- 여기서 n과 W와는 아무런 상관관계가 없다. 따라서, $W = n!$이라고 한다면 수행시간은 $\Theta(n \times n!)$이 된다. 그렇게 되면 이 알고리즘은 앞에서 얘기한 주먹구구식 알고리즘보다도 나을게 하나도 없다.
- 그럼 이 알고리즘을 최악의 경우에 $\Theta(2^n)$ 시간에 수행될 수 있도록, 즉 주먹구구식 알고리즘 보다 느리지 않고, 때로는 훨씬 빠르게 수행될 수 있도록 개량할 수 있을까? 착안점은 $P[n][W]$를 계산하기 위해서 $(n-1)$번째 행을 모두 계산할 필요가 없다는데 있다. 즉, $P[n-1][W]$와 $P[n-1][W-w_n]$ 두 항만 계산하면 된다. 이런 식으로 $n = 1$이나 $w \le 0$일때까지 계속 뒤로 계산해 나가면 된다.
- 예로서 위의 예에서 $P[3][30]$을 계산해 보자. 개량 알고리즘은 다음과 같이 7개 항만 계산하는데 비해서, 이전 알고리즘은 $3 \times 30 = 90$ 항을 계산해야 한다.

$$\begin{aligned} P[3][30] &= \max([2][30], 140 + P[2][10]) = \max(110, 140+60) = 200 \\ P[2][30] &= \max([1][30], 60 + P[1][20]) = \max(50, 60+50) = 110 \\ P[2][10] &= \max([1][10], 60 + P[1][0]) = \max(50, 60+0) = 60 \\ P[1][0] &= 0 \\ P[1][10] &= 50 \\ P[1][20] &= 50 \\ P[1][30] &= 50 \end{aligned}$$

- 그러면 개량 알고리즘의 최악의 경우 수행시간을 계산해 보자. $(n-i)$번째 행에서 기껏해야 2^i 항을 계산하므로, 총 계산하는 항 수는 $1+2+2^2+\cdots+2^{n-1} = 2^n - 1$이 된다. 따라서 $\Theta(2^n)$가 된다. 위의 두 가지 경우를 합하면 최악의 경우의 수행시간은 $O(minimum(2^n, nW))$ 이다.
- 분할정복 방법으로도 이 알고리즘을 설계할 수도 있고, 그 최악의 경우 수행시간은 $\Theta(2^n)$이다. 아직 아무도 이 문제의 최악의 경우 수행시간이 지수(exponential)보다 나은 알고리즘을 발견하지 못했고, 아직 아무도 그러한 알고리즘은 없다라고 증명한 사람도 없다. – NP 문제

제 5 장

되추적

제 1 절 되추적(Backtracking) 기술

- **트리에서 깊이우선검색(depth-first search)**: 뿌리(root)가 되는 마디(node)를 먼저 방문한 뒤, 그 마디의 모든 후손(descendant)들을 차례로 (보통 왼쪽에서 오른쪽으로) 방문한다 (= preorder tree traversal). 교재 178쪽의 Figure 5.1 참조.

```
void depth_first_tree_search (node v)
{
   node u;

   visit v;
   for (each child u of v)
      depth_first_tree_search(u);
}
```

- **4-Queens 문제**
  - 4개의 Queen을 서로 상대방을 위협하지 않도록 4×4 서양장기(chess) 판에 위치시키는 문제이다. 서로 상대방을 위협하지 않기 위해서는 같은 행이나, 같은 열이나, 같은 대각선 상에 위치하지 않아야 한다.
  - 주먹구구식 알고리즘: 각 Queen을 각각 다른 행에 할당한 후에, 어떤 열에 위치하면 해답은 얻을 수 있는지를 차례대로 점검해 보면 된다. 이때, 각 Queen은 4개의 열 중에서 한 열에 위치할 수 있기 때문에, 해답을 얻기 위해서 점검해 보아야 하는 모든 경우의 수는 $4 \times 4 \times 4 \times 4 = 256$ 가지가 된다.
  - 가능한 경우를 모두 나열하기 위해서 교재 179쪽의 그림 5.2와 같이 상태공간트리(state space tree)를 구축할 수 있다. 뿌리에서 잎(leaf)까지의 경로는 해답후보(candidate solution)가 되는데, 깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다. 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는 마디의 후손(descendant)들도 모두 검색해야 하므로 비효율적이다.
  - 더 빨리 해답을 찾기 위해서, 가능성이 전혀 없는 후손을 가지고 있는 마디에 표시를 하여 검색을 못하게 하면, 훨씬 더 효율적으로 해답을 찾을 수 있을 것이다.
- **되추적 알고리즘(backtracking algorithm)**
  - 마디의 유망성: 전혀 해답이 나올 가능성이 없는 마디는 유망하지 않다(non-promising)고 하고, 그렇지 않으면 유망하다(promising)고 한다.
  - 따라서 되추적이란 어떤 마디의 유망성을 점검한 후, 유망하지 않다고 판정이 되면 그 마디의 부모(parent) 마디로 돌아가서("backtrack") 다음 후손에 대한 검색을 계속하게 되는 절차이다.
  - 결국 되추적 알고리즘은 위에서 언급한 상태공간트리에서 깊이우선검색을 실시하는데, 유망하지 않은 마디들은 가지쳐서(pruning) 검색을 하지 않으며, 유망한 마디에 대해서만 그 마디의 자식마디(children)를 검색한다. 이 알고리즘은 다음과 같은 절차로 진행된다.
    1. 상태공간트리의 깊이우선검색을 실시한다.
    2. 각 마디가 유망한지를 점검한다.
    3. 만일 그 마디가 유망하지 않으면, 그 마디의 부모마디로 돌아가서 검색을 계속한다.
  - **일반 되추적 알고리즘:**

```
void checknode (node v)
{
   if (promising(v))
      if (there is a solution at v)
         write the solution;
      else
         for (each child u of v)
            checknode(u);
}
```

- **4-Queens 문제 (계속)**
  - ▷ 되추적 알고리즘을 사용하여 가지친 상태공간트리를 구축하면 교재 182쪽의 Figure 5.4 같이 된다.
  - ▷ 여기서 순수한 깊이우선검색으로 해답을 얻기 위해서는 155개의 마디를 검색해야 하는데 비해서, 되추적을 적용하면 27개의 마디만 검색하면 된다.
- **개량된 되추적 알고리즘:**

```
void expand (node v)
{
   for (each child u of v)
      if (promising(u))
         if (there is a solution at u)
            write the solution;
      else
         expand(u);
}
```

이 개량된 알고리즘은 유망성 여부의 점검을 마디를 방문하기 전에 실시하므로, 그만큼 방문할 마디의 수가 적어져서 더 효율적이다. 그러나 일반 알고리즘이 이해하기는 더 쉽고, 일반 알고리즘을 개량된 알고리즘으로 변환하기는 간단하므로, 앞으로 이 강의에서의 모든 되추적 알고리즘은 일반 알고리즘과 같은 형태로 표시한다.

제 2 절 *n*-Queens 문제

- n개의 Queen을 서로 상대방을 위협하지 않도록 $n \times n$ 서양장기(chess) 판에 위치시키는 문제이다. 서로 상대방을 위협하지 않기 위해서는 같은 행이나, 같은 열이나, 같은 대각선 상에 위치하지 않아야 한다.
- ***n*-Queens 문제의 되추적 알고리즘**: 4-Queens 문제를 n-Queens 문제로 확장시키면 된다.
- **분석 1:** 상태공간트리 전체에 있는 마디의 수 를 구함으로서, 가지친 상태공간트리의 마디의 갯수의 상한을 구한다. 깊이가 i인 마디의 갯수는 n^i개 이고 이 트리의 깊이는 n이므로, 마디의 총 갯수의 상한값(upper bound)은:

$$1+n+n^2+n^3+\cdots+n^n = \frac{n^{n+1}-1}{n-1}$$

따라서 $n=8$일때, $\frac{8^9-1}{8-1} = 19,173,961$. 그러나 이 분석은 별 가치가 없다. 왜냐하면 되추적함으로서 점검하는 노드 수를 얼마나 줄였는지 상한값을 구해서는 전혀 알 수 없기 때문이다.

- **분석 2:** 유망한 마디만 세어서 상한값을 구한다. 이 값을 구하기 위해서는 어떤 두개의 Queen이 같은 열 상에 위치할 수 없다는 사실을 이용하면 된다. 예를들어 $n=8$일 경우를 생각해 보자. 첫번째 Queen은 어떤 열에도 위치시킬 수 있고, 두번째는 기껏해야 남은 7열 중에서만 위치시킬 수 있고, 세번째는 남은 6열 중에서 위치시킬 수 있다. 이런 식으로 계속했을 경우 마디의 수는 $1+8+8\times7+8\times7\times6+\cdots+8! = 109,601$ 가 된다. 이 결과를 일반화 하면 유망한 마디의 수는

$$1+n+n(n-1)+n(n-1)(n-2)+\cdots+n!$$

을 넘지 않는다.

- 위 2가지 분석 방법은 알고리즘의 복잡도를 정확히 얘기 해주지 못하고 있다. 왜냐하면:
  - ▷ 대각선을 점검하는 경우를 고려하지 않았다. 따라서 실제 유망한 마디의 수는 훨씬 더 작을 수 있다.
  - ▷ 유망하지 않은 마디를 포함하고 있는데, 실제로 해석의 결과에 포함된 마디 중에서 유망하지 않은 마디가 훨씬 더 많을 수 있다.
- **분석 3:** 유망한 마디의 갯수를 정확하게 구하기 위한 유일한 방법은 실제로 알고리즘을 수행하여 구축된 상태공간트리의 마디의 갯수를 세어보는 수 밖에 없다. 그러나 이 방법은 진정한 해석 방법이 될 수 없다. 왜냐하면 해석은 알고리즘을 실제로 수행하지 않고 이루어져야 하기 때문이다. 188쪽의 표 5.1을 보고 각 알고리즘의 수행시간을 비교해 보시오.
- **타당성 있는 분석:** 어떤 특정 입력이 주어졌을때, Monte Carlo 기법(확률적 알고리즘)을 이용하여 되추적 알고리즘의 수행시간을 추정할 수 있다.

제 3 절 Monte Carlo 기법을 사용한 백트랙킹 알고리즘의 수행시간 추정

- Monte Carlo 기법은 어떤 입력이 주어졌을 때 점검하게 되는 상태공간트리의 "전형적인" 경로를 무작위(random)로 생성하여 그 경로 상에 있는 마디의 수를 센다. 이 과정을 여러번 반복하여 나오는 결과의 평균치를 추정치로 한다.
- 이 기법을 적용하기 위해서는 다음 두 조건을 반드시 만족하여야 한다.
  - ▷ 상태공간트리의 같은 수준(level)에 있는 모든 마디의 유망성 여부를 점검하는 절차는 같아야 한다.
  - ▷ 상태공간트리의 같은 수준에 있는 모든 마디는 반드시 같은 수의 자식마디를 가지고 있어야 한다.

n-Queens 문제는 이 두조건을 만족한다.

- **Monte Carlo 기법의 절차:**
  1. 뿌리의 유망한 자식마디의 갯수를 m_0 이라고 한다.
  2. 상태공간트리의 수준 1에서 유망한 마디를 하나 랜덤하게 정하고, 이 마디의 유망한 자식마디의 갯수를 m_1 이라고 한다.
  3. 위에서 정한 마디의 유망한 마디를 하나 랜덤하게 정하고, 이 마디의 유망한 자식마디의 갯수를 m_2 라고 한다.
  4. $\cdots$
  5. 더 이상 유망한 자식마디가 없을 때 까지 이 과정을 반복한다.

