알고리즘(최지웅)

Lee Jun Hyeok (wnsx0000@gmail.com)

December 17, 2024

목차

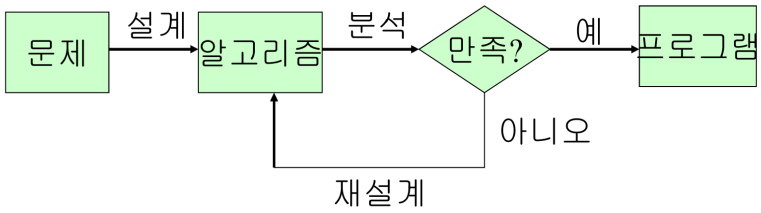
1. 서론

1.1. 알고리즘

1.1.1. 알고리즘

알고리즘(Algorithm)이란 문제 해결을 위한 상세한 절차임.

프로그램 설계에는 아래의 그림과 같이 알고리즘의 설계와 분석, 재설계 과정이 필수적임.



프로그램은 성능적인 제한을 포함하고 있고, 알고리즘이 해당 제한에 부합하는지를 검토하는 과정이 필요함. 프로그램 제작 이후 테스트를 통해 수정하도록 할 수도 있지만, 프로그램의 규모가 커질수록 테스트 이후 수정에 따른 cost가 너무 많이 듦. 즉, 알고리즘을 설계하고 그 효율성을 분석한 뒤 실제 코드를 작성해야 함.

1.1.2. 알고리즘의 표기

1. 표기 언어
알고리즘은 자연어, PL(Programming Language), 의사코드(pseudo code)로 표현할 수 있는데, 주로 pseudo code로 표기함.

pseudo code에는 정해진 규칙이 없지만, 당연하게도 특수한 표기를 사용하는 경우 그 정의를 제공해 줘야 함.

이 요약본에서는 c++ 형태의 pseudo code를 사용함.

2. basic structure
알고리즘을 학술적으로 표현할 때는 아래와 같은 항목들로 구성함.

definition part : 문제 상황(problem), input, output, 자료구조 등을 정의하는 부분.
procedure part : 알고리즘의 실제 동작 방식을 pseudo code 등으로 나타내는 부분.

추가적으로 아래에 성능 분석을 작성하기도 함.

이 요약본에서 pseudo code에 출력은 매개변수에 작성한 참조 변수(&)에 대입하는 것으로 함.

1.2. 성능 분석

1.2.1. 성능 분석

알고리즘의 성능 분석은 특정한 상황에 대해 복잡도 함수를 구하고, 해당 복잡도 함수가 속하는 order를 구하는 과정임. 이 과정을 정리하면 아래와 같음.

0. 문제 상황 이해 및 더 쉬운 개념으로 변환.
1. 기본 연산 정의.
2. 어떤 case(시/공간, 최선/최악/평균/모든 경우)에 대해 분석할지 정의.
3. 복잡도 함수 정의.
4. 필요한 경우 복잡도 함수 검증.
5. order 계산.

기본 연산, case, order의 종류, 검증을 모두 잘 고려해서 분석해야 함.

1.2.2. case

알고리즘의 성능, 효율(efficiency)은 시/공간에 대한 성능인지, 최선/최악/평균/모든 중 어떤 경우인지에 따라 다르게 측정될 수 있음. 각 경우에 대한 분석은 complexity를 구하게 되고, 해당 complexity가 어떤 order에 포함되는지로 성능을 분류할 수 있음.

1. 시간 vs. 공간
알고리즘의 성능은 시간과 공간(memory 사용량)의 측면에서 각각 분석할 수 있음.

대부분의 분석은 시간에 대한 분석임. 공간 분석은 세부적인 분석 방법이 많지 않고, 제한 사항을 위반하지 않는지만 확인하는 경우가 많음.

2. 최선/최악/평균/모든 경우
알고리즘은 입력 데이터의 종류에 따라 성능이 달라질 수 있음.

모든 경우 중 연산량이 가장 적은 경우를 최선의 경우(Best Cases)라고 하고, complexity를 B(n)으로 나타냄. 즉, 성능이 입력 값과 입력 크기 모두에 종속되는 알고리즘에 대해 고려함.

모든 경우 중 연산량이 가장 많은 경우를 최악의 경우(Worst Cases)라고 하고, complexity를 W(n)으로 나타냄. 즉, 성능이 입력 값과 입력 크기 모두에 종속되는 알고리즘에 대해 고려함.

전체 경우에 대한 평균을 낸 것을 평균적 경우(Average Cases)라고 하고, complexity를 A(n)으로 나타냄. 즉, 성능이 입력 값과 입력 크기 모두에 종속되는 알고리즘에 대해 고려함. 평균을 계산한 것이므로 구하는 과정이 비교저 복잡하고, 각 경우에 대한 확률을 이야기할 수 있음.

모든 경우가 동일한 연산량을 가지는 알고리즘은 모든 경우(Every Case)에 대해 분석하며, 이때의 complexity은 T(n) 또는 E(n)으로 나타냄. 즉, 성능이 입력 값과는 관계없이 입력 크기에만 종속되는 알고리즘에 대해 고려함.

3. average case의 계산
average case를 계산할 때는 각 경우에 대한 확률을 고려하여 평균을 계산해야 함. 즉, 가중치 평균을 구해야 함.

각 n개의 경우가 있고 각 경우의 기본 연산 수행횟수를 N_k, 확률을 P_k라 하면 average case의 시간복잡도는 아래와 같이 구할 수 있음.

$$A(n) = \sum_{k=1}^{n} N_k P_k$$

단순히 각 경우에 대한 확률이 같다고 무지성하게 가정하고 산술평균을 구해서는 안 됨. 어떤 상황을 가정하더라도 그 가정이 고려되어야 하지, 당연하게 적용해버리면 안 됨.

물론 각 확률을 알려면 해당 알고리즘이 적용되는 상황을 알아야 하므로, 알고리즘만을 논할 때는 가정을 두고 함.

1.2.3. 복잡도

1. 기본 연산
기본 연산(Basic Operation)은 알고리즘 성능 분석의 기준이 되는 연산임. 데이터 입력 크기에 상관없이 실행 시간이 상수인 작업을 기본 연산으로 함.

어떤 연산을 기본 연산으로 정하는지에 따라 복잡도 또는 분석 가능한 case들이 달라지기도 하므로, 성능 분석 전에 기본 연산을 명확히 정의해야 함.

주로 대입 연산, 비교 연산, 함수 호출 등을 기본 연산으로 함.

2. Input Size
입력 크기(Input Size)는 알고리즘의 입력으로 주어지는 데이터의 크기를 말함. 주로 수의 개수, 수의 크기, 배열의 길이, 자료구조의 길이, 행렬에서는 행과 열의 길이, 그래프에서는 정점과 간선의 수를 말함.

3. 복잡도
알고리즘의 성능은 복잡도(Complexity)로 나타낼 수 있음. 복잡도는 복잡도 함수(Complexity Function)으로 나타냄.

주로 시간에 대해서 분석하므로 시간복잡도(Time Complexity)에 대해 분석하고, 이를 시간복잡도 함수로 나타냄. 시간복잡도 함수는 데이터의 input size에 따른 기본 연산의 수행 횟수로 나타냄.

복잡도 함수가 n에 따라 다른 형태일 수 있는데, order를 구할 때는 n을 특정 조건으로 한정해도 결과가 같으므로 편리하게 계산할 수 있음.

every case를 구하는 것이 좋겠지만, 입력 값에 따라 성능이 달라지는 알고리즘에 대해서는 계산이 불가능함. 그래서 일반적인 경우에 대한 성능이 중요하다면 average case를, 특정 성능에 대한 보장이 중요하다면 worst case를 사용함.

복잡도 함수를 한눈에 파악하기 어렵다면, n에 따른 연산 횟수를 직접 작성해 보며 찾는 것이 쉬울 수 있음.

1.2.4. 복잡도 함수의 증명

1. 증명의 필요성
복잡도 함수 또는 그 성능은 대체로 귀납적으로 정의됨. 단순히 order만을 구하는 것이 목적이라면 증명을 할 필요가 없지만, 학술적으로 알고리즘의 성능을 보이려면 엄밀한 증명이 필수적임.

2. 수학적 귀납법
귀납적으로 정의된 복잡도 함수나 그 성능을 연역적으로 검증하는 방식으로는 수학적 귀납법(Mathematical Induction)이 있음.

수학적 귀납법은 아래의 3가지 단계로 구성함.
귀납출발점(induction base) : 귀납의 시작점을 보임.
귀납가정(induction hypothesis) : 귀납 가정을 함.
귀납단계(induction step) : 귀납의 다음 단계가 성립함을 보임. 즉, 귀납의 시작점부터 모든 경우에 대해 성립함을 보인 게 됨.

특히 수학적 귀납법은 무한한 경우에 대해 보여야 하는 증명에서 유용하게 사용됨.

복잡도 함수는 pseudo code를 보고 적절히 작성하거나, 입력을 하나씩 대입해 봄으로써 구하게 됨.

1.2.5. order

Big-O, Omega, Theta, little-o는 모두 복잡도 함수에 대한 집합으로, 알고리즘의 점근적(asymptotic, n이 무한대인 경우) 성능을 나타냄.

이때 $g(n) \in O(f(n))$ 등인 경우, "g(n)은 f(n)의 big-O"의 순서로 읽음.

1. Big-O
Big-O 표기법(O)은 복잡도 함수에 대한 점근적 상한(Asymptotic Upper Bound)을 제공하는 표기법임. 아래의 조건을 만족시키는 복잡도 함수들의 집합을 $O(f(n))$이라고 하고, 복잡도 함수가 이 집합에 포함되는지로 성능을 나타냄.

$O(f(n)) = \{g(n) \mid \textit{there exists two positive constants } c \textit{ and } N \textit{ such that } g(n) \leq cf(n) \textit{ for all } n \geq N\}$

2. Omega
Omega 표기법(Ω)은 복잡도 함수에 대한 점근적 하한(Asymptotic Lower bound)을 제공하는 표기법임. 아래의 조건을 만족시키는 복잡도 함수들의 집합을 $\Omega(f(n))$이라고 하고, 복잡도 함수가 이 집합에 포함되는지로 성능을 나타냄.

$\Omega(f(n)) = \{g(n) \mid \textit{there exists two positive constants } c \textit{ and } N \textit{ such that } g(n) \geq cf(n) \textit{ for all } n \geq N\}$

3. Theta
Theta 표기법(Θ) 또는 order는 복잡도 함수에 대한 점근적 성능(Asymptotic Tight Bound)을 제공하는 표기법임. 아래의 조건을 만족시키는 복잡도 함수들의 집합을 $\Theta(f(n))$이라고 하고, 복잡도 함수가 이 집합에 포함되는지로 성능을 나타냄.

$\Theta(f(n)) = \{g(n) \mid$ *there exists three positive constants* c_1, c_2 *and* N *such that* $c_1 f(n) \leq g(n) \leq c_2 f(n)$ *for all* $n \geq N\}$

Big-O와 Omega의 개념을 모두 포함하는 개념임.