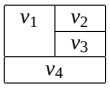
  여기서 m_i는 수준 i에 있는 마디의 유망한 자식마디의 갯수의 평균의 추정치이다. 수준 i에 있는 한 마디의 자식마디의 총 갯수를 t_i라고 하면 (유망하지 않은 마디도 포함), 백트랙킹 알고리즘에 의해서 점검한 마디의 총 갯수의 추정치는

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots + m_0 m_1 \cdots m_{i-1} t_i + \cdots$$

  가 된다.

제 4 절 그래프 색칠하기

- **지도에 m가지 색으로 색칠하는 문제(Graph Coloring):** m개의 색을 가지고, 인접한 지역이 같은 색이 되지 않도록 지도에 색칠하는 문제
- 평면그래프(planar graph)란?: 평면 상에서 이음선(edge)들이 서로 엇갈리지 않게 그릴 수 있는 그래프. 예로서 교재 201쪽의 Figure 5.11을 보시오.
- 지도에서 각 지역을 그래프의 정점으로 하고, 한지역이 어떤 다른 지역과 인접해 있으면 그 지역들을 나타내는 정점들 사이에 이음선을 그으면, 모든 지도는 그에 상응하는 평면그래프로 표시할 수 있다. 따라서 이 문제는 평면그래프를 사용하여 푼다.
- **보기:**



  이 지도는 교재 200쪽의 Figure 5.10의 그래프로 나타낼 수 있다. 이 그래프에서 두가지 색으로 문제를 풀기는 불가능하다. 세가지 색을 사용하면 총 6가지의 해답을 얻을 수 있다.
- 이 문제는 되추적 알고리즘을 사용하여 효과적으로 풀 수 있다. 위의 보기에 대한 가지친 상태공간트리를 그려서 해답을 구해 보시오.
- **분석:** 상태공간트리 상의 마디의 총수는

$$1 + m + m^2 + \cdots + m^m = \frac{m^{m+1} - 1}{m - 1}$$

  가 된다. 여기서도 Monte Carlo 기법을 사용하여 수행시간을 추정할 수 있다.

제 5 절 해밀토니안 회로 문제

- 연결된 비방향성 그래프에서, 해밀토니안 회로(Hamiltonian Circuits)/ 여행경로(tour)는 어떤 한 마디에서 출발하여 그래프 상의 각 마디를 한번씩 만 경유하여 다시 출발한 마디로 돌아오는 경로이다. 예로서, 교재 205쪽의 그림 5.13을 보시오.
- **해밀토니안 회로 문제:** 연결된 비방향성 그래프에서 해밀토니안 회로를 결정하는 문제
- 되추적 방법을 적용하기 위해서 다음 사항을 고려해야 한다.
  - ▷ 경로 상의 i번째 마디는 그 경로 상의 $(i-1)$번째 마디와 반드시 이웃해야 한다.
  - ▷ $(n-1)$번째 마디는 반드시 0번째 마디(출발점)와 이웃해야 한다.
  - ▷ i번째 마디는 처음 $i-1$개의 마디가 될 수 없다.
- 상태공간트리 상의 마디의 수는

$$1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{(n-1)} = \frac{(n-1)^n - 1}{n-2}$$

제 6 장

분기한정법

제 1 절 분기한정법(Branch-and-Bound)

- 특징:
  - ▷ 되추적 기법과 같이 상태공간트리를 구축하여 문제를 해결한다.
  - ▷ 최적의 해를 구하는 문제(optimization problem)에 적용할 수 있다.
  - ▷ 최적의 해를 구하기 위해서는 어짜피 모든 해를 다 고려해 보아야 하므로 트리의 마디를 순회(traverse)하는 방법에 구애받지 않는다.
- 분기한정 알고리즘은 각 마디를 검색할 때 마다, 그 마디가 유망한지의 여부를 결정하기 위해서 한계치(bound)를 계산한다. 그 한계치는 그 마디로 부터 가지를 뻗어나가서(branch) 얻을 수 있는 해답치의 한계를 나타낸다. 따라서 만약 그 한계치가 지금까지 찾은 최적의 해답치 보다 좋지 않은 경우는 더 이상 가지를 뻗어서 검색을 계속할 필요가 없으므로, 그 마디는 유망하지 않다고 할 수 있다.

제 2 절 0-1 배낭채우기 문제(0-1 Knapsack Problem)

2.1 분기한정식 가지치기로 깊이우선검색 (= 되추적)

- 상태공간트리를 구축하여 되추적 기법으로 문제를 푼다: 뿌리에서 왼쪽으로 가면 첫번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 첫번째 아이템을 배낭에 넣지 않는 경우이다. 동일한 방법으로 트리의 수준 1에서 왼쪽으로 가면 두번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 그렇지 않는 경우이다. 이런 식으로 계속하여 상태공간트리를 구축하면, 뿌리로 부터 잎까지의 모든 경로는 해답후보가 된다.
- 이 문제는 최적의 해를 찾는 문제(optimization problem)이므로 검색이 완전히 끝나기 전에는 해답을 알 수가 없다. 따라서 검색을 하는 과정 동안 항상 그 때까지 찾은 최적의 해를 기억해 두어야 한다.
- **최적화 문제를 풀기 위한 일반적인 되추적 알고리즘:**

```
void checknode(node v)
{
   if (value(v) is better than best)
      best = value(v);
   if (promising(v))
      for (each child u of v)
         checknode(u);
}
```

  - ▷ `best`: 지금까지 찾은 제일 좋은 해답치.
  - ▷ `value(v)`: `v` 마디에서의 해답치.
- **알고리즘의 스켓치:**
  - ▷ Let:
    - ⋆ *profit* : 그 마디에 오기까지 넣었던 아이템의 값어치의 합.
    - ⋆ *weight*: 그 마디에 오기까지 넣었던 아이템의 무게의 합.

⋆ *bound*: 마디가 수준 i에 있다고 하고, 수준 k에 있는 마디에서 총무게가 W를 넘는다고 하자. 그러면

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \left(profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times \frac{p_k}{w_k}$$

⋆ *maxprofit*: 지금까지 찾은 최선의 해답이 주는 값어치

▷ w_i와 p_i를 각각 i 번째 아이템의 무게와 값어치라고 하면, p_i/w_i의 값이 큰 것 부터 내림차순으로 아이템을 정렬한다. (일종의 탐욕적인 방법이 되는 셈이지만, 알고리즘 자체는 탐욕적인 알고리즘은 아니다.)

▷ *maxprofit* := \$0; *profit* := \$0; *weight* := 0

▷ 깊이우선순위로 각 마디를 방문하여 다음을 수행한다:

1. 그 마디의 *profit* 와 *weight*를 계산한다.
2. 그 마디의 *bound*를 계산한다.
3. *weight* $< W$ and *bound* $>$ *maxprofit*이면, 검색을 계속한다;
   그렇지 않으면, 되추적.

▷ 고찰: 최선이라고 여겼던 마디를 선택했다고 해서 실제로 그 마디로 부터 최적해가 항상 나온다는 보장은 없다.

- 보기: $n = 4, W = 16$이고,

i	p_i	w_i	p_i/w_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

  일때, 되추적을 사용하여 구축되는 가지친 상태공간트리를 그려 보시오. (교재의 210쪽 참조)

- **분석:** 이 알고리즘이 점검하는 마디의 수는 $\Theta(2^n)$ 이다.
  위 보기의 경우의 분석: 점검한 마디는 13개 이다. 이 알고리즘이 동적계획법으로 설계한 알고리즘 보다 좋은가? 확실하게 대답하기 불가능 하다. Horowitz와 Sahni(1978)는 Monte Carlo 기법을 사용하여 되추적 알고리즘이 동적계획법 알고리즘 보다 일반적으로 더 빠르다는 것을 입증하였다. Horowitz와 Sahni(1974)가 분할정복과 동적계획법을 적절히 조화하여 개발한 알고리즘은 $O(2^{n/2})$의 시간복잡도를 가지는데, 이 알고리즘은 일반적으로 되추적 알고리즘 보다 일반적으로 빠르다고 한다.

2.2 분기한정식 가지치기로 너비우선검색

- 너비우선검색(Breadth-first Search)순서: (1) 뿌리를 먼저 검색한다, (2) 다음에 수준 1에 있는 모든 마디를 검색한다 (왼쪽에서 오른쪽으로), (3) 다음에 수준 2에 있는 모든 마디를 검색한다 (왼쪽에서 오른쪽으로), (4) 등등

- 일반적인 너비우선검색 알고리즘:
  되부름(recursive) 알고리즘을 작성하기는 상당히 복잡하다. 따라서 큐(queue)를 사용한다.

```
void breadth_first_tree_search(tree T)
{
   queue_of_node Q;
   node u,v;

   initialize(Q);
   v = root of T;
   visit v;
   enqueue(Q,v);
   while (!empty(Q)) {
      dequeue(Q,v);
      for (each child u of v) {
         visit u;
         enqueue(Q,u);
      }
   }
}
```

- 분기한정식 가지치기로 너비우선검색 알고리즘:

```
void breadth_first_branch_and_bound(state_space_tree T,
                                    number& best)
{
   queue_of_the_node Q;
   node u,v;
```

```
      initialize(Q);
      v = root of T;
      enqueue(Q,v);
      best = value(v);
      while (!empty(Q)) {
         dequeue(Q,v);
         for (each child u of v) {
            if (value(u) is better than best)
               best = value(u);
            if (bound(u) is better than best)
               enqueue(Q,u);
         }
      }
   }
```

- 보기: 앞에서와 같은 예를 사용하여 분기한정 가지치기로 너비우선검색을 하여 가지친 상태공간트리를 그려보면, 교재 226쪽의 그림과 같이 된다. 이때 검색하는 마디의 갯수는 17 이다. 되추적 알고리즘보다 좋지 않다!

2.3 분기한정식 가지치기로 최고우선검색(Best-First Search)

- 최고우선검색(Best-First Search)은 너비우선검색에 비해서 좋아짐
- 최적의 해답에 더 빨리 도달하기위한 전략:
  1. 주어진 마디의 모든 자식마디를 검색한 후,
  2. 유망하면서 확장되지 않은(unexpanded) 마디를 살펴보고,
  3. 그중에서 가장좋은(최고의) 한계치(bound)를 가진 마디를 확장한다.
- 분기한정식 가지치기로 최고우선검색 알고리즘:
  최고의 한계치를 가진 마디를 우선적으로 선택하기 위해서 우선순위가 있는 큐(priority queue)를 사용한다. 우선순위가 있는 큐는 힙(heap)을 사용하여 효율적으로 구현할 수 있다.

```
   void best_first_branch_and_bound(state_space_tree T,
                                    number best)
   {
      priority_queue_of_node PQ;
      node u,v;

      initialize(Q);
      v = root of T;
      best = value(v);
      insert(PQ,v);
      while (!empty(PQ)) {
         remove(PQ,v);
         if (bound(v) is better than best)
            for (each child u of v) {
               if (value(u) is better than best)
                  best = value(u);
               if (bound(u) is better than best)
                  insert(PQ,u);
            }
      }
   }
```

- 보기: 앞에서와 같은 예를 사용하여 분기한정 가지치기로 최고우선검색을 하여 가지친 상태공간트리을 그려보면, 교재 231쪽의 그림과 같이 된다. 이때 검색하는 마디의 갯수는 11 이다.

제 3 절 외판원 문제(Traveling Salesperson Problem)

- 교재의 123-130쪽의 Chapter 3.6를 보시오.
- 외판원의 집이 위치하고 있는 도시에서 출발하여 다른 도시들을 각각 한번씩만 방문하고, 다시 집으로 돌아오는 가장 짧은 여행경로(tour)를 결정하는 문제.
- 이 문제는 음이 아닌 가중치가 있는, 방향성 그래프로 나타낼 수 있다. 그래프 상에서 여행경로(tour, Hamiltonian circuits)는 한 마디를 출발하여 다른 모든 마디를 한번씩 만 거쳐서 다시 그 마디로 돌아오는 경로이다. 여러개의 여행경로 중에서 길이가 최소가 되는 경로가 최적여행경로(optimal tour)가 된다.
- 보기: 교재 124쪽의 Figure 3.16을 보시오. 가장 최적이 되는 여행길은?
- 무작정 알고리즘: 가능한 모든 길을 다 고려한 후, 그 중에서 가장 짧은 여행경로을 선택한다. 가능한 여행길의 총 갯수는 $(n-1)!$이다.

3.1 동적계획법을 이용한 접근방법

- V는 모든 마디의 집합이고, A는 V의 부분집합이라고 하자. 그리고 $D[v_i][A]$는 A에 속한 각 마디를 정확히 한번씩만 거쳐서 v_i에서 v_1로 가는 최단경로의 길이라고 하자. 그러면 위의 보기에서 $D[v_2][\{v_3, v_4\}]$의 값은? (= 20)
- 최적 여행경로의 길이:

$$D[v_1][V-\{v_1\}] = \min_{2\le j\le n}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

일반적으로 표시하면 $i \neq 1$이고, v_i가 A에 속하지 않을때, 다음과 같이 된다.

$$D[v_i][A] = \min_{v_j \in A}(W[i][j] + D[v_j][A-\{v_j\}]) \text{ if } A \neq \emptyset$$

$$D[v_i][\emptyset] = W[i][1]$$

- 연습으로 위의 보기의 그래프에서 최적여행경로를 찾아보시오. 즉, $D[v_1][\{v_2, v_3, v_4\}]$를 계산해 보시오.
- **알고리즘**
  - ▷ 문제: 가중치가 있는 방향성 그래프에서 최적여행경로를 결정하시오. 여기서 가중치는 음수아닌 정수이다.
  - ▷ 입력: 가중치가 있는 방향성 그래프와 그 그래프에 있는 마디의 갯수 n. 그래프는 행렬 W로 표시가 되는데, 여기서 $W[i][j]$는 v_i에서 v_j를 잇는 이음선 상에 있는 가중치를 나타낸다. V는 그래프 상의 모든 마디의 집합을 나타낸다.
  - ▷ 출력: 최적여행경로의 길이를 나타내는 값을 가지는 변수 *minlength*와 배열 P (이 배열로 부터 최적여행경로을 구축할 수 있다). $P[i][A]$는 A에속한 각 마디를 정확히 한번씩만 거쳐서 v_i에서 v_1로 가는 최단경로 상에서, v_i 다음의 도달하는 첫번째 마디의 인덱스이다.

```
void travel(int n,
            const number W[][],
            index P[][],
            number& minlength)
{
   index i,j,k;
   number D[1..n][subset of V-{v_1}];

   for (i = 2; i<= n; i++)
      D[i][emptyset] := W[i][1];
   for (k = 1; k <= n-2; k++)
      for (V-{v_1}의 부분집합 중에서 k개의 마디를 가진 모든 부분집합 A)
         for (i = 1 이 아니고 v_i가 A에 속하지 않는 모든 i) {
            D[i][A] = minimum_{v_j in A} (W[i][j] + D[v_j][A-{v_j}]);
            P[i][A] = value of j that gave the minimum;
         }
   D[1][V-{v_1}] = minimum_{2<=j<=n} (W[1][j] + D[v_j][V-{v_1}]);
   P[1][V-{v_1}] = value of j that gave the minimum;
   minlength = D[1][V-{v_1}];
}
```

- **정리:** $n \ge 1$를 만족하는 모든 n에 대해서,

$$\sum_{k=1}^{n} k \left[\begin{array}{c} n \\ k \end{array} \right] = n2^{n-1}$$

증명:

$$\left[\begin{array}{c} n \\ k \end{array} \right] = \frac{n!}{k!(n-k)!} = \frac{n}{k}\frac{(n-1)!}{(k-1)!(n-k)!} = \frac{n}{k}\frac{(n-1)!}{(k-1)!(n-1-(k-1))!} = \frac{n}{k}\left[\begin{array}{c} n-1 \\ k-1 \end{array} \right]$$

$$k\left[\begin{array}{c} n \\ k \end{array} \right] = k\frac{n}{k}\left[\begin{array}{c} n-1 \\ k-1 \end{array} \right] = n\left[\begin{array}{c} n-1 \\ k-1 \end{array} \right]$$

$$\sum_{k=1}^{n} k\left[\begin{array}{c} n \\ k \end{array} \right] = \sum_{k=1}^{n} n\left[\begin{array}{c} n-1 \\ k-1 \end{array} \right] = n\sum_{k=0}^{n-1}\left[\begin{array}{c} n-1 \\ k \end{array} \right] 1^k 1^{n-1-k} = n(1+1)^{n-1} = n2^{n-1}$$

- **분석:**
  - ▷ 단위연산: 중간에 위치한 루프가 수행시간을 지배한다. 왜냐하면 이 루프는 여러 겹으로 쌓여 있기 때문이다. 따라서, 기본동작은 v_j의 각 값에 대해서 수행되는 명령문이다 (덧셈하는 명령문 포함).
  - ▷ 입력의 크기: 그래프에서 정점의 갯수 n
  - ▷ 시간 복잡도: 알고리즘에서 두번째 for-맴돌이(loop)가 시간복잡도를 좌우한다.

```
    for (k = 1; k <= n-2; k++)
(1)   for (V-{v_1}의 부분집합 중에서 k개의 마디를 가진 모든 부분집합 A)
(2)     for (i = 1 이 아니고 v_i가 A에 속하지 않는 모든 i)
(3)       D[i][A] := minimum_{v_j in A} (W[i][j] + D[v_j][A-{v_j}]);
          P[i][A] := value of j that gave the minimum
```

(1)번 루프는 $\left[\begin{array}{c} n-1 \\ k \end{array} \right]$ 번 반복하고 ($n-1$개의 마디에서 k개를 뽑는 경우의 수), (2)번 루프는 $n-k-1$번 반복하고 (v_1을 제외하고 A에 속하지 않는 마디의 갯수), (3)번 루프는 A의 크기가 k이므로 k번 반복한다 (A에 속한 마디의 갯수). 따라서 시간복잡도는

$$T(n) = \sum_{k=1}^{n-2} (n-k-1)k \left[\begin{array}{c} n-1 \\ k \end{array} \right]$$

가 되고, 이를 위의 정리를 이용하여 풀면, $T(n) = (n-1)(n-2)2^{n-3} = \Theta(n^2 2^n)$가 된다. 자세한 풀이과정은 교재의 128-129쪽을 참조하시오.

- ▷ 공간복잡도: 배열 $D[v_i, A]$와 $P[v_i, A]$가 얼마나 커야하는지를 결정하면 된다. $V - \{v_1\}$는 $n-1$개의 마디를 가지고 있기 때문에, 이 배열은 2^{n-1}개의 부분집합 A를 가지고 있다. (n개의 아이템이 포함되어 있는 어떤 집합의 부분집합의 갯수는 2^n 이다.) 따라서 $M(n) = 2 \times n \times 2^{n-1} = n2^n \in \Theta(n2^n)$

- **비교:** $n = 20$ 일때,
  - ▷ 무작정 알고리즘: 각 여행경로의 길이를 계산하는데 걸리는 시간은 $1\mu sec$ 이라고 할때, $(20-1)! = 19!\mu sec =$ 3857년이 걸린다.
  - ▷ 동적계획법 알고리즘: 동적계획법 알고리즘의 기본동작을 수행하는데 걸리는 시간을 $1\mu sec$ 이라고 할때, $T(20) = (20-1)(20-2)2^{20-3}\mu sec = 45$초가 걸리고, $M(20) = 20 \times 2^{20} = 20,971,520$의 배열의 slot이 필요하다.

- 배열 P로 부터 어떻게 최적 여행경로를 구축할까요?

$$P[1, \{v_2, v_3, v_4\}] = 3 \Rightarrow P[3, \{v_2, v_4\}] = 4 \Rightarrow P[4, \{v_2\}] = 2$$

따라서 최적 여행경로는 $[v_1, v_3, v_4, v_2, v_1]$.

3.2 분기한정 접근법

- $n = 40$ 일때, 동적계획법 알고리즘은 6년 이상이 걸린다. 그러므로 분기한정법을 시도해 본다.
- **보기:** 다음 인접행렬로 표현된 그래프를 살펴보시오. (교재 237쪽 참조)

$$\left[\begin{array}{ccccc} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{array} \right]$$

- 어떻게 상태공간트리를 구축할까? (교재 238쪽의 Figure 6.5 참조)
  - ▷ 각 마디는 출발마디로 부터의 여행경로를 나타내게 되는데, 몇개 만 예를 들어 보면, 뿌리의 여행경로는 [1]이 되고, 뿌리에서 뻗어나가는 수준 1에 있는 마디들의 여행경로는 각각 [1,2], [1,3], …, [1,5]가 되고, 마디 [1,2]에서 뻗어나가는 수준 2에 있는 마디들의 여행경로는 각각 [1,2,3], …, [1,2,5]가 되고, 이런 식으로 뻗어나가서 잎마디에 도달하게 되면 완전한 여행경로를 가지게 된다.
  - ▷ 따라서 최적여행경로를 구하기 위해서는 잎마디에 있는 여행경로를 모두 검사하여 그 중에서 가장 길이가 짧은 여행경로를 찾으면 된다.
  - ▷ 참고: 위 예에서 각 마디에 저장되어 있는 마디가 4개가 되면 더 이상 뻗어 나갈 필요가 없다. 왜냐하면, 남은 경로는 더 이상 뻗어나가지 않고도 알 수 있기 때문이다.
- 분기한정식 가지치기로 최고우선 검색을 사용하기 위해서 각 마디의 한계치를 구할 수 있어야 한다. 이 문제에서는 주어진 마디에서 뻗어나가서 얻을 수 있는 여행경로의 길이의 하한(최소치)을 구하여 한계치로 한다. 그리고 각 마디를 검색할 때 최소여행경로의 길이 보다 한계치가 작은 경우 그 마디는 유망하다고 한다. 최소여행경로의 초기값은 ∞로 놓는다. 따라서 완전한 여행경로를 처음 얻을 때 까지는 한계치가 무조건 최소여행경로의 길이 보다 작게 되므로 모든 마디는 유망하다.
- 각 마디의 한계치는 어떻게 구하나?
  $[1, \ldots, k]$의 여행경로를 가진 마디의 한계치는 다음과 같이 구한다. Let:

$$A = V - ([1, \ldots, k] \text{ 경로에 속한 모든 마디의 집합})$$

$$\begin{array}{rcl} \text{bound} & = & [1, \ldots, k] \text{ 경로 상의 총거리} \\ & + & v_k \text{에서 } A \text{에 속한 마디로 가는 이음선의 길이들 중에서 최소치} \\ & + & \sum_{i \in A} (v_i \text{에서 } A \cup \{v_1\} - \{v_i\} \text{에 속한 마디로 가는 이음선의 길이들 중에서 최소치}) \end{array}$$

- 분기한정 가지치기로 최고우선검색을 하여 상태공간트리를 구축해 보시오. (교재 241쪽의 Figure 6.6 참조)
- 이 알고리즘은 방문하는 정점의 갯수가 더 적다. 그러나 아직도 알고리즘의 시간복잡도는 지수적이거나 그보다 못하다! 다시말해서 $n = 40$이 되면 문제를 풀 수 없는 것과 다름없다고 할 수 있다.
- 다른 방법이 있을까? – 근사(approximation) 알고리즘: 최적의 해답을 준다는 보장은 없지만, 무리없이 최적에 가까운 해답을 주는 알고리즘.