4. little-o

little-o(small-o)는 복잡도 함수끼리의 관계를 나타내기 위한 표기법임. 아래의 조건을 만족시키는지를 확인하여 두 복잡도 함수 중 어떤 것의 성능이 점근적으로 항상 좋은지를 판별할 수 있음.

$o(f(n)) = \{g(n) \mid$ *for every positive constant* c *there exists a positive constant* N *such that* $g(n) < cf(n)$ *for all* $n \geq N\}$

즉, $g(n) \in o(f(n))$*이면 g(n)보다 f(n)가 항상 점근적으로 더 큰 값을 가지게 되고, g(n)의 성능이 f(n)보다 항상 좋다는 것임. 또한 이때 g(n)은 f(n)의 little-o라고 함.*

교재에서는 $g(n) \leq cf(n)$*로 되어 있는데, 실제로 주로 사용되는 의미는 <라고 함.*

f(n)이 g(n)보다 복잡도 카테고리 왼쪽(더 낮은 복잡도)에 위치하면, $f(n) \in o(g(n))$*이 성립함.*

5. 주요 복잡도 카테고리

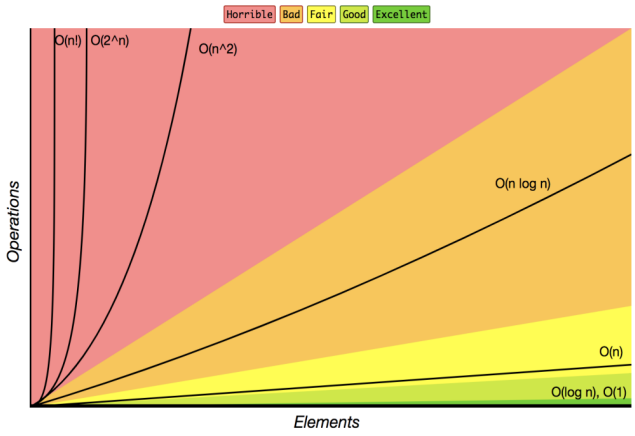
알고리즘이 가지는 복잡도 함수를 점근적 기법으로 나타내면 대체로 복잡도 카테고리 안에 포함됨. order는 정확히는 Θ를 뜻하는 말이지만, 점근적인 분류에 의한 복잡도 함수 간의 계급, 즉 복잡도 카테고리의 각 계층을 의미하기도 함.

주요 복잡도 카테고리를 성능에 따라 나열하면 아래와 같음.

Base Func.	Name
1	Contant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log-linear
n^2	Quadratic
n^3	Cubic
n^p	Polynomial
2^n	Exponential
$n!$	Factorial

Scalability

Good



Poor

집합이므로 기본적으로 포함관계를 $\in$로 나타내지만, $=$로 나타내기도 함.

다항함수는 차수만으로 Big-O를 구할 수 있음.

참고로 input size은 0이 될 수 없음. 0이면 입력되지 않은 것.

input size로 두 개 이상의 매개변수가 올 수도 있음. 자연스럽게 해당 매개변수들에 대해 복잡도 함수를 작성하면 됨. 예를 들어, 복잡도 함수가 n, m에 대해 작성된 경우 $T(n, m) \in \Theta(n \times m)$와 같이 두 변수에 대한 함수를 작성함.

1.2.6. order의 증명

어떤 알고리즘의 복잡도 함수가 특정 order에 속하는지 판별하는 것은 간단하지만, 이 정보를 학술적으로 사용하려면 증명해야 함.

1. 어떤 order에 속함을 증명

알고리즘의 복잡도 함수가 어떤 order에 속함을 증명하는 것은, 적절한 c와 n을 보이면 됨.

증명의 구조는 아래와 같음.

1. 해당 order(big-O/Omega/Theta)의 정의 제시.
2. 적절한 c 지정.
3. n이 존재함을 보임.

이때 유의할 점은, Theta는 Big-O와 Omega에 대해 각각 증명해야 함. 각각 c를 정하고 n은 둘 중 더 큰 것으로 보이면 됨.

2. 어떤 order에 속하지 않음을 증명
알고리즘의 복잡도 함수가 어떤 order에 속하지 않음을 증명하는 것은, 귀류법으로 보이면 됨. 즉, 특정 상황을 가정했을 때 모순을 보이면 됨.

귀류법은 증명의 대상이 배중률을 가져야 적용이 가능함. 배중률은 대상이 2가지 상태(a와 a가 아닌 상태) 만을 가지는 성질을 말함.

증명의 구조는 아래와 같음.
1. 해당 order(big-O/Omega/Theta)의 정의 제시.
2. 해당 복잡도 함수가 해당 order에 속한다고 가정.
3. 식을 정리하여 모순임을 보임.

3. 극한을 활용한 증명
아래와 같이 극한을 활용해 복잡도 함수가 특정 order에 속하는지를 판정할 수 있음. 이때 연산에 번거롭다면 로피탈 정리 등을 사용할 수 있음.

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases}c & \text{if } g(n)\in\Theta(f(n))\\ 0 & \text{if } g(n)\in o(f(n))\\ \infty & \text{if } f(n)\in o(g(n))\end{cases}$$

즉, Theta와 little-o는 극한으로 쉽게 구할 수 있음.

예를 들어, $n^2 \in O(n^3)$을 보여야 한다면, Big-O의 정의를 작성하고, c = 100으로 지정했을 때 N=1이면 성립함을 보이면 됨.

예를 들어, $n^3 \in O(n^2)$이 거짓임을 보여야 한다면, Big-O의 정의를 작성함. $n^3 \leq cn^2$를 만족하는 N, c가 존재해야 하는데, 이를 정리하면 $n \leq c$임. 즉, c보다 작은 n에 대해서만 식이 성립하므로 모순임.

1.2.7. NP-complete

NP-complete은 해당 문제에 대해 성능을 지수함수보다 개선한 알고리즘이 없고, 그런 알고리즘이 존재할 수 없다는 것이 증명되지도 않은 문제를 뜻함.

1.3. order의 성질

1.3.1. order의 성질

order의 주요 성질들은 아래와 같음.

1. Big-O와 Omega의 포함관계
$g(n) \in O(f(n)) \leftrightarrow f(n) \in \Omega(g(n))$

즉, Big-O와 Omega는 반대로 뒤집어 생각할 수 있음.

2. Theta 포함관계
$g(n) \in \Theta(f(n)) \leftrightarrow f(n) \in \Theta(g(n))$

3. 로그 복잡도 함수
임의의 실수 a>1에 대해서, $\log_a n \in \Theta(\log n)$ *항상 성립함.*

즉, 로그 복잡도 함수는 모두 같은 카테고리에 속함. 주로 $\Theta(\log n)$ *또는* $\Theta(\lg n)$*으로 표기함. 참고로* $\lg n$*은* $\log_2 n$*을 줄인 것임.*

4. 지수 복잡도 함수
임의의 실수 $b>a>0$에 대해서, $a^n \in o(b^n)$임.

즉, 지수 복잡도 함수는 밑에 따라 다른 카테고리에 속함.

5. 팩토리얼의 복잡도
임의의 실수 $a>0$에 대해서, $a^n \in o(n!)$임.

즉, 팩토리얼은 모든 지수함수보다 복잡도가 높음.

1.3.2. 단순화 규칙

어떤 알고리즘에 대해, 최악의 경우의 복잡도를 분석할 때, 그 과정을 아래의 5가지 규칙에 의해 그 과정을 단순화할 수 있음. O에 대해 작성했지만 Ω, Θ에도 적용 가능함.

1. Rule 1 : 포함관계 성립
If $T(n) \in O(f(n))$ and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$

$T(n)$이 $O(f(n))$에 속하고, $f(n)$이 $O(g(n))$에 속하면 $T(n)$ 또한 $O(g(n))$에 속한다는 것. big-O가 상한을 지정한다는 것을 생각하면 당연함.

2. Rule 2 : 계수 삭제 가능
If $T(n) \in O(kf(n))$ for constant $k > 0$, then $T(n) \in O(f(n))$

$T(n)$이 속하는 big-O에 대해 생각할 때, $O()$ 내부 함수의 계수는 삭제가 가능함. big-O의 정의를 생각하면 당연함.

3. Rule 3 : 두 병렬적 시간복잡도의 병합
If $T_1(n) \in O(f_1(n))$ and $T_2(n) \in O(f_2(n))$, then $T_1(n)+T_2(n) = (T_1+T_2)(n) \in O(max(f_1(n), f_2(n)))$

두 시간복잡도 모두에 대한 big-O는, 두 시간복잡도 각각의 big-O 중 basic funciton이 더 큰 것임. 주로 프로그램 내에서 순서대로 실행되는 두 코드에 대해서 적용함.

4. Rule 4 : 조건문 분기 처리
If $T(n)$ varies by conditions, then take greater complexity.

최악의 경우에 대한 규칙들이므로, 당연히 조건문 분기에 따라 다른 시간복잡도를 가지게 될 때 둘 중에 더 큰 것을 적용해야 함.

5. Rule 5 : 두 직렬적 시간복잡도의 병합
If $T_1(n) \in O(f_1(n))$ and $T_2(n) \in O(f_2(n))$, then $T_1(n) \times T_2(n) \in O(f_1(n) \times f_2(n))$

반복문 등에서 외부와 내부 모두에 대한 시간복잡도의 big-O는, 내부와 외부의 big-O basic function을 곱한 것임.

이때 주의해야 할 점은, 반복문 내부의 반복이 외부 변수에 영향을 받는 경우 단순 곱으로 계산해 버리면 loose한 upper bound가 만들어진다는 것임. 이 경우 직접 세서 계산해야 함.

2. 분할 정복

2.1. 분할 정복

2.1.1. 분할 정복

1. 분할 정복
분할 정복(Divide-and-Conquer)는 문제를 여러 개의 작은 부분으로 나눈(divide) 뒤 각각을 해결(conquer)하고, 필요하다면 해결한 결과를 모으는(combine) 문제 접근 방식임.

분할 정복은 하향식 접근법(Top-Down approach)임.

분할 정복에서는 주로 재귀를 사용하고, 그에 따라 문제 상황과 코드에 대한 가독성이 좋음.

분할 정복은 나누어진 부분들 사이에 상관관계가 없는 문제를 해결하는 데에 적합한 방식임. 상관관계가 있다면 중복 연산으로 인한 비효율이 발생할 수 있음.

2. 분할 정복과 DP
재귀를 사용한 분할 정복은 굉장히 비효율적인 경우들이 있는데, 주로 작업을 중복해서 처리하기 때문임. 이 경우 중복 작업을 DP 해결할 수 있음. 분할 정복을 시도했는데 성능이 써먹을 수 없을 정도라면 (지수이거나 $n^{\lg n}$ 등인 경우) DP로 방향을 전환해야 함. 이때 시간 복잡도 함수의 order만 빠르게 구해보고 쓸만한지를 판단하는 것이 중요함.

분할 정복으로 풀 수 있는 문제는 DP로도 풀 수 있음. 분할 정복을 우선 시도하는 이유는 재귀를 사용하므로 문제 상황에 대한 직관적인 이해에 도움이 되기 때문임.

물론 분할 정복을 했을 때 성능이 괜찮은 문제 상황들이 존재하고, 여기에서는 그런 경우들에 대해 주로 정리함.