제 7 장

계산복잡도 개론: 정렬 문제

☞ 계산복잡도(Computational Complexity):

- 알고리즘의 분석: 어떤 특정 알고리즘의 효율도(efficiency)를 측정함 – 시간복잡도(time complexity), 공간복잡도(space/memory complexity)
- 문제의 분석: 일반적으로 "계산복잡도 분석"이란 이를 지칭하는데, 어떤 문제에 대해서 그 문제를 풀 수 있는 모든 알고리즘의 효율도의 하한선(lower-bound)을 결정한다.
- 보기: (행렬곱셈문제)
  - ▷ 일반알고리즘: $\Theta(n^3)$
  - ▷ Strassen의 알고리즘: $\Theta(n^{2.81})$
  - ▷ Coppersmith/Winograd의 알고리즘: $\Theta(n^{2.38})$
  - ▷ 이 문제의 복잡도의 하한선은 $\Omega(n^2)$
  - ▷ 더 빠른 알고리즘이 존재할까? 아직 이 하한선 만큼 좋은 알고리즘을 찾지 못하였고, 그렇다고 하한선이 이보다 더 큰지도 알아내지 못하였다.
- 복잡도의 하한선이 $\Omega(f(n))$인 문제에 대해서 복잡도가 $\Theta(f(n))$인 알고리즘을 만들어 내는 것이 목표이다. 다시말하면 문제의 복잡도 하한선보다 낮은 알고리즘을 만들어 낸다는 것은 불가능하다. (물론 상수적으로 알고리즘을 향상시키는 것은 가능하다.)
- 보기: 정렬문제(sorting)
  - ▷ 교환정렬(Exchange sort): $\Theta(n^2)$
  - ▷ 합병정렬(Mergesort): $\Theta(n\lg n)$
  - ▷ 정렬문제의 시간복잡도 하한선은 $\Omega(n\lg n)$ (키를 비교하여 정렬하는 경우에만 해당됨)
  - ▷ 이 정렬 문제의 경우는 하한선 만큼의 시간 복잡도를 가진 알고리즘을 찾았다.

제 1 절 삽입정렬

- 이미 정렬된 배열에 항목을 끼워 넣음으로서 정렬하는 알고리즘: 예) 8 4 2 7 9 5 1 3
- **알고리즘: 삽입정렬(Insertion Sort)**
  - ▷ 문제: 비내림차순으로 n개의 키를 정렬
  - ▷ 입력: 양의 정수 n; 키의 배열 S[1..n]
  - ▷ 출력: 비내림차순으로 정렬된 키의 배열 S[1..n]

```
void insertionsort(int n, keytype S[])
{
   index i,j;
   keytype x;

   for (i = 2; i <= n; i++) {
      x = S[i];
      j = i - 1;
      while (j > 0 && S[j] > x) {
         S[j+1] = S[j];
         j--;
      }
      S[j+1] = x;
   }
}
```

- **최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  i가 주어졌을때, while-맴돌이(loop)에서 최대한 $i-1$번의 비교가 이루어 진다. 그러면 비교하는 총 횟수는 최대한
  $$W(n) = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$
- **평균의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  i가 주어졌을때, x가 삽입될 수 있는 장소가 i개 있다.

삽입할 장소의 인덱스	i	$i-1$	$\cdots$	2	1
비교 횟수	1	2	$\cdots$	$i-1$	$i-1$

  x를 삽입하는데 필요한 평균 비교 횟수는:
  $$1\frac{1}{i} + 2\frac{1}{i} + \cdots + (i-1)\frac{1}{i} + (i-1)\frac{1}{i} = \frac{1}{i}\sum_{k=1}^{i-1}k + \frac{i-1}{i} = \frac{(i-1)i}{2i} + \frac{i-1}{i} = \frac{i+1}{2} - \frac{1}{i}$$
  따라서 정렬하는데 필요한 평균 비교 횟수는:
  $$\sum_{i=2}^{n}(\frac{i+1}{2} - \frac{1}{i}) = \sum_{i=2}^{n}\frac{i+1}{2} - \sum_{i=2}^{n}\frac{1}{i} \approx \frac{(n+4)(n-1)}{4} - \ln n \approx \frac{n^2}{4}$$
- 공간복잡도 분석: 저장장소가 추가로 필요하지 않다. 따라서 $M(n) = \Theta(1)$. — 제자리정렬(in-place sorting) = 추가로 소요되는 저장장소의 크기가 상수 인 경우.

제 2 절 선택정렬

- **알고리즘: 선택정렬(Selection Sort)**
  - ▷ 문제: 비내림차순으로 n개의 키를 정렬
  - ▷ 입력: 양의 정수 n; 키의 배열 S[1..n]
  - ▷ 출력: 비내림차순으로 정렬된 키의 배열 S[1..n]

```
void selectionsort(int n, keytype S[])
{
   index i,j,smallest;

   for (i = 1; i <= n-1; i++) {
      smallest = i;
      for (j = i+1; j <= n; j++)
         if (S[j] < S[smallest])
            smallest = j;
      exchange S[i] and S[smallest];
   }
}
```

- **모든경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  i가 1일때 비교횟수는 $n-1$, i가 2일때 비교횟수는 $n-2$, $\cdots$, i가 $n-1$일때 비교횟수는 1 이 된다. 이를 모두 합하면, $\frac{n(n-1)}{2}$ 이다. 따라서
  $$T(n) = \frac{n^2}{2}$$
- **모든경우 시간복잡도 분석 - 지정(assignment)하는 횟수를 기준:**
  1번 교환하는데 3번 지정하므로
  $$T(n) = 3(n-1)$$

제 3 절 $\Theta(n^2)$ 정렬알고리즘의 비교

- 비교 테이블:

알고리즘	비교횟수	지정횟수	추가저장장소사용량
교환정렬	$T(n) = \frac{n^2}{2}$	$W(n) = \frac{3n^2}{2}$ $A(n) = \frac{3n^2}{4}$	제자리정렬
삽입정렬	$W(n) = \frac{n^2}{2}$ $A(n) = \frac{n^2}{4}$	$W(n) = \frac{n^2}{2}$ $A(n) = \frac{n^2}{4}$	제자리정렬
선택정렬	$T(n) = \frac{n^2}{2}$	$T(n) = 3n$	제자리정렬

- 삽입정렬은 교환정렬 보다는 항상 최소한 빠르게 수행된다고 할 수 있다.
- 선택정렬이 교환정렬 보다 빠른가? – 일반적으로는 선택정렬 알고리즘이 빠르다고 할 수 있다. 그러나 입력이 이미 정렬되어 있는 경우, 선택정렬은 지정이 이루어지지만 교환정렬은 지정이 이루어지지 않으므로 교환정렬이 빠르다.
- 선택정렬 알고리즘이 삽입정렬 알고리즘 보다 빠른가? – n의 크기가 크고, 키의 크기가 큰 자료구조 일때는 지정하는 시간이 많이 걸리므로 선택정렬 알고리즘이 더 빠르다.

제 4 절 한번 비교하는데 최대한 하나의 역을 제거하는 알고리즘의 하한선

- n개의 별개의(다른) 키를 정렬하는데, 정렬할 키들을 단순히 양정수 $1,2,\ldots,n$ 이라고 가정해도 보편성이 희생되지 않는다.
- n개의 양수는 $n!$개의 순열(permutation)이 존재한다. 다시말하면, $n!$가지의 다른 순서로 배열할 수 있다는 뜻이다.
- k_i를 i번째 자리에 위치한 정수라고 할때, 순열은 $[k_1,k_2,\ldots,k_n]$으로 나타낼 수 있다. 예를들면, $[3,1,2]$는 $k_1=3,k_2=1,k_3=2$ 로 표시할 수 있다.
- $i<j$와 $k_i>k_j$의 조건을 만족하는 쌍(pair) (k_i,k_j)를 순열에 존재하는 역(inversion)이라고 한다. 예를들어 순열 $[3,2,4,1,6,5]$에는 몇개의 역이 존재할까? 답: 5개
- **정리 7.1**: 서로 다른 n개의 키를 키끼리 비교하여 정렬할때, 매번 비교할때 마다 기껏해야 하나의 역 만을 제거하는 알고리즘은 최악의 경우에 최소한
$$\frac{n(n-1)}{2}$$ 의 비교를 수행해야 하고

  평균적으로 최소한
$$\frac{n(n-1)}{4}$$ 의 비교를 수행해야 한다.

  증명:

  ▷ 경우 1: (최악의 경우)
  $n(n-1)/2$개의 역을 가진 순열이 존재한다고 보이면 충분하다. 왜냐하면 그러한 순열이 있으면 알고리즘이 그만큼의 역을 제거해야 하고, 따라서 최소한 그만큼의 비교를 수행해야 하기 때문이다. 그러한 순열은: $[n,n-1,\ldots,2,1]$.

  ▷ 경우 2: (평균적으로)
  어떤 임의의 순열 $P=[k_1,k_2,\ldots,k_n]$이 있을 때, 그 순열을 뒤집어 놓으면(transpose) $P^T=[k_n,\ldots,k_2,k_1]$이 된다. 이때 어떤 역이 되는 쌍(pair) (s,r)은 반드시 P에 속하거나, 아니면 P^T에 속하게 된다. 그러면 P와 P^T에는 정확하게 총 $n(n-1)/2$개의 역이 존재하게 된다. 따라서 P와 P^T에 존재하는 역의 평균 갯수는 $n(n-1)/4$이 된다. 여기서 알고리즘이 매번 비교할때 마다 기껏해야 하나의 역 만을 제거한다고 가정하였으므로, 알고리즘이 모든 역을 제거하기 위해서는 최소한 역의 갯수 만큼의 비교를 평균적으로 수행해야 한다. 따라서 정렬하는데도 그만큼의 비교를 수행해야 한다.
- 교환, 삽입, 선택 정렬은 한번 비교할 때 기껏해야 하나의 역 이상은 제거할 수 없으므로 시간복잡도가 최악의 경우 $\frac{n(n-1)}{2}$, 평균적으로는 $\frac{n(n-1)}{4}$ 보다 좋을 수 없다.

제 5 절 합병정렬 알고리즘 재검토

- 합병정렬(Mergesort)에서는 한번 비교해서 하나 이상의 역을 제거하는 경우가 있다. 예를들어, 두 부분배열 [3,4]과 [1,2]를 합병해 보자. 3과 1을 비교한 후에 1은 배열의 첫번째 자리에 놓게 되고, 따라서 두개의 역인 (3,1)과 (4,1)을 동시에 제거하게 된다. 다음, 3과 2를 비교한 후에는 2는 배열의 두번째 자리에 놓게 되고, 따라서 두개의 역인 (3,2)와 (4,2)를 동시에 제거하게 된다.
- 키를 비교하는 횟수를 기준으로 시간복잡도를 계산하면: $W(n)=A(n)=\Theta(n\lg n)$
  키를 지정하는 횟수를 기준으로 (모든경우) 시간복잡도를 계산하면: $T(n)\approx 2n\lg n$
- 공간복잡도: $M(n)=\Theta(n)$ — 제자리정렬이 아님.