재귀함수에서 지역변수를 많이 사용하면 반복 호출에 의한 메모리 초과가 날 수 있으므로, 알고리즘만 고려하는 상황에서는 전역변수를 사용하는 것이 유리할 수 있음.

꼬리 재귀(Tail Recursion)는 재귀 호출이 함수 마지막에 발생하는 재귀를 말함. 즉, 해당 함수에서 재귀 호출 이후에 수행하는 작업이 없는 재귀임. 재귀에는 중복 연산의 가능성이 존재하지만, 꼬리 재귀의 경우는 중복 연산에 따른 비효율을 고려하지 않아도 될 수 있음. LISP 계통(c, c++, java 등) 중 일부 언어의 컴파일러는 컴파일 시에 꼬리 재귀를 반복문(DP)으로 대치하는 기능을 가지고 있음.

2.1.2. 분할 정복의 성능 분석

1. Repeated Substitution
반복 대치(Repeated Substitution)는 점화식끼리 반복적으로 더하거나 빼거나 대치해서 일반항을 구하는 방법임.

분할 정복에서는 재귀를 주로 사용하므로, 복잡도 함수가 점화식의 형태로 나타나게 되는데, 일반항을 구하는 데에 repeated substitution을 시도할 수 있음.

큰 쪽에서부터 작은 쪽으로 구하는 것을 backward repeated substitution, 작은 쪽에서부터 큰 쪽으로 구하는 것을 forward repeated substitution이라고 함.

2. 복잡도 함수
재귀를 사용한 분할 정복 알고리즘의 시간복잡도는 주로 점화식의 형태를 가짐. repeated substituion을 사용하여 시간복잡도의 일반항을 구할 수 있음.

물론 이렇게 구한 일반항이 모든 경우에 대해 성립하는지는 수학적 귀납법 등을 사용하여 따로 증명해야 함. 수학적 귀납법을 사용하는 경우, ***귀납가정은 특정 값들에 대해 해당 일반항이 성립한다는 것으로 하고, 귀납단계에서는 복잡도 함수의 점화식을 작성하고 가정을 이용해 정리하는 식으로 증명****할 수 있음.*

2.1.3. 마스터 정리

마스터 정리(Master Theorem)은 아래와 같이 재귀적으로 정의된 알고리즘의 시간복잡도에 대한 분석을 단순화한 정리로, 주로 분할 정복 알고리즘에서 사용함.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$h(n) = n^{\log_b a}$*라 하자. 아래가 성립함.*

1. 어떤 양의 상수 ϵ*에 대해서* $\frac{f(n)}{h(n)} \in O(\frac{1}{n^\epsilon})$*이면,* $T(n) \in \Theta(h(n))$*임.*
2. 어떤 양의 상수 ϵ*에 대해서* $\frac{f(n)}{h(n)} \in \Omega(n^\epsilon)$*이고, 충분히 큰* n*에 대해* $af(\frac{n}{b}) \leq cf(n)$*이면,* $T(n) \in \Theta(h(n))$*임.*
3. 어떤 양의 상수 ϵ*에 대해서* $\frac{f(n)}{h(n)} = O(\frac{1}{n^\epsilon})$*이면,* $T(n) \in \Theta(h(n))$*임.*

즉, 쉽게 정리하면 아래와 같음.

1. h(n)이 f(n)보다 성능이 나쁜(값이 큰) 경우, h에 의해 결정되어 $T(n) \in \Theta(h(n))$임.
2. f(n)이 h(n)보다 성능이 나쁜(값이 큰) 경우, f에 의해 결정되어 $T(n) \in \Theta(f(n))$임.
3. f(n)과 h(n)의 성능이 같다면, $T(n) \in \Theta(h(n) \log n)$임.

분할 정복의 관점에서는, 한 계층(하나의 함수)에 대해 생각할 때, 분할하여 만들어지는 부분의 개수가 a이고 분할하여 만들어진 각 부분의 크기가 b임.

분할 정복 등을 사용할 때 시간 복잡도 함수가 위의 형식이라면, 직접 계산하지 않아도 order를 굉장히 간단히 구할 수 있음.

2.2. 이분 탐색

2.2.1. 이분 탐색

1. 이분 탐색
이분 탐색(Binary Search)는 정렬된 배열을 반씩 나눠서 탐색하는 분할 정복임.

비내림차순 배열이라고 가정함. 배열의 중간에 있는 값과 찾으려는 값이 같으면 종료하고, 다르면 배열을 반으로 나눠서 찾으려는 값아 중간 값보다 작으면 왼쪽 배열에 대해서, 크면 오른쪽 배열에 대해서 탐색함.

pseudo code는 아래와 같음.

```
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 반을 선택함
        else
            return location(mid+1, high); // 오른쪽 반을 선택함
    }
}
```

2. 성능 분석
시간에 대한 최악의 경우 복잡도를 구해보자. 비교 연산을 기본 연산으로 함. 이때 대부분의 경우에 대해 비교 연산이 2번씩 발생하므로 이를 하나로 묶어서 취급함.

$n = 2^k$*라고 가정하면* $W(n) = W(\frac{n}{2}) + 1$*이므로 이를 repeated substitution으로 정리하면* $W(n) = \lg n + 1$*임. 즉,* $W(n) \in \Theta(\lg n)$*임.*

3. 증명
일반항이 모든 경우에 대해 성립하는지를 보여야 함.

우선 $n = 2^k$*일 때는* $W(n) = W(\frac{n}{2}) + 1$*이고* $W(n) = \lg n + 1$*임.* $n \neq 2^k$*일 때는 n이 짝수인지 홀수인지에 따라서 나뉜 배열의 길이를 고려하면* $W(n) = W(\lfloor \frac{n}{2} \rfloor) + 1$*임(최악의 경우이므로.). 즉, 최악의 경우 임의의 n에 대해 복잡도 함수는* $W(n) = W(\lfloor \frac{n}{2} \rfloor) + 1$*임.*

수학적 귀납법으로 증명함.

Induction Base. $W(1) = 1 = \lfloor \lg 1 \rfloor + 1$.

Induction Hypothesis. $1 \leq k < n$인 k에 대해 $W(k) = \lfloor \lg k \rfloor + 1$가 성립함.

Induction Step. $W(n) = W(\lfloor \frac{n}{2} \rfloor) + 1$를 n이 짝수인 경우와 홀수인 경우로 나눠서 $W(n) = \lfloor \lg n \rfloor + 1$임을 보이면 됨. 이때 n이 홀수이면 $\lfloor \lg(n-1) \rfloor = \lfloor \lg n \rfloor$인 것을 활용함.

2.3. 합병 정렬

2.3.1. 합병 정렬

1. 합병 정렬

합병 정렬(Merge sort)은 배열을 재귀적으로 분할(divide)하고 합병(merge)하는 분할 정복 정렬임.

배열을 2개로 나누고 재귀적으로 정렬한 뒤, 두 배열을 적절하게 합치는 방식임.

pseudo code는 아래와 같음. 비내림차순으로 정렬함.

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

        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);
    }
}
```

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

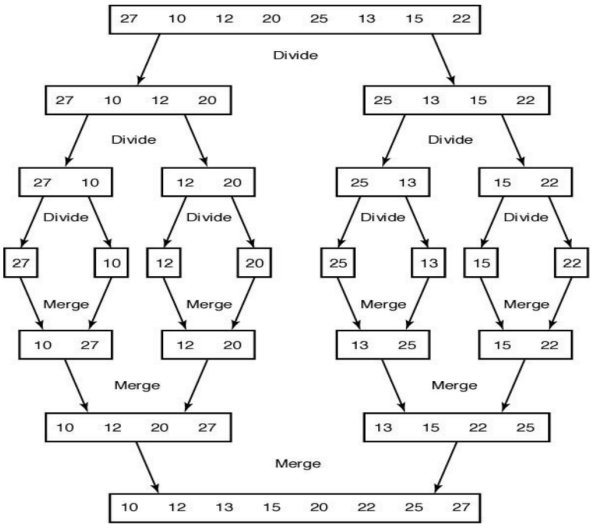
2. 성능 분석

시간에 대한 최악의 경우 복잡도를 구해보자. 원소 간의 비교를 기본 연산으로 함. 참고로 정렬 알고리즘에서는 주로 비교를 기본 연산으로 함.

코드를 보면 i/j가 특정 값을 가지면 해당 값보다 1 작은 위치까지 배열 S에 작성된 것이고, 작성 횟수와 비교 횟수가 동일함. 즉, 비교 연산만 봤을 때 최악의 경우 복잡도는 i가 h이고 j가 m+1인 경우 또는 i가 h+1이고 j가 m인 경우이므로 $W(h, m) = h + m - 1$임(h와 m에 따라서 연산 횟수가 달라짐.).

전체 알고리즘의 복잡도는 $W(h + m) = W(h) + W(m) + h + m - 1$인 것을 알 수 있음. $n = 2^k$로 가정하면, 코드에 의해 $h = m = \frac{n}{2}$임. 즉, $W(n) = 2W(\frac{n}{2}) + n - 1$임. 이를 repeated substitution으로 정리하면 $W(n) = n \lg n - n + 1 \in \Theta(n \lg n)$임.

order는 구했고, 일반적인 경우와 증명에 대해서는 복잡하므로 더 다루지 않음.



2.3.2. 공간복잡도 개선

1. 제자리정렬 알고리즘

제자리정렬(In-place sort) 알고리즘은 정렬 시에 부가적인 메모리 공간이 사용되지 않는 정렬임.

합병 정렬 알고리즘은 제자리정렬 알고리즘이 되도록 구현할 수는 없지만, 위의 pseudo code를 수정해 공간복잡도를 낮출 수는 있음.

2. 공간복잡도 개선

기존의 메모리 코드를 생각해 보면 추가적인 메모리는 $n+(\frac{1}{2})n+(\frac{1}{2})^2n+(\frac{1}{2})^3n+\cdots=2n$*만큼 필요함 (메모리를 사용하고 반납하므로.). 즉, 공간복잡도가* $2n \in \Theta(n)$ *임.*

부가적인 메모리 공간이 필요한 이유는 merge 시에 기존 값을 덮어쓰게 될 수 있기 때문이므로 merge에 대한 공간만 별도로 사용하면 됨. 매번 새로운 배열을 생성해서 전달하는 대신, n만큼의 길이를 가지는 배열 U를 선언해 두고 merge한 결과를 U에 저장한 뒤 이를 배열 S에 반영하는 식으로 구현할 수 있음.

pseudo code는 아래와 같음. 비내림차순으로 정렬함.

```
// mergesort2는 배열을 생성하는 부분만 빼면 되므로 pseudo code는 생략함.