제 6 절 빠른정렬 알고리즘 재검토

- 키를 비교하는 횟수를 기준으로 시간복잡도를 계산하면: $W(n)=\Theta(n^2)$ and $A(n)\approx 1.38(n+1)\lg n$
  키를 지정하는 횟수를 기준으로 (모든경우) 시간복잡도를 계산하면: $A(n)\approx 3\times 0.69(n+1)\lg n$
- 공간복잡도: 이 알고리즘은 합병정렬알고리즘과 비교해서 추가적인 배열이 필요하지 않는 장점이 있다. 그러나 합병정렬과는 달리 배열이 정확하게 반으로 나누어지지 않으므로, 한쪽의 부분배열을 정렬할때, 다른 한쪽의 부분배열의 첫번과 마지막 인덱스를 활성레코드(activation records) 스택에 저장하여 둘 필요가 있다. 따라서 추가적으로 필요한 저장장소는 최악의 경우 $M(n)=\Theta(n)$가 된다. 따라서 제자리정렬 알고리즘이라고 할 수 없다. 그러나 추가적으로 필요한 저장장소가 $M(n)=\Theta(\lg n)$가 되도록 알고리즘 수정이 가능하다. 교재의 274-275쪽 참조.

제 7 절 힙정렬

- 완전이진트리(complete binary tree): 트리의 내부에 있는 모든 마디에 두개씩 자식마디가 있는 이진 트리. 따라서 모든 잎의 깊이(depth) d는 동일하다.
- 실질적으로 완전이진트리(essentially complete binary tree): 깊이 $d-1$ 까지는 완전이진트리이고, 깊이 d의 마디는 왼쪽 끝에서 부터 채워진 이진트리.
- 힙의 성질(heap property): 어떤 마디에 저장된 값은 그 마디의 자식마디에 저장된 값보다 크거나 같다.

- 힙(heap): 힙의 성질을 만족하는 실질적으로 완전이진트리
- 보기: 교재 276쪽의 Figure 7.4와 7.5를 보시오.
- **알고리즘: 힙정렬(Heapsort)**

```
void siftdown (heap& H)
{
   node parent, largerchild;

   parent = root of H;
   largerchild = parent's child containing larger key;
   while (key at parent is smaller than key at largerchild) {
      exchange key at parent and key at largerchild;
      parent = largerchild;
      largerchild = parent's child containing larger key;
   }
}

keytype root(heap& H)
{
   keytype keyout;

   keyout = key at the root;
   move the key at the bottom node to the root;
   delete the bottom node;
   siftdown(H);
   return keyout;
}

void removekeys(int n,
                heap H,
                keytype S[])
{
   index i;

   for (i = n; i >= 1; i--)
      S[i] = root(H);
}

void makeheap (int n, heap& H)
{
   index i;
   heap Hsub;

   for (i = d-1; i >= 0; i--)
      for (all subtrees Hsub whose roots have depth i)
         siftdown(Hsub);
}

void heapsort(int n,
              heap H,
              keytype S[])
{
   makeheap(n,H);
   removekeys(n,H,S);
}
```

- 이 알고리즘이 어떻게 동작되는지 교재 277-278쪽의 Figure 7.6 and 7.7를 보고 이해하시오.
- 이 알고리즘이 제자리정렬 알고리즘인가? 다음과 같이 힙을 배열로 구현한 경우에는 제자리정렬 알고리즘이다. 공간복잡도: $\Theta(1)$
- 실질적으로 완전이진트리를 배열로 표현하는 방법: 예로서 교재 276쪽의 Figure 7.5의 실질적으로 완전이진트리를 배열로 표현하면, 교재 280쪽의 Figure 7.8과 같이 된다.
  - ▷ 어떤 마디의 왼쪽 자식마디의 인덱스는 그 마디의 인덱스의 두배이다.
  - ▷ 어떤 마디의 오른쪽 자식마디의 인덱스는 그 마디의 인덱스의 두배보다 1만큼 크다.
- **알고리즘: 제자리**

```
void heapsort(int n, heap& H)
{
   makeheap(n,H);
```

```
    removekeys(n,H,H.S);
}
```

- **최악의 경우 시간복잡도 분석 - 비교하는 횟수를 기준:**
  - ▷ 기본동작: `siftdown` 프로시저에서의 키의 비교
  - ▷ 입력의 크기: n, 총 키의 갯수
  - ▷ `makeheap`와 `removekeys` 모두 `siftdown`을 부르므로 따로 분석을 한다.
  - ▷ `makeheap`의 분석: $n = 2^k$라 가정.
    d를 실질적으로 완전이진트리의 깊이라고 하면, $d = \lg n$. 이때 d의 깊이를 가진 마디는 정확히 하나이고 그 마디는 d개의 조상(ancestor)을 가진다. 일단 깊이가 d인 그 마디가 없다고 가정하고 키가 sift되는 상한값(upper bound)을 구해보자.

depth	node수	키가 sift되는 최대횟수
0	2^0	$d-1$
1	2^1	$d-2$
2	s^2	$d-3$
$\cdots$	$\cdots$	$\cdots$
j	2^j	$d-j-1$
$\cdots$	$\cdots$	$\cdots$
$d-2$	2^{d-2}	1
$d-1$	2^{d-1}	0

    따라서 키가 sift되는 총 횟수는 $\sum_{j=0}^{d-1} 2^j(d-j-1)$를 넘지 않는다. 이를 $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$ 와 $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1}+2$를 이용하여 계산해보면,

$$\sum_{j=0}^{d-1} 2^j(d-j-1) = (d-1)\sum_{j=0}^{d-1} 2^j - \sum_{j=0}^{d-1} j2^j = 2^d - d - 1$$

    여기에다 깊이가 d인 경우 sift될 횟수의 상한값인 d를 더하면, $2^d - 1 = n-1$이 된다. 그런데 한번 sift될때마다 2번씩 비교하므로 실제 비교횟수는 $2(n-1)$이 된다.
  - ▷ `removekeys`의 분석: $n = 2^k$라 가정.
    먼저 $n = 8$이고 $d = \lg 8 = 3$인 경우를 생각해보자. 교재 285쪽에 있는 Figure 7.10 를 보면, 처음 4개의 키를 제거하는데 sift되는 횟수가 2회, 다음 2개의 키를 제거하는데 sift되는 횟수가 1회, 그리고 마지막 2개의 키를 제거하는데는 sift되지 않았다. 따라서 총 sift횟수는 $1(2)+2(4) = \sum_{j=1}^{3-1} j2^j$ 가 된다. 따라서 일반적인 경우는

$$\sum_{j=1}^{d-1} j2^j = (d-2)2^d + 2 = d2^d - 22^d + 2 = n\lg n - 2n + 2$$

    가 된다. 그런데 한번 sift될때마다 2번씩 비교하므로 실제 비교횟수는 $2n\lg n - 4n + 4$이 된다.
  - ▷ 두 분석의 통합:
    키를 비교하는 총 횟수는 n이 2^k일때 $2(n-1)+2n\lg n - 4n + 4 = 2(n\lg n - 2n + 1) \approx 2n\lg n$을 넘지 않는다. 따라서 최악의 경우 $W(n) \in \Theta(n\lg n)$. 일반적으로 모든 n에 대해서도 $W(n) \in \Theta(n\lg n)$이라고 할수 있는데, 왜냐하면 $W(n)$은 결국은 증가함수이기 때문이다. 자세한 사항은 교재의 Appendix B의 Theorem B.4 참조.

제 8 절 $\Theta(n\lg n)$ 알고리즘의 비교

알고리즘	비교횟수	지정횟수	추가저장장소사용량
합병정렬	$W(n) = A(n) = n\lg n$	$T(n) = 2n\lg n$	$\Theta(n)$
빠른정렬	$W(n) = \frac{n^2}{2}$ $A(n) = 1.38n\lg n$	$A(n) = 0.69n\lg n$	$\Theta(\lg n)$
힙정렬	$W(n) = A(n) = 2n\lg n$	$W(n) = A(n) = n\lg n$	제자리정렬

제 9 절 키를 비교함으로만 정렬을 수행하는 경우의 하한선

- $n\lg n$보다 더 빠른 정렬 알고리즘을 개발할 수 있을까?
  – 답: 키를 비교하는 횟수를 기준으로 하는 한, 더 빠른 알고리즘은 불가능하다.

9.1 정렬알고리즘에 대한 결정트리

- 3개의 키를 정렬하는 알고리즘을 생각해보자 (교재 287쪽의 `sortthree`). 그 3개의 키를 각각 a,b,c라고 한다면 교재 288쪽의 Figure 7.11과 같은 결정트리(decision tree)를 그릴 수 있다.
- n개의 키를 정렬하기 위한 결정트리를 생각해 보자. 만약 n개의 키의 각 순열(permutation)에 대해서, 뿌리로 부터 잎으로 이르는 그 순열을 정렬하는 경로가 있는 경우, 그 결정트리는 유효하다(valid)고 한다. 즉, 크기가 n인 어떤 입력에 대해서도 정렬할 수 있다.

- 그러면 3개의 입력을 정렬하는 교환정렬 알고리즘의 결정트리는 어떤 모양을 가질까? 교재 289쪽의 Figure 7.12 참조. 여기서 잘 살펴보면 교환정렬에서는 꼭 하지 않아도 될 비교를 하고있다. 왜냐하면 교환정렬에서는 어떤 시점에서 비교가 이루어 질때, 그 이전에 이루어졌던 비교의 결과를 전혀 알 수 없기 때문이다. 이러한 현상은 최적(optimal)이 아닌 알고리즘에서 주로 나타난다.
- 가지친 결정트리(pruned decision tree): 일관성 있는 순서로 결정을 내림으로서 뿌리로 부터 모든 잎에 도달할 수 있는 경우. 교재의 Figure 7.12에 있는 결정트리는 가지친(pruned) 결정트리이다.
- **레마 7.1:** n개의 서로 다른 키를 정렬하는 결정적(deterministic) 알고리즘은, 그에 상응하는 정확하게 $n!$개의 잎을 가진, 유효하며 가지친 이진 결정트리가 존재한다.

9.2 최악의 경우 하한선

- **레마 7.2:** 결정트리에 의해서 수행된 최악의 경우 비교횟수는 그 결정트리의 깊이와 같다.
- **레마 7.3:** 이진트리(binary tree)의 잎의 수가 m이고, 깊이가 d이면, $d \geq \lceil \lg m \rceil$ 이다.
  증명: d에 대하여 귀납법으로 증명, 우선 $2^d \geq m$임을 먼저 보인다.
  - ▷ 귀납출발점: $d = 0$: 마디의 수가 하나인 이진트리. 따라서 명백히 $2^0 \geq 1$.
  - ▷ 귀납가정: 깊이가 d인 모든 이진트리에 대해서, $2^d \geq m$가 성립한다고 가정.
  - ▷ 귀납절차: 깊이가 $d+1$인 모든 이진트리에 대해서, $2^d \geq m'$임을 보이면 된다. 여기서 m'은 잎의 수이다.

    $$\begin{aligned} 2^{d+1} = 2 \times 2^d &\geq 2m && \text{귀납가정에 의해서 성립} \\ &\geq m' && \text{각 부모마디는 기껏해야 자식마디 2개를 가지므로} \end{aligned}$$

    그러므로 $2^d \geq m$이 성립한다. 여기서 양변에 lg를 씌우면, $d \geq \lg m$이 된다. 그런데 d는 정수이므로, $d \geq \lceil \lg m \rceil$이 된다.
- **정리 7.2:** n개의 서로 다른 키를 비교함으로 만 정렬하는 결정적 알고리즘은 최악의 경우 최소한 $\lceil n \lg n - 1.45n \rceil$ 번의 비교를 수행한다.
  증명: 레마 7.1에 의하면, $n!$개의 잎을 가진 가지친, 유효한, 이진결정트리가 존재한다. 다시 레마 7.3에 의하면, 그 트리의 깊이는 $\geq \lceil \lg(n!) \rceil$가 되고, 그러면

  $$\begin{aligned} \lg(n!) &= \lg[n(n-1)(n-2)\cdots 2] \\ &= \textstyle\sum_{i=2}^{n} \lg i \\ &\geq \int_1^n \lg dx \\ &= \frac{1}{\ln 2}(n \ln n - n + 1) \\ &\geq n \lg n - 1.45n \end{aligned}$$

  따라서 레마 7.2에 의해서, 결정트리의 최악의 경우의 비교횟수는 그 트리의 깊이와 같다. 증명 끝.

제 10 절 정렬알고리즘의 분류

- **선택하여 정렬하는 부류의 알고리즘:** 순서대로 항목을 선택하여 적절한 곳에 위치시키는 알고리즘. 예로서, 교환정렬, 삽입정렬, 선택정렬, 힙정렬 알고리즘이 이 부류에 속한다. (공간복잡도는 모두 $\Theta(1)$ 이다. 따라서 제자리정렬 알고리즘이다.)
- **자리를 바꾸어서 정렬하는 부류의 알고리즘:** 입접해 있는 항목을 교환하여서(자리를 바꾸어서) 정렬이 수행되는 알고리즘. 예로서, 거품정렬(Bubble Sort($\Theta(n^2)$)), 합병정렬, 빠른정렬 알고리즘이 이 부류에 속한다. (거품정렬을 제외하고는 모두 제자리정렬 알고리즘이 아니다.)

제 11 절 분배에 의한 정렬: 기수정렬

- 키에 대해서 아무런 정보가 없는 경우에는 키들을 비교하는 것 이외에는 다른 방법이 없으므로 $\Theta(n \lg n)$보다 더 좋은 알고리즘을 만드는 것은 불가능하다.
- 그러나 키에 대한 어느 정도의 정보를 알고 있는 경우에는 상황이 조금 달라질 수 있다. 가령 키들이 음이 아닌 수이고 디지트(digit)의 갯수가 모두 같다면, 첫번째 디지트가 같은 수끼리 따로 모으고, 그 중에서 두번째 디지트가 같은 수끼리 따로 모으고, 마지막 디지트까지 이런 식으로 계속 모은다면, 각 디지트를 한번씩만 조사를 하면 정렬을 완료할 수 있다. 교재 297쪽의 Figure 7.14에 예가 제시되어 있는데, 이런 방법으로 정렬하는 것을 "분배에 의한 정렬(sorting by distribution)" - 기수정렬(radix sort) - 이라고 한다.
- 이와 같이 정렬하는 경우 항상 뭉치(pile)를 구성하는 갯수가 일정하지 않으므로 관리하기가 쉽지 않다. 이를 해결하기 위해서는 다음 예와 같이 오른쪽 끝에있는 디지트부터 먼저 조사를 시작하면 된다.

```
            239 234 879 878 123 358 416 317 137 225
            ___ ___ ___ ___ _______ _______ _______
1st pass    123 234 225 416 317 137 878 358 239 879
            _______ _______ _______ ___ _______
2nd pass    416 317 123 225 234 137 239 358 878 879
            _______ _______ _______ ___ _______
3rd pass    123 137 225 234 237 317 358 416 878 879    끝
```

11.1 시간복잡도 분석

- 기본동작: 뭉치에 수를 추가하는 동작
- 입력의 크기: 정렬하려는 정수의 갯수 = n, 각 정수를 이루는 디지트의 최대 갯수 = $numdigits$
- 모든 경우 시간복잡도 분석:

$$T(n) = numdigits(n+10) \in \Theta(numdigits \times n)$$

따라서 $numdigits$가 n과 같으면, 시간복잡도는 $\Theta(n^2)$가 된다. 그러나 일반적으로 서로 다른 n개의 수가 있을 때 그것을 표현하는데 필요한 디지트의 수는 $\lg n$으로 볼 수 있다. 따라서 기수정렬의 최선의 경우 시간복잡도는 $\Theta(n \lg n)$이고, 보통 최선의 경우가 성취된다. 예: 주민등록번호는 13개의 디지트로 되어 있는데, 표현할 수 있는 갯수는 10,000,000,000,000개이다. 이 10조개의 번호를 기수정렬하는데 걸리는 시간은 $10,000,000,000,000 \times \log_{10} 10,000,000,000,000 = 130$조

11.2 공간복잡도 분석

- 추가적으로 필요한 공간은 키를 연결된 리스트로 만드는데 필요한 공간, 즉, $\Theta(n)$이다.

제 8 장

계산복잡도 개론: 검색 문제

제 1 절 키를 비교함으로만 검색을 수행하는 경우의 하한

- **검색(Searching) 문제:** n개의 키를 가진 배열 S와 어떤 키 x가 주어졌을 때, $x = S[i]$가 되는 첨자 i를 찾는다. 만약 x가 배열 S에 없을때는 오류로 처리한다.
- **이진 검색 알고리즘:** 배열이 정렬이 되어 있는 경우 시간복잡도가 $W(n) = \lfloor \lg n \rfloor + 1$가 되어서, 매우 효율적인 알고리즘이라고 할 수 있다.

  문: 이 보다 더 좋은(빠른) 알고리즘은 존재할까?

  답: 키를 비교함으로만 검색을 수행하는 경우에는 이진검색알고리즘 보다 더 좋은 알고리즘은 존재할 수 없다.
- **검색문제의 하한(lower-bound) 찾기:** (보기) 7개의 키를 검색하는 문제를 생각해보자. 교재 309쪽의 Figure 8.1의 이진검색을 위한 결정트리(decision tree)를 보자. 여기서 비교하는 마디만 고려하면 이 결정트리는 이진트리(binary tree)가 된다. 이 이진트리는 내부마디(internal node)에서 비교가 이루어지고, 각 가지(branch)는 최대한 3개의 내부마디를 가진다. 순차검색의 경우 결정트리는 교재 310쪽의 Figure 8.2와 같이 된다. 여기에서 가지는 7개의 내부마디를 가진다.

1.1 최악의 경우 하한 찾기

- 최악의 경우 비교하는 횟수는 결정트리의 뿌리(root)에서 잎(leaf)까지 가장 긴 경로(path) 상의 마디(node)의 수 인데, 이는 트리의 깊이(depth) + 1 이다.
- **레마 8.1:** n이 이진트리의 마디의 갯수이고, d가 깊이 이면, $d \geq \lfloor \lg n \rfloor$

  증명:

$$\begin{aligned} n &\leq 1 + 2 + 2^2 + \cdots + 2^d \\ n &< 2^{d+1} \\ \lg n &< d + 1 \\ \lfloor \lg n \rfloor &\leq d \end{aligned}$$

- **레마 8.2:** n개의 다른 키 중에서 어떤 키 x를 찾는 가지친 유효 결정트리(pruned, valid decision tree)는 비교하는 마디가 최소한 n개 있다.

  증명: 교재 참조.
- **정리 8.1:** n개의 다른 키가 있는 배열에서 어떤 키 x를 찾는 알고리즘은 키를 비교하는 횟수를 기준으로 했을 때 최악의 경우 최소한 $\lfloor \lg n \rfloor + 1$ 만큼의 비교를 한다.

  증명: 그 알고리즘의 결정트리에서 비교하는 마디 만으로 이루어진 이진트리를 보면, 최악의 경우 비교횟수는 이진트리의 뿌리에서 부터 잎까지의 모든 경로 가운데 가장 긴 것의 마디 수가 된다. 그 수는 이진트리의 깊이 + 1 이 된다. 그런데 레마 8.2에서 이 이진트리의 마디 수는 n이라 하였으므로, 그 이진트리의 깊이는 $\lfloor \lg n \rfloor$ 보다 크거나 같다.

1.2 평균의 경우 하한 찾기

- 최악의 경우와 비슷하게 $\lfloor \lg n \rfloor - 1$ 이다. 증명 생략.

제 2 절 보간 검색

- 비교하는 이외에, 다른 추가적인 정보를 이용하여 검색을 할 수 있는 경우 하한을 개선할 수 있다.
- 보기: 10개의 정수를 검색하는데, 여기서 첫번째 정수는 0-9 중에 하나 이고, 두번째 정수는 10-19 중에 하나이고, 세번째는 20-29 중에 하나이고, $\cdots$, 열번째 정수는 90-99 중에 하나라고 하자. 이 경우 만약 검색 키 `x`가 0 보다 작거나, 99보다 크면 바로 검색이 실패임을 알수 있고, 그렇지 않으면 검색 키 `x`와 `S[1 + x div 10]` 과 바로 비교해 보면 된다.

- 보통 위의 보기 만큼 자세한 정보를 항상 가질 수 있다고는 보기에는 무리가 있다. 그러나 일반적으로 데이터가 가장 큰 값과 가장 작은 값이 균등하게 분포되어 있다고 가정해 보는 것은 타당성이 있어 보인다. 이 경우 반으로 무작정 나누는 대신에, 키가 있을 만한곳을 바로 가서 검사해 보면 더욱 빨리 검색을 마칠 수가 있을 것이다. 이런 식의 검색 알고리즘을 **보간검색(interpolation search)** 이라고 한다.
- 보간검색에서는 선형보간법(linear interpolation)을 사용하여 `mid`를 다음 식을 이용하여 구한다.

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

  보기: $S[1] = 4$이고 $S[10] = 97$ 일때 검색 키가 25 이면, $mid = 3$이 된다.
- **분석:** 아이템이 균등하게 분포되어 있고, 검색 키가 각 슬롯에 있을 확률이 같다고 가정하면, 보간검색의 평균적인 시간복잡도는 $A(n) \approx \lg(\lg n)$이 된다. 예로서 n이 10억이라면, $\lg(\lg n)$은 약 5가 된다. 단 보간검색의 단점은 최악의 경우의 시간복잡도가 나쁘다는 것이다. 예로서 10개의 아이템이 1,2,3,4,5,6,7,8,9,100 이고, 여기서 10을 찾으려고 한다면, mid값은 항상 low값이 되어서 모든 아이템과 비교를 해야한다. 따라서 최악의 경우 시간복잡도는 순차검색과 같다.