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)
        move S[j] through S[high] to U[k] through U[high];
    else
        move S[i] through S[mid] to U[k] through U[high];

    move U[low] through U[high] to S[low] through S[high];
}
```

2.4. 퀵 정렬

2.4.1. 퀵 정렬

1. 퀵 정렬

퀵 정렬(Quick sort)은 중심점(pivot)을 기준으로 배열을 나누고(partition) 각 부분을 재귀적으로 정렬(sort)하는 분할 정복 정렬임.

partition은 pivot을 기준으로 작은 것은 왼쪽 부분으로, 큰 것은 오른쪽 부분으로 나누는 과정임. partition에 의해 pivot에 해당하는 요소의 위치가 결정되므로, 배열을 정렬하는 핵심 연산으로 생각할 수 있음.

pseudo code는 아래와 같음. 비내림차순으로 정렬함. 주어진 배열에서 맨 앞 요소를 pivot으로 지정하고, j는 pivot보다 작은 요소가 들어갈 위치이고, i는 순차적으로 확인 중인 요소의 위치임. 마지막에 pivot의 위치는 j가 됨.

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

```
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
}
```

2. 성능 분석
시간에 대한 최악의 경우 복잡도를 구해보자. 원소 간의 비교를 기본 연산으로 함.

항상 비교가 배열의 처음부터 끝까지 수행되므로, partition 연산의 복잡도는 every case에 대해 n-1, 즉 $T(n) = n - 1$*임.*

정렬 자체의 복잡도는 pivot의 마지막 위치에 의해 다르게 결정됨. 즉, pivot의 위치를 $p(1 \leq p \leq n)$ *라고 하면, every case에 대한 복잡도를* $W(n) = W(p-1) + W(n-p) + (n-1)$*로 표현할 수 있음. 이때 p가 어떤 값을 가지는 경우가 최악인지를 귀납적으로(이것 저것 시도하여) 확인해 보면, p가 1 또는 n 일 때 최악인 것을 확인할 수 있음. 나뉜 부분마다 n-1번의 연산이 수행되므로 나뉜 부분이 적을수록 성능이 떨어지는 것임.*

$W(0) = W(1) = 0$*이므로, p가 1 또는 n인 경우는* $W(n) = W(n-1) + (n-1)$*이고 이를 repeated substitution으로 정리하면* $T(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$*임.*

3. 증명
p가 1 또는 n일 때 최악임을, 즉 모든 n에 대해 $W(n) \leq \frac{n(n-1)}{2}$*이 성립함을 보여야 함. 수학적 귀납법을 사용함.*

Induction Base. $W(0) = 0 \leq \frac{0(0-1)}{2}$*임.*

Induction Hypothesis. $0 \leq k < n$*인 k에 대해* $W(k) \leq \frac{k(k-1)}{2}$*이 성립함.*

Induction Step. $W(n) = W(p-1) + W(n-p) + (n-1)$*이므로,* $W(n) \leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + (n-1) = (p - \frac{n+1}{2})^2 + \frac{(n+1)(n-1)}{4} = f(p)$*임. f(p)는 p가 1 또는 n일 때 최댓값* $\frac{n(n-1)}{2}$*를 가지므로, 임의의 n에 대해* $W(n) \leq \frac{n(n-1)}{2}$*가 성립함.*

2.4.2. 평균적 경우의 성능

퀵 정렬을 이름과는 다르게 가장 빠른 정렬 알고리즘은 아님. 특히 최악의 경우에는 순차 정렬과 성능이 동일함. 물론 평균적인 경우에서는 성능이 더 좋음. 이제 평균적인 경우에 대한 시간에 대한 성능을 구해보자. 여기에서도 비교 연산을 기본 연산으로 함.

위에서 보인 것처럼 임의의 n에 대해 $W(n) = W(p-1) + W(n-p) + (n-1)$*임. 이때 평균을 계산하기 위한 각 경우는 p값(pivot의 위치)에 따라 나뉨. p는 1부터 n까지를 가질 수 있고, 각 값을 가질 확률을* $\frac{1}{n}$*이라고 가정함.*

평균적 경우는 각 경우에 대한 기본 연산의 연산 횟수와 확률을 곱해서 전부 더한 것이므로, $A(n) = \sum_{p=1}^{n} \frac{1}{n}(A(p-1) + A(n-p) + (n-1))$*임.*

p에 1부터 n까지를 대입해 보면 $W(p-1)$*과* $W(n-p)$*가 같으므로,* $A(n) = \sum_{p=1}^{n} \frac{1}{n}(2A(p-1) + (n-1))$*로 정리할 수 있음.* $\frac{1}{n}$*을 넘긴 뒤, n-1을 대입한 식과 빼주고, 양변에* $\frac{1}{n(n+1)}$*을 곱해 정리하면,* $\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$*임.*

$a_n = \frac{A(n)}{n+1}$으로 하고, *repeated substitution*으로 정리하면 $a_n = 2\sum_{k=1}^{n} \frac{1}{k+1} - \sum_{k=1}^{n} \frac{1}{k(k+1)}$임. 이때 $\sum_{k=1}^{n} \frac{1}{k+1}$는 $\ln n$으로 근사되고, $\sum_{k=1}^{n} \frac{1}{k(k+1)}$는 *dominant factor*가 아니므로 *order*를 구할 때 무시할 수 있음. 즉, $a_n \approx 2\ln n$, $A(n) \approx 2(n+1)\ln n \in \Theta(n \lg n)$임.

즉, 평균적 경우 성능은 $\Theta(n \lg n)$임.

참고로, 비교 연산을 기본 연산으로 했을 때 정렬 알고리즘이 가지는 최선의 성능은 $\Theta(n \log n)$이고, 이것보다 성능이 더 좋을 수는 없음(증명되어 있음). 합병 정렬, 퀵 정렬, heap 정렬이 특정 경우에 대해 $\Theta(n \log n)$에 포함됨.

2.5. 행렬 곱

행렬 곱을 수행하는 알고리즘들.

2.5.1. 단순 곱

1. 단순 곱
단순히 행렬 곱의 정의대로 계산할 수 있음.

*pseudo code*는 아래와 같음.

```
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];
        }
}
```

2. 성능 분석
시간에 대한 복잡도를 구해보자.

두 값의 곱셈 연산을 기본 연산으로 하든, 가장 안쪽 덧셈 연산을 기본 연산으로 하든 당연하게도 every case에 대해 $\Theta(n^3)$의 성능을 가짐.

2.5.2. 스트라센의 행렬 곱셈 알고리즘

1. 스트라센의 행렬 곱셈 알고리즘
스트라센(Strassen)의 행렬 곱셈 알고리즘은 기본적으로 행/열의 길이 가 2의 거듭제곱인 행렬에 대해, 4개의 부분행렬(submatrix)로 나누어 재귀적으로 행렬 곱을 구하는 알고리즘임.

아래와 같이 행렬을 4개의 부분행렬로 나누고, $M_1 \sim M_7$을 계산하고 이를 활용하여 행렬 c의 각 부분 행렬을 구할 수 있음.

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

$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
$M_2 = (A_{21} + A_{22}) \times B_{11}$
$M_3 = A_{11} \times (B_{12} - B_{22})$
$M_4 = A_{22} \times (B_{21} - B_{11})$
$M_5 = (A_{11} + A_{12}) \times B_{22}$

$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

*pseudo code*는 아래와 같음.

```
void strassen (int n, n×n_matrix A, n×n_matrix B, n×n_matrix& C) {
    if (n <= threshold)
        compute C = A × B using the standard algorithm;
    else {
        partition A into 4 submatrices A11, A12, A21, A22;
        partition B into 4 submatrices B11, B12, B21, B22;
        compute C = A × B using Strassen's method;

        // example recursive call: strassen(n/2, A11+A22, B11+B22, M1)
    }
}
```

행렬의 크기가 작을 때는 단순 곱의 성능이 더 좋지만, 크기가 커지면 스트라센의 방법이 성능이 더 좋으므로 threshold를 지정하여 사용함.

2. 성능 분석
시간에 대한 복잡도를 구해보자.

한 번의 기본 곱셈 연산을 기본 연산으로 함. 임계값(threshold)이 1이라고 가정하자(임계값은 order의 결정에 영향을 미치지 못함.). $T(1) = 1$*임.* $M_1 \sim M_7$*을 구할 때 곱셈 연산이 7번 발생하므로, every case에 대해* $n = 2^k > 1$*일 때* $T(n) = 7T(\frac{n}{2})$*,* $T(n) = 1$*임. repeated substitution으로 정리하면* $T(n) = 7^{\log_2 n} = n^{\lg 7} = n^{2.81} \in \Theta(n^{2.81})$*로, 단순 행렬 곱셈보다 성능이 좋음.*

한 번의 기본 덧셈/뺄셈 연산을 기본 연산으로 했을 때도 동일함. 이 경우 $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$*,* $T(1) = 0$*이므로 정리하면* $T(n) \in \Theta(n^{2.81})$*임.*

즉, every case에 대해 성능이 $\Theta(n^{2.81})$*임.*

3. DP

3.1. DP

3.1.1. DP

1. DP
동적 계획법(DP, Dynamic Programming)은 문제를 여러 개의 작은 부분으로 나눈(divide) 뒤, 작은 문제부터 해결하고 그 결과를 활용하여 큰 문제를 해결하는 문제 접근 방식임.

DP는 상향식 접근법(Bottom-up approach)임.

DP에서는 주로 반복문을 사용함.

DP는 큰 문제의 답을 작은 문제의 답으로 찾을 수 있는 문제, 큰 문제에서 작은 문제의 답을 참조하는 문제, 나눈 부분들 사이에 상관관계가 있는 문제를 해결하는 데에 적합한 방식임.

2. DP 적용 방법
DP를 적용할 때는 1. 문제의 대한 재귀식(점화식)을 정의하고, 2. 상향식 접근법으로 작은 문제부터 해결함.

DP에서는 중복 작업을 피하거나 작은 문제의 해답을 활용하기 위해 작은 문제의 해답을 배열, 테이블 등에 저장하여 사용함.

3. top-down DP
엄밀히 보면, DP는 2가지로 나눌 수 있음.

1) Tabulation(Tabulation method)
순수한 bottom-up 방식. 연산 결과를 table에 작성하고 사용함.

2) Memoization
DP인데 top-down으로 푸는 방법. 즉, DQ와 섞인 것임.

DQ처럼 재귀 호출을 하는데, 연산 결과를 memo해 둠. 연산 시작 전에 if문으로 연산 결과가 존재하는지 검사하여 존재하면 연산을 다시 하는 대신 값을 가져다 씀.

분할 정복에서는 성능 분석을 주로 봤다면, DP에서는 알고리즘, 점화식 이해가 중점임. 어차피 반복문으로 구현되므로 구조와 성능 분석은 별 거 없음.

문제를 잘 쪼개는 것이 중요함. DP를 활용한 문제 풀이의 핵심은 점화식을 작성하는 것임. 직관적으로 바로 와닿는 문제도 있지만, 알아서 잘 쪼개야 하는 경우도 있음. 처음 보는 문제는 직접 예시를 들어 시도해보며 확인하는 방식이 효과적임.

3.1.2. 최적의 원칙

1. 최적화 문제
최적화 문제(Optimization Problem)는 주어진 문제에 대해서 하나 이상의 해답(candidate solution)이 도출되고 그 중 최적인 것을 구하는 유형의 문제임.

예를 들어, 최단경로 문제는 최적화 문제임.

2. 최적의 원칙
어떤 문제에 대한 최적해가 해당 문제를 쪼갠 하위 문제의 최적해를 항상 포함한다면, 그 문제는 최적의 원칙(The principle of optimality)이 적용된다고 함. 또는 점화식이 optimal substructure라고도 함.

즉, 최적화 문제 중 부분 문제의 최적해를 사용해 전체 문제의 최적해를 구할 수 있는 경우를 말함.

당연하게도 모든 최적화 문제에 최적의 원칙이 적용되는 것은 아님. 문제를 쪼개서 구했을 때 각 경우에 대해 답이 달라진다면 optimal substructure가 아닌 것. 예를 들어, 최장거리를 구하는 문제에서는 전체 문제의 최적해가 $\{v_1, v_2, v_3\}$일 때, v_1과 v_2 사이의 최장거리는 다르게 정의될 수 있음. (ppt에 구체적인 예시가 있음.)

점화식이 나왔는데 이 문제가 최적화 문제이고, 최적의 원칙이 적용된다면, DP로 문제를 해결할 수 있음. 이때 분할정복을 시도하지 말고 바로 DP로 가는 것이 좋음. *만약 점화식이 나왔는데 이 문제가 최적화 문제이고, optimal substructure가 아니라면 추후에 배울 다른 방법을 사용해야 함.*

DP를 활용한 최적화 문제에서는 각 값들 사이의 토너먼트를 통해 최적의 값을 찾음.

3.2. 이항계수

3.2.1. 이항계수

1. 이항계수
이항계수(Binomial Coefficient)는 조합의 개수를 수학적으로 나타내는 개념으로, 아래와 같이 계산함.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \, (0 \le k \le n)$$

이항계수가 가지는 대표적인 성질 중 하나는 아래와 같이 2개의 이항계수로 쪼개서 계산이 가능하다는 것임.

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

2. 구현
분할 정복 또는 DP를 사용해 이항 계수를 구할 수 있음.

참고로 $\binom{n}{0} = \binom{n}{n} = 1$임.

3.2.2. 분할 정복

1. 분할 정복
분할 정복으로 간단히 이항계수를 구할 수 있음.

*pseudo code*는 아래와 같음

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

2. 성능 분석
시간에 대한 복잡도를 구해보자.

함수 호출을 기본 연산으로 했을 때, $\binom{n}{r}$*에서 발생하는 연산 횟수는* $2\binom{n}{r}-1$*임. 이를 수학적 귀납법으로 증명해보자.*

Induction Base. $\binom{1}{0} = \binom{0}{0} = 1$*임.*

Induction Hypothesis. $T(n,r) = 2\binom{n}{r} - 1$*임.*

Induction Step. $T(n+1,r) = T(n,r) + T(n,r-1) = 2\binom{n+1}{r} - 1$*이 성립함.*

자세한 성능 분석을 하지 않아도, 팩토리얼이 등장하므로 지수함수보다 성능이 안 좋은 것을 알 수 있음. 재귀 호출에 의한 중복 연산이 존재함.

3.2.3. DP

1. DP
DP에서의 설계는 우선 재귀식을 정의하고, 상향식으로 문제를 해결하는 절차를 거침.

우선 재귀식을 정의함. 2차원 배열을 사용하여, $B[i][j]$*에* $\binom{i}{j}$*를 저장한다고 하면 아래의 재귀식을 당연하게 작성할 수 있음.*

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

정의한 재귀식을 바탕으로 상향식 접근을 함. 2차월 배열 B를 표에 나타내 보면, $B[i][j]$*를 구하기 위해서는 바로 위 원소와 왼쪽 위 원소가 필요함을 알 수 있음.*

*pseudo code*는 아래와 같음.