2.1 보강된 보간검색(Robust Interpolation Search)

- gap이란 변수를 항상 $mid - low$와 $high - mid$ 보다 항상 작도록 유지하여 다음과 같이 mid를 구한다.
  1. $gap = \lfloor (high - low + 1)^{\frac{1}{2}} \rfloor$
  2. mid 값을 위와같이 선형보간법으로 구한다.
  3. 다음식으로 새로운 mid 값을 구한다.

$$mid = minimum(high - gap, maximum(mid, low + gap))$$

- 이 방법으로 위의 예의 mid 값을 다시 구해 보시오. 답: $mid = 4$.
- **분석:** 아이템이 균등하게 분포되어 있고, 검색 키가 각 슬롯에 있을 확률이 같다고 가정하면, 보강된 보간검색의 평균적인 시간복잡도는 $A(n) \approx \lg(\lg n)$이 되고, 최악의 경우는 $W(n) \approx (\lg n)^2$

제 3 절 트리구조를 검색하기

- **정적검색(Static searching):** 데이터가 한꺼번에 파일에 저장되어 추후에 추가나 삭제가 이루어 지지않는 경우에 이루어 지는 검색, 즉 예를들면 OS 명령에 의한 검색.
- **동적검색(Dynamic searching):** 데이터가 수시로 추가 삭제되는 유동적인 경우, 예로서 비행기예약시스템.
- 배열 자료구조를 사용하면, 정적검색의 경우는 문제없이 이진검색이나 보간검색 알고리즘을 적용할 수 있지만, 동적검색의 경우는 이 알고리즘을 적용하기가 불가능하다. 따라서 트리(tree) 구조를 사용하여야 한다.

3.1 동적검색을 위한 이진검색트리

- **정의:** 이진검색트리는(Binary search tree)는 다음 조건을 만족하는 이진트리이다:
  - ▷ 각 마디 마다 하나의 키가 할당되어 있다.
  - ▷ 어떤 마디 n의 왼쪽부분트리(left subtree)에 속한 모든 마디의 키는 그 마디 n의 키 보다 작거나 같다.
  - ▷ 어떤 마디 n의 오른쪽부분트리(right subtree)에 속한 모든 마디의 키는 그 마디 n의 키 보다 크거나 같다.
- in-order 트리순환(tree traversal)을 하면 정렬된 순서로 아이템을 추출할 수 있다.
- 평균 검색시간을 낮게 유지한다: 동적으로 트리에 아이템이 추가되고 삭제되므로 트리가 항상 균형(balabnce)을 유지한다는 보장이 없다. 특히 최악의 경우는 연결된 리스트(linked list)를 사용하는것(skewed tree)과 같은 효과를 준다. 그러나 랜덤(random)하게 아이템이 트리에 추가되는 경우는 대체로 트리가 균형을 유지한다고 볼 수 있으므로 평균적으로 효율적인 검색시간을 기대할 수 있다.
- 이진검색트리를 이용하면 빨리 아이템을 추가하고 삭제한다.
- **정리:** 검색하는 아이템 x가 n개의 아이템 중의 하나가 될 확률이 동일하다고 가정하면, n개의 아이템을 가진 모든 입력을 검색하는 데 걸리는 평균시간은 이진검색트리를 사용하면 대략 $A(n) = 1.38 \lg n$ 이 된다.

  증명: k번째로 작은 아이템이 뿌리에 위치하고 있는 n개의 아이템을 가진 모든 이진검색트리들을 생각해 보자. 이 트리들은 모두 왼쪽 부분트리에 $k-1$개의 마디가 있고, 오른쪽 부분트리에 $n-k$개의 마디가 있다. $A(k-1)$을 왼쪽 부분트리를 검색하는데 걸리는 평균검색시간 이라고 하고, $A(n-k)$을 오른쪽 부분트리를 검색하는데 걸리는 평균검색시간 이라고 하자. 그러면 x가 왼쪽 부분트리에 있을 확률은 $(k-1)/n$이 되고, 오른쪽 부분트리에 있을 확률은 $(n-k)/n$이 된다. 이때 $A(n|k)$를 k번째로 작은 아이템이 뿌리에 위치하고 있는 이진검색트리를 생성하는 크기 n의 입력에 대한 평균검색시간 이라고 하면,

$$A(n|k) = A(k-1)\frac{k-1}{n} + A(n-k)\frac{n-k}{n} + 1$$

  그러면,

$$A(n) = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}A(k-1) + \frac{n-k}{n}A(n-k) + 1\right]$$

여기서 $C(n) = nA(n)$으로 놓으면,

$$\frac{C(n)}{n} = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}\frac{C(k-1)}{k-1} + \frac{n-k}{n}\frac{C(n-k)}{n-k} + 1\right]$$

$$C(n) = \sum_{k=1}^{n}\left[\frac{C(k-1)}{n} + \frac{C(n-k)}{n} + 1\right] = \sum_{k=1}^{n}\frac{1}{n}\left[C(k-1) + C(n-k)\right] + n$$

$$C(1) = 1A(1) = 1$$

이 재현방정식(recurrence)은 빠른정렬의 평균의 경우의 시간복잡도와 거의 같다. 따라서 $C(n) \approx 1.38(n+1)\lg n \approx 1.38\lg n$

- 그러나 최악의 경우의 시간복잡도는 $\Theta(n)$ 이므로 이 방법이 항상 효율적이라고는 보장할 수 없다.

3.2 B-트리

- 디스크에 접근하는 시간(외부검색:external search)은 RAM에 접근하는 시간(내부검색:internal search) 보다 훨씬 느리기 때문에 실제로는 검색시간이 선형으로 나타난다 하더라도 외부검색의 경우에는 좋다고 받아들여 질 수 없다. 따라서 이런 경우 이진트리가 항상 균형을 이루게 함으로서 검색시간을 줄인다.
- AVL 트리: 아이템의 추가시간, 삭제시간이 모두 $\Theta(\lg n)$이고, 검색시간도 마찬가지 이다.
- B-트리(B-Trees)/2-3 트리: 잎들의 깊이(수준)를 항상 같게 유지.

제 4 절 해쉉(Hashing)

- 1에서 100사이의 정수로 된 키가 있고, 100개의 레코드가 있다고 하자. 그러면 크기가 100인 배열 `S`를 만들어서 저장하면 빠른 시간 안에 검색할 수 있다. 그런데 만약 키가 주민등록번호라면 너무 많은 저장장소가 필요하게 된다.
- 해법: 0..99의 첨자를 가진 크기가 100인 배열을 만든 후에, 키를 0..99 사이의 값을 가지도록 **해쉬(hash)**한다. 여기서 해쉬함수는 키를 배열의 첨자 값으로 변환하는 함수이다. 해쉬함수의 예: `h(key) = key % 100`
- 여기서 2개 이상의 키가 같은 해쉬값을 갖는 경우 충돌(collision)이 생긴다.
- 충돌 방지법: 오픈 해쉉(open addressing)
  같은 해쉬값을 갖는 키들을 바구니에 모아 놓는다. 주로 바구니는 연결된 리스트(linked list)로 구현을 한다. 나중에 바구니를 검색할 때는 순차검색으로 한다.
- 만일 모든 키가 같은 해쉬값을 갖는 경우, 즉, 같은 바구니에 모두 모여 있는 경우에는 순차검색과 동일하다. 그러나 다행히도 그럴 확률은 거의 없다.
$$100 \times \left(\frac{1}{100}\right)^{100} = 10^{-198}$$
- 해쉉이 효과를 얻기 위해서는 키가 바구니에 균일하게 분포하면 된다. 즉, 예를 들면, n개의 키과 m개의 바구니가 있을때, 각 바구니에 평균적으로 $\frac{n}{m}$개의 키를 갖게 하면 된다.
- **정리 8.4:** n개의 키가 m개의 바구니에 균일하게 분포되어 있다면, 검색에 실패한 경우 비교 횟수는 $\frac{n}{m}$이다.
- **정리 8.4:** n개의 키가 m개의 바구니에 균일하게 분포되어 있고, 각 키가 검색하게 될 확률이 모두 같다면, 검색에 성공한 경우 비교 횟수는 $\frac{n}{2m} + \frac{1}{2}$이다.
  증명: 각 바구니의 평균 검색시간은 $\frac{n}{m}$개의 키를 순차검색하는 평균시간과 같다. x개의 키를 순차검색하는데 걸리는 평균 검색시간은
$$1 \times \frac{1}{x} + 2 \times \frac{1}{x} + \cdots + x \times \frac{1}{x} = \frac{1}{x}\sum_{i=1}^{x} i = \frac{1}{x} \times \frac{x(x+1)}{2} = \frac{x+1}{2}$$
  따라서
$$\frac{\frac{n}{m}+1}{2} = \frac{n}{2m} + \frac{1}{2}$$
- 보기: 키가 균일하게 분포되어 있고 $n = 2m$일때
  - ▷ 검색 실패시 걸리는 시간 $= \frac{2m}{m} = 2$
  - ▷ 검색 성공시 걸리는 시간 $= \frac{2m}{2m} + \frac{1}{2} = \frac{3}{2}$

제 9 장

계산복잡도와 다루기 힘든 정도: NP이론의 소개

제 1 절 다루기 힘든 정도

- **다항식으로 시간복잡도가 표시되는 알고리즘(Polynomial-time algorithm):** 입력의 크기가 n일때, 최악의 경우 수행시간이 $W(n) \in O(p(n))$ 인 알고리즘. 여기서 $p(n)$은 n의 다항식 함수(polynomial function)이다. 예를 들면, 시간 복잡도가 $2n$, $3n^3+4n$, $5n+n^{10}$,$n\lg n$인 알고리즘들은 모두 다항식으로 시간복잡도가 표시되는 알고리즘에 속한다. 2^n, $2^{0.01n}$, $2^{\sqrt{n}}$, $n!$ 은 다항식이 아니다.
- **다루기 힘든 정도(Intractability):** 어떤 문제를 다항식으로 시간복잡도가 표시되는 알고리즘으로 풀 수가 없을때, 그 문제를 "다루기 힘들다(intractable)"라고 한다.
  반예: 연쇄행렬곱셈문제(Chained Matrix Multiplication Problem) (Sec.3.4)
  - ▷ 무작정알고리즘: $\Theta(2^n)$
  - ▷ 동적계획법 알고리즘: $\Theta(n^3)$
  - ▷ 따라서 이 문제는 다루기 힘들지 않다 (not intractable)

제 2 절 문제의 분류: 3가지의 문제 카테고리

2.1 다항식으로 시간복잡도가 표시되는 알고리즘이 존재하는 문제

- 모든 알고리즘들의 수행시간이 다항식인 문제
  - ▷ 정렬(sorting)하는 문제: $\Theta(n\lg n)$
  - ▷ 정렬된 배열를 키를 검색하는(searching) 문제: $\Theta(\lg n)$
  - ▷ 행렬(matrix) 곱셈문제: $\Theta(n^{2.38})$
- 다항식이 아닌 수행시간을 가진 알고리즘도 있으나, 다항식의 수행시간을 가진 알고리즘을 찾은 경우
  - ▷ 연쇄행렬곱셈 문제
  - ▷ 최단경로 문제
  - ▷ 최소비용신장트리(Minimum spanning tree) 문제

2.2 다루기 힘들다고 증명이 된 문제

- 비다항식(nonpolynomial) 크기의 결과를 요구하는 문제:
  예: 모든 해밀토니안 회로를 결정하는 문제 - 만일 모든 정점들 간에 이음선이 있다면, $(n-1)!$개의 답을 얻어야 한다. 이러한 문제는 하나의 해밀토니안 회로를 구하는 문제에 비해서 필요이상으로 많은 답을 요구하므로 사실상 비현실적이고, 다루기 힘든 문제로 분류된다.
- 비다항식 크기의 결과를 요구하지 않고, 다항식 시간에 풀 수 없다고 증명된 문제: 놀랍게도 이런 부류에 속하는 문제는 상대적으로 별로 없다.
  - ▷ 결정불가능한 문제(undecidable problem): 그 문제를 풀 수 있는 알고리즘이 존재할 수 없다고 증명이 된 문제
    (예) Halting 문제: 어떤 프로그램 P가 정상적으로 수행이 되어서 종료하는지를 결정하는 문제. (1936년 Alan Turing에 의해서 증명되었음)
    정리: Halting 문제는 결정불가능(undecidable)하다.