```
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];
}
```

3. 성능 분석
시간에 대한 복잡도를 구해보자. j 반복문 내부를 기본 연산으로 했을 때, 표를 보면 각 원소별로 연산이 한 번씩 발생함. 즉, 몇 개의 원소를 거쳤는지를 세면 됨.

세어 보면 $\binom{n}{r}$의 연산 횟수는 $1+2+\cdots+(r-1)+r+\cdots+r$이고, 여기서 r은 $n-k+1$번 등장함. 즉, every case에 대해 $T(n,r)=nr-\frac{1}{2}r^2+\frac{1}{2}r\in\Theta(nr)$임($r\le n$이므로.).

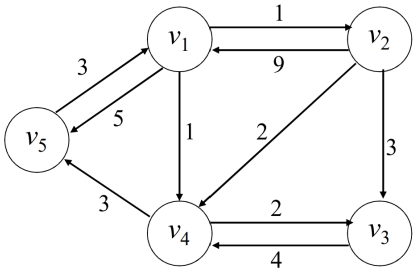
n단계를 구할 때는 n-1단계의 값만이 필요하므로 일차원 배열로 사용하고, $\binom{n}{r}=\binom{n}{n-r}$임을 이용해서 연산량을 최적화하여 개선할 수 있음.

3.3. 최단경로

3.3.1. 최단경로

1. 최단 경로
최단 경로(Shortest Path) 문제는 가중치가 가장 적은 경로(path)를 찾는 문제임. 여기서는 가중치 방향 그래프에서의 최단경로에 대해서 다룸.

여기서에서는 최단 경로를 해결하는 알고리즘으로 brute force와 floyd를 다룸.



2. 그래프 기본 개념
최단 경로 등 그래프 관련 문제에서는 그래프가 입력으로 들어옴. 여기서는 그래프를 인접 행렬로 표현함. 즉, $W[i][j]$는 i에서 j로 가는 간선(edge)의 유무와 가중치를 나타냄.

그래프는 순서쌍(tuple)으로 나타내고, 하나의 그래프는 $G=(V,E)$로 표기함. V는 vertex(노드, 정점)의 집합으로 $V=v_1,\cdots,v_n$와 같이 정의하고, E는 edge(arc, 간선)의 집합으로 $E=(v_1,v_2),\cdots,(v_4,v)5)$와 같이 정의함. 이때 각 집합이 가지는 원소의 개수는 $|V|$, $|E|$로 작성함.

경로(path)는 []로 나타냄. 예를 들어, $[v_2,v_4,v_5]$ 등으로 작성할 수 있음.

cycle은 방문했던 정점을 다시 방문하는 경로이고, cycle이 있는 그래프를 cyclic graph라고 함.

경로의 길이(length)는 경로에 있는 간선의 weight를 전부 더한 것임.

그래프 관련 구체적인 기본 개념은 이산수학, 자료구조 필기를 참고하자.

3.3.2. Brute force

1. Brute force
Brute force는 가능한 경우의 수를 모두 탐색하여 문제를 해결하는 방법임. 즉, 최단경로에서는 한 정점에서 다른 정점으로 가는 경로의 길이를 모두 구한 뒤, 최소 길이를 도출하는 방법임.

2. 성능 분석
한 정점에서 다른 정점으로 가는 경로의 개수만 대략적으로 구해보자. 그래프가 n개의 정점을 가지고 있고, 모든 정점이 서로 간선으로 연결되어 있다고 가정함. 이때 한 정점에서 다른 정점으로 가는데 모든 다른 정점을 거치는 경로의 개수만 구해 봐도, 출발지와 도착지를 제외한 n-2개의 정점을 나열하므로 $(n-2)!$*개임.*

성능이 $\Omega((n-2)!)$*이고, 당연하게도 더 안 좋을 수도 있으므로 일반적인 경우 최단경로에서 brute force는 적용하기에 현실성이 없음.*

3.3.3. Floyd's algorithm

1. Floyd's algorithm
Floyd's algorithm은 DP를 활용해 최단 경로를 구하는 알고리즘으로, 중간 정점이 없는 상태에서 시작해 중간 정점을 추가하며 최단 경로를 계산함.

2. 최단거리 구하기
최단경로를 구하기에 앞서, 최단거리부터 구해봄. 최단거리를 구할 수 있으면 각 정점을 잘 저장해서 최단경로를 얻을 수 있음.

우선 그래프의 인접행렬은 아래와 같이 W로 표기함. ∞*는 사용 환경에 따라 적절한 값을 사용하면 됨.*

$$W[i][j] = \begin{cases} weight & \text{if an edge exists} \\ 0 & \text{if } i = j \\ \infty & \text{if the edge doesn't exists} \end{cases}$$

그래프의 최단경로 길이는 $D^k[i][j]$*로 표기함. 이는* $0 \le k \le n$*,* $\{v_1, v_2, \cdots, v_k\}$*의 정점들만을 사용해서* v_i*에서* v_j*로 가는 최단경로의 길이를 나타냄. k가 0인 경우는 두 정점 사이에 다른 정점을 거치지 않는 경우임. 경로가 존재하지 않으면* ∞*을 값으로 가지고, 자기 자신에 대한 값은 0임.*

D를 활용하여 아래와 같이 점화식을 구할 수 있음. v_k*가 포함되는 경우(i부터 k까지와 k부터 j까지를 더한 것.)와 포함되지 않는 경우 중 더 나은 경우를 선택한 것임. 이 점화식에 따라 상향식으로 원하는 지점의 값을 구할 수 있음.*

$$D^k[i][j] = min(D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j])$$

$D^{(0)} = W$*이고, k번째를 구하려면 k-1번째들만 알면 됨. 이때 k값에 따라 원소가 선택되므로 k-1번째에 해당하는 값들은 전부 구하도록(모든 부분을 완성하도록) 코드를 짤 수 있음.*

어차피 어떤 단계의 계산에 필요한 것은 바로 이전 단계의 값들뿐이므로, 이차원 배열 하나만 써도 됨. 생각해 볼 만한 의문점으로는, 현재 단계에서 수정한 부분을 참조해서 계산하는 경우가 존재할 수도 있지 않은가임. 하지만 생각해 보면 배열에서 가져다 쓰는 위치는 값이 수정되지 않은 부분임. $D_{ij}^k = min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$*에서* D_{ij}^{k-1}*는 이제 넣을 값이므로 상관없음.* $D_{ik}^{k-1} + D_{kj}^{k-1}$*가 문제인데, i와 j에 각각 k를 넣은 경우를 생각해 보면* $D_{Ok}^{k-1} = 0$*이므로 항상* D_{ij}^{k-1}*와 동일한 값을 가짐. 즉, 해당 부분은 값이 수정되지 않은 부분임.*

참고로 이때 i와 k 사이에서 선택된 정점이 k와 j 사이에서 중복 선택되는 경우는 최단거리가 아니므로 도출되지 않음.

pseudo code는 아래와 같음.

```
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]);
}
```

시간에 대한 복잡도를 생각해 보자. 가장 안쪽 반복문의 내부 연산을 기본 연산으로 하면, every case 에 대해 당연하게도 $T(n) = n^3 \in \Theta(n^3)$임.

3. 최단경로 구하기
아래와 같이 정의된 배열 P를 사용해 두 정점 사이에 존재하는 정점 하나를 저장하고, 재귀적으로 각 정점을 확인하면 최단경로를 구할 수 있음.

$$P[i][j] = \begin{cases} v_i와\ v_j\ 사이\ 정점\ 중\ 가장\ 큰\ 인덱스\ 값 & 정점이\ 존재하는\ 경우 \\ 0 & 정점이\ 존재하지\ 않는\ 경우 \end{cases}$$

floyd() 함수에는 아래와 같이 P에 값을 저장하는 코드만 추가하면 됨.

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

아래와 같이 P로부터 경로를 재귀적으로 얻을 수 있음.

```
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);
    }
}
```

그래프의 경우 여기서 한 것처럼 정점을 하나씩 추가하는 방식으로 DP를 적용해 볼 수 있음.

3.4. 연쇄 행렬 곱셈

3.4.1. 연쇄 행렬 곱셈

1. 연쇄 행렬 곱셈
연쇄 행렬 곱셈(Chained Matrix Multiplicaton)은 여러 개의 행렬을 곱하는 계산임.

$i \times j$ 행렬 A와 $j \times k$ 행렬 B가 있을 때, $AB = C$를 계산한다고 하자. 이때 C의 각 원소에 대해 A의 각 행의 원소와 B의 각 열의 원소를 곱하므로 하나의 원소에서 발생하는 곱셈 연산은 j번임. C는 $i \times k$ 행렬이므로 A와 B를 곱할 때 발생하는 전체 곱셈 연산은 ijk번임. 즉, 서로 행과 열의 크기가 다른 연쇄 행렬 곱셈 $A_1A_2 \cdots A_n$을 계산한다고 하면 어떤 행렬들끼리 먼저 곱하는지에 따라 전체 연산량이 달라질 수 있음.

2. Minimum Multiplication
Minimum Multiplication 알고리즘은 n개의 행렬을 곱하는 데에 필요한 최소 연산량을 구하고, 해당 연산량을 구하는 연산 순서를 찾는 알고리즘임.

연쇄 행렬 곱셈에서는 각 원소에 대한 곱셈 연산이 최소가 되는 것이 중요함. 여기에서는 어떤 연쇄 행렬 곱셈이 주어졌을 때 최소 연산량과, 이때의 최적의 방법(곱하는 순서)을 구하는 알고리즘을 알아봄. 당연히 이때 곱하는 행렬들 자체의 순서를 바꾸는 것이 아니라, 괄호를 어떻게 잘 칠 지를 정하는 것임.

연쇄 행렬 곱셈은 어떤 순서로 곱해야 최적인지를 구하는 것이므로, 최적화 문제임.

또한 연산량은 행렬의 크기에만 영향을 받으므로 입력으로는 각 행렬의 전체 원소 값이 아니라, 각 행렬의 크기만 받으면 됨.

3.4.2. Brute force

*brute force*로 연쇄 행렬 곱셈의 최적 곱셈 순서를 구하려면 곱셈에 대한 각 경우의 수를 전부 확인해야 함.

n개의 행렬 $A_1A_2 \cdots A_n$을 계산할 때 곱셈에 대한 경우의 수를 t_n이라고 하자. A_1과 A_n을 마지막에 곱하는 경우를 생각하면 각각 $t_n - 1$의 경우의 수가 존재함. A_1과 A_n을 마지막에 곱하는 경우 말고도 괄호를 씌우는 방법은 다양하므로, $t_n \geq 2t_{n-1}$라고 할 수 있음. $T_2 = 1$이므로 이를 정리하면 $t_n \geq 2t_{n-1} > \cdots \geq 2^{n-2}t_2 = 2^{n-2}$임. 즉, 최소한 2^{n-2}개의 경우의 수(실제로는 훨씬 많음.)를 일일이 확인해야 하므로 $T(n) \in \Omega(2^n)$임. 즉, 못 써먹음.

수업에서는 이렇게 설명했지만, 잘 생각해 보면 $t_n = (n-1)t_{n-1}$이므로, 정리하면 $t_n = (n-2)!$임을 알 수 있음. 아무튼 실사용할 수 있는 성능이 아님.

이는 각 경우에 대한 계산을 할 때 매번 새롭게 계산하는 것이므로 겹치는 연산이 발생하기 때문에 비효율적임.

3.4.3. DP

연쇄 행렬 곱셈 $A_1A_2 \cdots A_n$의 *Minimum Multiplication*을 DP로 구하는 알고리즘을 살펴보자.

1. 점화식
d_K를 행렬 A_k의 열의 크기라고 하자. 행렬 곱은 앞 행렬 열의 크기와 뒷 행렬 행의 크기가 같아야 하므로 A_k의 행의 크기는 d_{k-1}임. 추가로 d_0은 A_1의 행의 크기로 함.

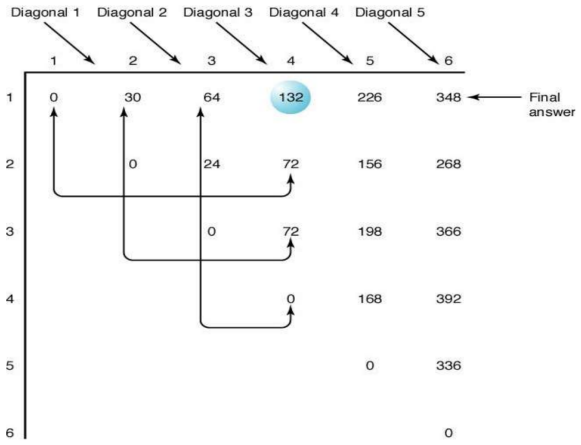
$1 \leq i \leq j \leq n$에 대해 $M[i][j]$를 A_i부터 A_j까지의 행렬을 곱하는 데에 필요한 곱셈 연산의 최소 횟수라고 하면, $M[i][j]$는 괄호를 치는 각각의 경우를 비교하여 얻을 수 있고, 특정 경우에 대한 연산량은 $d_{i-1}d_kd_j$로 구할 수 있음. 즉, 아래와 같이 점화식을 정의할 수 있음.

$$M[i][j] = minimum_{i \leq k \leq j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_kdj)$$
$$M[i][i] = 0$$

2. pseudo code
*pseudo code*는 아래와 같음. $A_1A_2 \cdots A_n$에 대한 최적 곱을 구하는 상황이고, 입력으로 $d_0 \sim d_n$이 들어옴.

```
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]);
                (i <= k <= j-1인 k에 대해 계산.)
            P[i][j] = 최소치를 주는 k의 값
        }
    return M[1][n];
}
```

$M[i][j]$는 $i = j$인 대각 성분부터 시작해, 루프가 돌면서 대각선으로 한 줄씩 채워져 상삼각행렬로 구성됨. 이때 각 성분을 구할 때는 해당 성분을 도착지로 하는 부분 상삼각행렬의 원소들(왼쪽과 아래쪽에 있는 원소들)이 전부 필요함. 결과적으로 구해야 하는 값은 마지막에 구해지는 맨 오른쪽 맨 위 값임.



floyd's algorithm에서와 같이 곱하는 순서는 P 배열에 저장함. $P[i][j]$에는 A_i부터 A_j까지를 곱할 때 최적으로 갈라지는 지점(마지막에 곱하는 지점)이 저장됨. P 배열의 값을 아래와 같이 재귀적으로 추출하여 행렬의 곱셈 순서를 얻을 수 있음. order(i,j)는 A_i부터 A_j를 곱하는 최적 곱을 괄호로 출력함.

```
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 << ")";
        }
}
```

3. 성능 분석

시간에 대한 복잡도를 구해보자. 항상 M 배열 전체를 구성하므로 every case를 구할 수 있음. 기본연산은 k 루프 내부의 연산으로 함.

diagonal 루프에서 diagonal 변수는 $1 \sim n-1$의 값을 가지고, i 루프는 $n - diagonal$번, k 루프는

$j-1-i+1=j-i=diagonal$번 돌게 됨. 즉, 정리하면 아래와 같음.

$$T(n)=\sim_{diagonal=1}^{n-1}(n-diagonal)\times diagonal=\frac{n(n-1)(n-2)}{6}\in\Theta(n^3)$$

이를 동일한 형태의 점화식을 가지는 분할 정복(재귀)으로 해결하려고 하면 중복 연산이 발생하여 성능이 크게 떨어짐.

3.5. Optimal BST

3.5.1. Optimal BST

1. BST

이진탐색트리(Binary Search Tree. BST)는 자신보다 key값이 작거나 같은 노드는 왼쪽 자식으로, 자신보다 key값이 큰 노드는 오른쪽 자식으로 보내 구성하는 binary tree임.

BST는 ordered set을 입력으로 받아 구성됨. ordered set은 각 원소들끼리의 순서가 정의되어 있는 set을 말함. 새로 입력된 노드는 key 값에 따라 왼쪽/오른쪽으로 이동하며 leaf에 도달하면 해당 위치를 자신의 자리로 함.

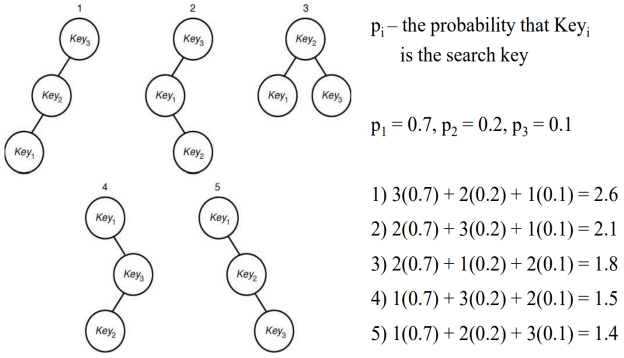
특정 노드의 depth는 root로부터 이어진 edge의 개수임. root는 depth가 0임.

임의의 두 노드의 depth 차이가 1 이하이면 balanced, 그렇지 않으면 unbalanced한 BST라고 함.

2. Optimal BST

set의 원소들을 어떤 순서로 BST에 입력하는지에 따라 BST는 다르게 구성되고, 이에 따라 노드에 대한 average search time이 달라짐. 즉, optimal BST는 average search time이 최소가 되는(평균적으로 탐색이 가장 빠른) BST임. 여기에서의 최솟값을 optimal value라고 함.

특정 노드에 대한 search time은 해당 노드를 찾기까지 수행한 비교 연산의 횟수임. 여기에서 이는 depth + 1임. 각 경우에 대해 특정 키가 등장할 확률을 가지고 average search time을 구할 수 있음.



key가 들어오는 순서에 따라 BST가 자동으로 구성되므로, 원소를 어떤 순서로 집어넣는지가 관건임. 여기에서는 결과적으로 나오는 optimal value, bst, 입력 순서를 구하는 알고리즘을 알아봄.

이때 key의 구체적인 값보다는 각 key의 대소관계가 중요하므로 노드가 n개라면 각 노드는 $1\sim n$의 값을 key로 가지는 것으로 함. 또한 key 값이 노드 수에 따라 결정되어 있으므로 입력으로는 각 key의 확률만 있으면 됨.

물론 BST의 노드는 key와 포인터 2개를 가진 구조체로 정의할 수 있고, 실제로 BST에서 탐색을 수행하는 코드는 단순히 key값을 비교하여 재귀 호출을 하는 식으로 구현할 수 있음. 하지만 이 알고리즘에서는 단순히 각 key에 대한 확률만 있으면 됨.

3.5.2. DP

1. 점화식

n개의 노드가 있다고 하자. i번째 노드는 i를 key값으로 가지고, 그 등장 확률은 p_i 임.

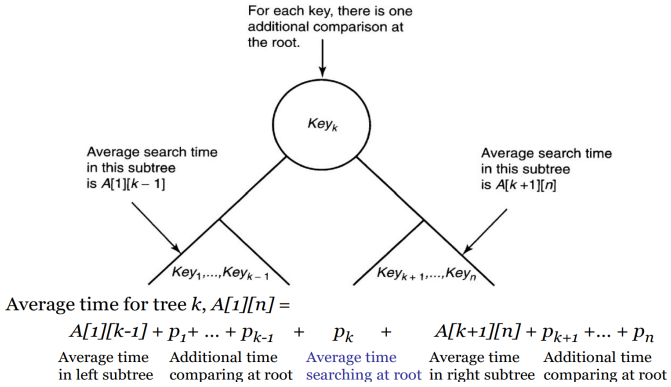
배열 A의 원소 A[i][j]에는 i번째 노드부터 j번째 노드까지를 사용해서 구성한 optimal bst의 average search time을 저장함. 즉, 본 알고리즘에서는 A[1][n]까지 구하는 것이 목표임.

이때 A[i][i]=0, A[i][i-1]=A[j][j+1]=0임.

점화식은 아래와 같음.

$$A[i][j] = min_{i \le k \le j}(A[i][k-1] + A[k+1][j]) + \sum_{m=1}^{n} p_m$$

즉, A[i][j]는 i번째부터 j번째까지의 노드 중 하나가 루트인 경우를 전부 구하고, 그 중 최솟값을 값으로 함. 이때 p_m을 더하는 것은 특정 노드가 루트가 됐을 때 각 노드는 비교가 한 번씩 더 발생하기 때문이고, 어차피 상수이므로 min 밖으로 뺀 것임.



2. pseudo code

$p_1 \sim p_n$을 입력으로 받음.