증명: 이 문제를 풀 수 있는 알고리즘이 존재한다고 가정하자. 그 알고리즘은 어떤 프로그램을 input으로 받아서 그 프로그램이 종료하면 "예"라는 답을 주고, 그 프로그램이 종료하지 않으면 "아니오"라는 답을 줄 것이다. 그 알고리즘을 `Halt`라고 하면, 다음과 같은 "말도안돼" 알고리즘을 만들 수 있다.

```
algorithm 말도안돼
  if Halt(말도안돼) == "예" then
     while true do
       print "야호"
```

(1) 만일 "말도안돼" 알고리즘이 정상적으로 종료(halt)하는 알고리즘이라고 한다면, 조건식 `Halt(말도안돼)`는 "예"가 되고, 따라서 이 알고리즘은 절대로 종료하지 않는다. 이는 가정과 상반된다.

(2) 만일 "말도안돼" 알고리즘이 정상적으로 종료(halt)하지 않는 알고리즘이라고 한다면, `Halt(말도안돼)`는 "아니오"가 되고, 따라서 이 알고리즘은 종료하게 된다. 이도 마찬가지로 가정과 상반된다.

결론적으로, `Halt`라는 알고리즘은 존재할 수 없다.

▷ 결정가능(decidable)하면서 다루기 힘든(intractable) 문제:
(예) Presburger Arithmetic 문제 (Fischer와 Rabin에 의하여 증명됨, 1974)

2.3 다루기 힘들다고 증명되지도 않았고, 다항식으로 시간복잡도가 표시되는 알고리즘도 찾지 못한 문제

- 많은 문제들이 이 카테고리에 속한다. (예) 0-1 배낭채우기 문제, 외판원 문제, m-색칠하기 문제 ($m \geq 3$), 해밀토니안 회로 문제 등등
- 이러한 문제들은 다음 절에서 다룰 NP(Nondeterministic Polynomial) 문제에 속한다.

제 3 절 NP이론

- 최적의 해를 구하는 문제(optimization problems) - 최적의 해를 찾아야 한다
- 결정 문제(decision problems) - 대답이 "예" 또는 "아니오"로 이루어지는 문제 ⇒ 이론을 전개하고 이해하기 쉬움
  - ▷ 어떤 최적의 해를 구하는 문제에 대한 해답으로 부터 그 결정 문제에 대한 해답은 저절로 나온다.
  - ▷ 어떤 최적의 해를 구하는 문제에 대해서 다항식 시간 알고리즘이 있다면, 그 알고리즘으로 부터 그 문제에 해당하는 결정 문제에 대한 다항식 시간 알고리즘도 쉽게 유추해 낼 수 있다.
  - ▷ 따라서 NP와 P를 다룰때 결정 문제만 고려해도 충분하다.
- 보기: 외판원 문제
  - ▷ 최적의 해를 구하는 문제:
    가중치가 있는 방향성 그래프에서, 한 정점에서 출발하여 아른 모든 정점을 정확히 한번씩만 방문하면서, 총 여행거리가 최소가 되는 여행경로를 구하시오.
  - ▷ 결정 문제:
    어떤 양수 d이 주어지고, 총 여행거리가 d를 넘지 않는 경로가 있는지 없는지를 결정하시오.
- 보기: 0-1 배낭채우기 문제:
  - ▷ 최적의 해를 구하는 문제:
    배낭에 넣을 아이템의 무게와 가치를 알고 있을 때, 용량이 W가 되는 배낭에 아이템을 총 이익이 최대가 되도록 채우시오.
  - ▷ 결정 문제:
    용량이 W가 되는 배낭에 아이템을 총 이익이 최소한 P가 되도록 채울 수 있는지를 결정하시오.

3.1 P 와 NP

- **정의:** P는 다항식 시간 알고리즘으로 풀 수 있는 결정 문제들의 집합이다.
  P에 속해 있는 문제들 - 정렬 문제, 검색 문제, 행렬곱셈문제 등.
  P에 속해 있지 않은 문제 - Presburger Arithmetic
- **검증(Verification):** 결정 문제의 답이 "예"인지를 검증하는 절차. 예를들어 외판원 결정 문제의 답이 "예"라면, 한 여행경로가 주어졌을 때, 그 경로가 과연 그런지를 확인한다.

```
function verify(G: weighted-digraph;
                d: number;
                S: claimed-tour);
  begin
    if S is a tour and the total weight of the edges in S <= d then
      verify := true
    else
      verify := false
  end;
```

이 검증 절차는 다항식 시간 안에 수행될 수 있다. 즉, d 보다 작은 여행경로를 찾는 것이 아니라 (이 과정은 다항식 시간으로 해결하지 못할 수 도 있다), 주어진 여행경로가 d 보다 작게 걸리는 것인지를 알아보는 것이다,

- **비결정적(Nondeterministic) 알고리즘:**
  1. 추측단계(Guessing state:비결정적임):
     문제의 입력이 주어지면,단순히 해답을 추측한다. (말도 안돼는 답이어도 상관없음)
  2. 검증단계(Verification stage: 결정적임):
     입력: 입력과 추측한 해답
     출력:
     - ▷ "맞음"이라는 답을 주고 멈춘다
     - ▷ "틀림"이라는 답을 주고 멈춘다
     - ▷ 무한 루프

  실제 상황에서 이 비결정적 알고리즘으로 문제를 푸는 것은 불가능 하다. 그러나 다음과 같은 경우에는 비결정적 알고리즘이 결정 문제를 "푼다"고 한다:
  - ▷ "예" 답을 줄 입력에 대해서 검증단계가 "맞음" 답을 줄 추측한 해답이 있다.
  - ▷ "아니오" 답을 줄 입력에 대해서 검증단계가 "맞음" 답을 줄 추측한 해답이 없다.
- **정의:** 다항식 시간 비결정적 알고리즘 (Polynomial-time nondeterministic algorithm)이란 추측단계가 다항식 시간 알고리즘인 비결정적 알고리즘을 말한다.
- **정의:** NP(Nondeterministic Polynomial)는 다항식 시간 비결정적 알고리즘에 의해서 풀 수 있는 모든 결정 문제의 집합이다.
- 즉 어떤 결정 문제에 대해서 검증을 다항식 시간에 하는 알고리즘이 있다면, 그 결정 문제는 NP에 속한다. 다시말하면 어떤 결정 문제를 풀수있는 다항식 시간 알고리즘을 찾을 수 없을때, 다항식 시간 비결정적 알고리즘을 찾으면 그 문제는 NP에 속한다. 그러면 어떤 결정 문제가 주어졌을때, 그 문제가 NP 임을 보일 수가 있겠읍니까? 이런 문제는 좋은 학기말 시험문제가 되겠죠?
- P에 속하는 모든 문제는 당연히 NP에도 속한다.
- NP에 속하지 않는 문제는 어떤 것이 있는가? 다루기 힘들다(intractable) 고 증명이 된 문제, 즉 Halting 문제, Presburger Arithmetic 문제 등이다.
- (중요한 사실) NP에 속한 문제 중에서 P에 속하지 않는다고 증명이 된 문제는 하나도 없다. 따라서 아마도 $NP - P = \emptyset$ 일지도 모른다. 그럼 P = NP? 이것이 사실이라면 거의 모든 결정 문제가 NP에 속하기 때문에 우리가 아는 거의 모든 문제가 다항식 시간 알고리즘이 있다는 얘기가 된다. 사실 많은 사람들이 $P \neq NP$ 일것 이라고 생각하고 있기는 하지만 아무도 이것을 증명하지 못하고 있는 것이다.

3.2 NP-Complete 문제

- NP-Complete에 속하는 문제 중에서 어떤 하나 만이라도 P에 속한다는 것이 밝혀지면, 다른 모든 문제도 P에 속해야 한다.
- **CNF(Conjunctive Normal Form)** : x를 논리변수(logical variable) 라고 하면, x가 참이라는 말은 $\bar{x}$는 거짓이라는 말과 동일하다. x나 $\bar{x}$는 리터럴(literal)이라 하고, $\vee$ 연산자로 리터럴을 결합하면 절(clause)이라 한다. $\wedge$로 절을 연결하면 CNF가 된다. 예를들면, $(\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (x_1 \vee \bar{x_4}) \wedge (\bar{x_2} \vee x_3 \vee x_4)$는 CNF이다.
- **CNF-Satisfiability 결정 문제:** CNF가 참이 될수 있도록 논리값(참 또는 거짓)을 지정할 수 있는지의 여부를 결정하는 문제.
  보기:
  - ▷ $(x_1 \vee x_2) \wedge (x_2 \vee \bar{x_3}) \wedge \bar{x_2}$ - yes
  - ▷ $(x_1 \vee x_2) \wedge \bar{x_1} \wedge \bar{x_2}$ - no

  이 결정문제는 NP에 속한다.
- **변환(transformation) 알고리즘**: 풀고 싶은 어떤 문제를 A라고 하고, 이미 알고리즘을 알고 있는 어떤 문제를 B라고 하자. A에 대해서 "예"의 답을 해줄 모든 입력을 B에 대해서도 "예"의 답을 해줄 입력으로 변환하는 알고리즘. 그러면, 변환 알고리즘과 B에 대한 알고리즘을 합하면, A에 대한 알고리즘이 나온다.
- **정의:** A에서 B로 다항식시간에 변환하는 알고리즘이있다면, A는 "P-시간(polynomial-time) 다대일 축소가능(many-one reducible)" 하다고 하고, $A \propto B$ 로 쓴다.
  따라서 만일 B문제에 대해서 P-time 알고리즘이 있고, A에서 B로의 변환 알고리즘도 P-time이라면, 그 두 알고리즘을 합함으로서 A에 대한 P-time알고리즘을 얻게된다.
- **정리 9.1:** 결정 문제 B 가 P에 속하고 $A \propto B$ 이면 결정 문제 A는 P에 속한다.
- **정의:** 만일 (1) 문제 B가 NP에 속하고, (2) NP에 속해있는 모든 다른 문제 A에 대해서 $A \propto B$ 이면, B는 NP-Complete라고 불리운다.
- 어떤 문제가 NP-Complete인지를 위의 정의에 근거해서 증명하는 일은 매우 어렵다. 왜냐하면 NP에 속한 모든 문제가 그문제로 축소가능(reducible) 하다는 것을 보여야 하기 때문이다. 그러나 다행스럽게도, 1971년 Cook이 다음의 2 정리를 증명했다.
- **정리 9.2:** (Cook's Theorem) CNF-Satisfiability 문제는 NP-Complete 이다.
- **정리 9.3:** 만일 (1) 문제 C가 NP에 속하고, (2) 어떤 NP-Complete 문제 B에 대해서 $B \propto C$ 이면, C는 NP-Complete 이다.
- Cook에 의해서 CNF-Satisfiability 문제가 NP-Complete 임을 알았기 때문에, 주어진 문제가 NP-Complete 인지 아닌지는 위의 정리 9.3에 의해서 비교적 쉽게 증명할 수 있다.