```
void optsearchtree(int n, const float p[], float& minavg, index R[][]) {
    index i, j, k, diagonal;
    float A[1..n+1][0..n];

    // 초기화
    for(i=1; i<=n; i++) {
        A[i][i-1] = 0; A[i][i] = p[i]; R[i][i] = i; R[i][i-1] = 0;
    }
    A[n+1][n] = 0; R[n+1][n] = 0;

    // 계산
    for(diagonal=1; diagonal<= n-1; diagonal++) {
        for(i=1; i <= n-diagonal; i++) {
            j = i + diagonal;
            A[i][j] = min_{k=i~j}(A[i][k-1]+A[k+1][j]) + pi + ... + pj
            R[i][j] = a value of k that gave the minimum;
        }
    }

    minavg = A[1][n]; // 반환값
}
```

A[i][j]를 구하기 위해서는 그 이전 대각선(diagonal)의 값이 필요함. 즉, A[i][i]=0부터 시작하여 A[1][n]까지 대각선 별로 연산을 함.

배열 R을 사용하여 optimal bst를 형성하는 입력 순서를 구함. R[i][j]에는 min 수행 이후 k값을 저장함. 즉, R[i][j]에는 i번째부터 j번째까지로 optimal bst를 구성했을 때의 노드가 저장됨. 배열 R을 재귀 탐색하여 optimal bst를 구할 수 있음. 또한 bst에는 부모 쪽에 있는 노드부터 넣으면 되므로 입력 순서를 또한 구할 수 있음.

3. 성능 분석
시간에 대한 복잡도를 구해보자. min 내부에서의 연산을 기본 연산으로 함. every case에 대해 구할 수 있음.

dia=n-1, i=n-dia, j=i+dia, k=j-i+1=dia-1이므로, 아래와 같음.

$$\sum_{dia=1}^{n}(n-dia)(dia+1)=\frac{n(n-1)(n+4)}{6}\in\Theta(n^3)$$

즉, every case에 대해 $\Theta(n^3)$임.

4. Geedy

4.1. Greedy

4.1.1. Greedy

1. Greedy
Greedy approach는 매 순간 최적으로 보이는 선택을 하는 접근 방식으로, 최적화 문제를 풀 때 사용함.

즉, 반복문을 돌면서 최적의 선택(local optimal)을 하고, 그 결과를 검토하여 solution에 포함시킴. 최종적으로 나온 결과가 (global) optimal solution임.

greedy의 핵심은 선택임. 이때 선택은 매 순간 최적이어야 하는데, 최적임을 판단하는 기준을 greedy criterion이라고 함. greedy로 문제를 해결하려면 greedy criterion을 찾고, 그것이 항상 성립한다는 것을 증명해야 함. DP에서는 점화식을 찾으면 문제가 해결되는 것처럼, greedy에서는 greedy criterion을 찾으면 문제가 해결됨.

local에는 최적이더라도 global에서도 최적이라는 보장은 없으므로, greedy criterion이 항상 성립한다는 것은 연역적으로 증명해야 함. global에서 최적임을 보장할 수 없다면 greedy를 썼을 때 성능이 좋지 않을 수 있음.

DQ, DP가 문제를 나누어 푸는 것(점화식)으로 함께 고려된다면, DP와 greedy는 최적화 문제에 대해 함께 고려됨. 즉, 최적화 문제면 DP로도 풀 수 있고 greedy로도 풀 수 있음. 물론 DP는 적용하려면 optimal substructure여야 함.

2. 동작 과정
greedy는 아래와 같은 단계를 거쳐 동작함.

1) Selection : 현재의 상태에서 greedy criterion을 적용하여 선택을 함.
2) Feasibility Check : 현재 얻은 답을 solution에 포함시키는 것이 적절한지 확인함.
3) Solution Check : 최종적으로 모인 solution이 해답인지 확인함.

4.2. Minimum Spanning Tree

4.2.1. Minimum Spanning Tree

Spanning Tree는 연결된 비방향성 그래프에서 순환경로를 제거하여 생성할 수 있는 트리임. Minimum Spanning Tree는 그래프에서 얻을 수 있는 spanning tree 중 전체 가중치의 합이 최소인 것을 구하는 문제임.

주요 알고리즘으로는 prim과 kruscal이 있는데, sparse graph에서는 kruscal이, dense graph에서는 prim

이 유리함.

4.2.2. Brute force

모든 정점이 간선으로 연결된 그래프가 있다고 하고, 어떤 출발점으로부터 각 정점을 한 번씩 거치며 출발점으로 돌아오지 않는 경로의 개수를 생각해 보자. 이는 총 (n-1)!가지임. 즉, 최악의 경우 팩토리얼임.

4.2.3. Prim's algorithm

1. Prim's Algorithm
Prim's Algorithm에서는 v_1을 포함하는 집합 Y를 사용하여, $v_2 \sim v_n$까지의 정점을 하나씩 최적의 선택으로 Y에 추가하여 MST를 얻는 알고리즘임.

Y는 계산이 완료된 정점을, F는 그때의 edge를 저장함. 결과적으로 얻게 되는 답은 F에 저장됨.

이때 Y와의 거리가 가장 가까운 정점부터 하나씩 Y에 포함시키고(greedy criterion), 그 결과를 Y에 대한 거리에 반영함.

2. nearest와 distance
nearest와 distance 배열을 사용함.

nearest[i] : Y에 속한 정점들 중 v_i에 가장 가까운 정점의 인덱스.
distance[i] : nearest[i]의 정점과의 거리.

3. Pseudo Code
pseudo code는 아래와 같음.

```
void prim(int n, const number W[][], set_of_edges& F) {
    index i, vnear; number min; edge e;
    index nearest[2..n]; number distance[2..n];
    F = 공집합;
    for(i=2; i <= n; i++) { // 초기화
        nearest[i] = 1; // vi에서 가장 가까운 정점을 v1으로 초기화
        distance[i] = W[1][i]; // vi과 v1을 잇는 이음선의 가중치로 초기화
    }

    repeat(n-1 times) { // n-1개의 정점을 Y에 추가한다
        min = "infinite";
        for(i=2; i <= n; i++) { // 각 정점에 대해서
            if (0 <= distance[i] <= min) { // distance[i]를 검사하여
                min = distance[i]; // 가장 가까이 있는 vnear을
                vnear = i; // 찾는다.
            }
        }
        e = edge connecting vertices indexed by vnear and nearest[vnear];
        add e to F;
        distance[vnear] = -1; // 찾은 노드를 Y에 추가한다.
        for(i=2; i <= n; i++) {
            if (W[i][vnear] < distance[i]) { // Y에 없는 각 노드에 대해서
                distance[i] = W[i][vnear]; // distance[i]를 갱신한다.
                nearest[i] = vnear;
            }
        }
    }
}
```

4. 성능 분석
시간에 대한 성능 분석을 하자.

every case에 대해 반복문이 수행되므로, $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$*임.*

4.2.4. Kruskal's algorithm

1. Kruskal's Algorithm
Kruskal's Algorithm은 각 노드를 독립된 disjoint set으로 생각하고, 비내림차순(오름차순)으로 정렬한 간선을 하나씩 사용하여 disjoint set을 합쳐서 MST를 얻는 알고리즘임.

2. Pseudo Code
pseudo code는 아래와 같음.

```
void kruskal(int n, int m, set_of_edges E, set_of_edges& F) {
    index i, j;
    set_pointer p, q;
    edge e;
    Sort the m edges in E by weight in nondecreasing order;
    F = 공집합;
    initial(n);
    while (number of edges in F is less than n-1) {
        e = edges with least weight not yet considered;
        i, j = indices of vertices connected by e;
        p = find(i);
        q = find(j);
        if (!equal(p,q)) {
            merge(p,q);
            add e to F;
        }
    }
}
```

3. 성능 분석
시간에 대한 성능 분석을 함.

kruscal algromithm의 성능은 disjoint set이 어떻게 구현되었는지에 따라 달라짐. 특히 merge시에 두 set의 깊이를 고려하여 더 작은 쪽을 아래에 들어가도록 구현하여 성능을 개선할 수 있음.

또한 disjoint set에서 루트에 한 개의 간선으로 연결되도록 하는 path compression을 사용하면 find()의 최악의 경우를 상수로 떨어뜨릴 수 있음.

$m \geq n-1$*이므로 정렬이 dominant factor임. m개의 간선의 정렬에 대한 시간복잡도는* $\Theta(m \log m)$*임. 이때 모든 정점이 다른 정점과 간선으로 연결되는 최악의 경우, 정점이 n개이면* $m = \frac{n(n-1)}{2}$*이므로,* $\Theta(n^2 \log n)$*이 됨.*

즉, 성능이 $\Theta(n \log n)$*과* $\Theta(n^2 \log n)$ *사이에서 나옴.*

4.3. Dijkstra's algorithm

4.3.1. Dijkstra's algorithm

1. Dijkstra's Alogrithm
Dijkstra's Alogrithm은 한 특정 정점에서 다른 정점으로 가는 모든 최단 경로를 구하는 알고리즘임.

prim과 마찬가지로 Y는 계산이 완료된 정점을, F는 그때의 edge를 저장함.

2. touch와 length

touch와 length 배열을 사용함.

touch[i] : 정점 v_i에서 Y와의 거리가 최소가 되는 정점. length[] : touch[i]에서의 거리.

3. Pesudo Code
pseudo code는 아래와 같음.

```
void dijkstra (int n, const number W[][], set_of_edges& F) {
    index i, vnear; edge e;
    index touch[2..n]; number length[2..n];
    F = 공집합;
    for(i=2; i <= n; i++) { // For all vertices, initialize v1 to be the last
        touch[i] = 1; // vertex on the current shortest path from v1,
        length[i] = W[1][i]; // and initialize length of that path to be the
    } // weight on the edge from v1.
    repeat(n-1 times) { // Add all n-1 vertices to Y.
        min = "infinite";
        for(i=2; i <= n; i++) { // Check each vertex for having shortest path.
            if (0 <= length[i] <= min) {
            min = length[i];
            vnear = i;
            }
        }
        e = edge from vertex indexed by touch[vnear]
            to vertex indexed by vnear;
        add e to F;
        for(i=2; i <= n; i++) {
            if (length[vnear] + W[vnear][i] < length[i]) {
            length[i] = length[vnear] + W[vnear][i];
            touch[i] = vnear; // For each vertex not in Y, update its shortest
            } // path. Add vertex indexed by vnear to Y.
        }
        length[vnear] = -1;
    }
}
```

4. 성능 분석
$\Theta(n^2)$*의 성능을 가짐.*

5. Backtracking

5.1. Backtracking

5.1.1. Backtracking

1. Backtracking
Backtracking은 상태공간트리에서 깊이 우선 탐색(Depth-first Searh)을 하는데, 유망한 노드에 대해서만 작업을 계속 수행하고 유망하지 않은 노드와 그 자식은 작업을 수행하지 않는 접근 방식임.

greedy에서는 greedy criterion이 중요했다면, backtracking에서는 아래의 것들이 중요함.
1) 상태 공간 트리의 구성 방법.
2) 유망함을 판단하는 기준.

backtracking은 정해진 조건을 만족시키는 임의의 답을 찾기 위한 방식으로, 기본적으로는 최적화 문제

에 대한 방식은 아님. 물론 조건을 만족하는 답을 모두 구해서 비교하는 방식으로 최적화 문제를 해결할 수 있음.

backtracking은 brute force와 유사하지만, 가능성이 없는 경우는 더 이상 확인하지 않음으로써 성능을 개선함.

backtracking은 기본적으로 깊이 우선 탐색으로 상태공간트리를 탐색함. 이때 depth-first search는 재귀를 사용하므로 backtracking의 코드에서는 재귀 호출을 사용함.

2. 상태공간트리
상태공간트리(State Space Tree)는 루트부터 특정 지점(주로 leaf)까지의 경로가 해답에 대한 후보로 존재하는 트리로, 깊이 우선 탐색을 통해 특정 경로가 해답인 것을 판단할 수 있음. 이때 어떤 경로가 해답이라면 해당 경로 위에 있는 노드들이 해답을 구성함.

문제와 조건에 따라서 leaf가 아닌 지점에서 정답이 나올 수도 있음.

트리인데 단순 수형도로도 생각할 수 있음.

backtracking에서의 성능 분석은 상태공간트리에서 방문하는 노드의 개수를 세서 수행할 수 있음.

3. 유망함
유망함(Promising)은 해당 경우가 조건에 위배되지 않음을 의미함. 해당 경우가 조건에 위배될 수 밖에 없다면 유망하지 않다고 함.

전체 경우를 확인하는 대신, 해당 경우가 유망하지 않다면 하위 경우는 전부 무시하고(pruning) 분기점으로 돌아감(backtrack). 즉, 유망하면 계속 가고, 유망하지 않으면 유망했던 지점까지 돌아감.

당연하게도 결과로 도출된 정답 또한 유망함.

4. backtracking의 동작 과정
backtracking의 동작 과정은 단순히 아래와 같음.

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

    // 최적화 문제인 경우
    if (value(v) is better than best) {
        best = value(v);
    }

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

물론 부모 노드에서 자식 노드로 이동한 뒤 검사하는 것이 아니라, 우선 검사하고 유망한 노드만 방문하도록 구현하는 것이 더 효율적일 수 있음. 여기에서는 이해의 편의를 위해 위의 방법대로 pseudo code를 정리함.

backtracking은 개념 자체가 간단하기 때문에 구현 방식에 집중하자.

5.1.2. Monte Carlo Algorithm

1. Monte Carlo

Monte Carlo Algorithm은 backtracking의 성능을 분석하는 알고리즘으로, 상태공간트리에서 존재할 수 있는 전형적인 경로를 무작위로 생성하여 그 경로 상에 존재하는 노드의 수를 세는 기법임. 무작위 경로에 대한 노드 방문 횟수를 반복적으로 구하고, 그 평균치를 추정치로 함. 즉, monte acrlo algorithm은 결정적(deterministic) 알고리즘이 아닌 확률적(probabilistic) 알고리즘임.

특히 backtracking에서는 상황에 따라 노드가 유망할 수도 있고, 그렇지 않을 수도 있음. 즉, 일반적인 복잡도 함수를 얻을 수 없으므로, 확률적으로 접근해야 함..

monte carlo를 적용하려면 상태공간트리가 아래의 두 가지 조건을 만족해야 함. 즉, 같은 level의 노드들이 모두 동일한 조건에 있어야 함.

1) 상태공간트리에서 같은 level의 노드들은 모두 동일한 유망함의 조건을 사용해야 함.
2) 상태공간트리에서 같은 level의 노드들은 모두 동일한 개수의 자식노드를 가져야 함.

2. 계산 과정

monte carlo에서는 level별로 유망한 자식 노드 개수의 평균치(추정치)(m_i)와 해당 level의 노드가 가지는 자식 노드의 수(t_i)를 활용하여 전체 노드의 개수를 셈. 특정 level(i+1)에 존재하는 노드의 수는, 해당 level 상위에서 존재하는 평균 노드의 수와 해당 level 바로 위 level이 가지는 자식 노드의 수를 모두 곱해 얻을 수 있음. 즉, $m_0 m_1 \cdots m_{i-1} t_i$로 구할 수 있음. 이를 모든 level에 대해서 계산하면 됨.

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

m_i를 구하는 과정은 간단하지 않으므로 여기에서는 다루지 않음.

참고로, m_i와 t_i 모두 자식 노드의 개수에 대한 값인 것 유의하자.

이를 구하는 pseudo code는 아래와 같음. 참고로, mprod는 $m_0 m_1 ...$이고, numnodes는 계산 결과임.

```
int estimate() {
    node v;
    int m, mprod, t, numnodes;
    v = root of state space tree;
    numnodes = 1; // 전체 노드의 개수
    m=1; // m_i
    mprod = 1; // m_0 m_1...

    while (m != 0) {
        t = number of children of v;
        mprod = mprod*m;
        numnodes = numnodes + mprod*t;
        m = number of promising children of v;
        if (m != 0) {
            v = randomly selected promising child of v;
        }
    }
    return numnodes;
}
```

5.2. n-Queens Problem

5.2.1. n-Queens Problem

n-Queens Problem은 n개의 Queen을 서로 상대방을 위협하지 않도록(같은 열/행이나 대각선상에 위치하지 않도록) $n \times n$ 체스판에 배치시키는 문제임.

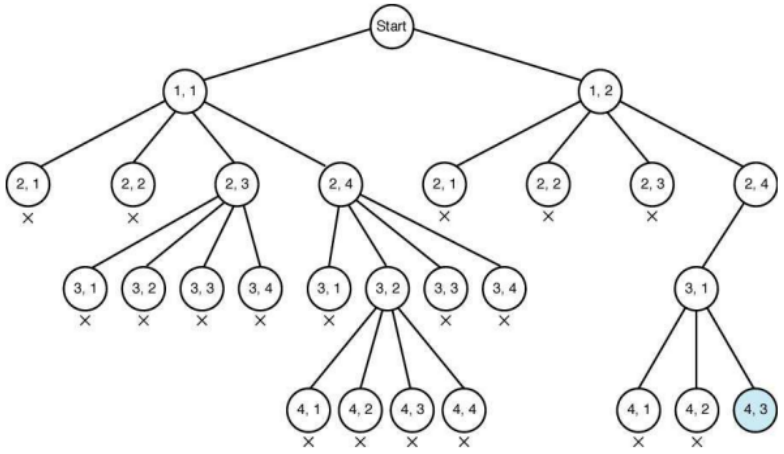
5.2.2. Brute Force

n개의 queen을 각 행별로 하나씩 배치한다고 생각하면, 행이 n개이므로 총 n^n가지를 점검해야 함. 즉, $O(n^n)$임.

5.2.3. Backtracking

1. 상태공간트리와 유망함의 조건

상태공간트리는 각 행 별로 하나의 노드를 구성하고, 그 값은 queen이 위치하는 좌표로 함. 이때 루트는 단순히 시작점으로, 좌표를 나타내지 않음. 즉, 아래와 같이 구성함.



유망함의 조건은, 해당 queen의 열(행은 이미 걸려졌음.)과 대각선에 다른 queen이 존재하는지로 함. 해당 노드까지의 경로에 존재하는 queen들에 대해서 모두 비교를 수행해야 함. 이때 두 queen이 서로 대각선(diagonal)에 위치하는지는, x좌표의 차이의 절댓값과 y좌표의 차이의 절댓값이 같은지를 비교하여 확인할 수 있음.

2. Pseudo Code

queen의 개수를 입력(n)으로 받음. queen()의 인자로는 해당 행에서의 열 좌표가 들어감. col[]은 열 정보를 저장하는 전역 배열임. col[a]==b라면 a번째 행에 놓인 말은 b번째 열에 있다는 것.

```
void queens (index i) {
    index j;
    if (promising (i)) {
        if (i == n) {
            cout << col[1] through col[n];
        }
        else {
            for (j=1; j<=n; j++) {
                col[i+1] = j;
                queens(i+1);
            }
        }
    }
}

bool promising (index i) {
    index k;
    bool switch;
    k = 1;
    switch = TRUE;

    while (k<i && switch) {
        if (col[i]==col[k] || abs(col[i]-col[k])==abs(i-k)) {
            switch = FALSE;
        }
        k++
    }
    return switch;
}
```

3. 성능 분석

시간에 대한 성능 분석을 해보자. 방문하는 노드의 개수를 세서 분석할 수 있음.

1) 수식적 계산

우선 최대 노드의 개수를 level 별로 세서 상한을 구해보자. depth가 i인 level의 노드의 개수는 n^i 개이고, 루트까지 포함하면 깊이는 $0 \sim n$임. 즉, 최대 노드의 개수는 $1 + n + n^2 + \cdots + n^n = \frac{n^{n+1}-1}{n-1}$ 임.

이번엔 열이 겹치는 경우(유망함 적용)를 고려한 최대 노드의 개수를 생각해 보자. 어떤 열에 queen이 존재한다면 다음 queen은 해당 열을 제외한 열에만 위치할 수 있으므로, 총 노드의 개수는 $1 + n + n * (n-1) + \cdots + n!$임.

두 방법 모두 실질적인 성능 지표로서 사용될 수 없음. 실제로는 대각선에 위치하는 경우 또한 유망함의 조건으로 사용되므로 방문 노드 수는 현저히 줄어들 수 있음. 수식적인 계산만으로는 적절한 성능 분석을 수행할 수 없고, 이에 따라 monte carlo를 활용함.

2) Monte Carlo

monte carlo를 활용하여 통계적으로 성능을 분석할 수 있음.

pseudo code는 아래와 같음. t_i는 n으로 정해져 있으므로 m_i(유망한 노드 개수)만 매번 구하면 됨. 또한 무작위 경로를 선택함. 즉, 아래의 코드로 무작위 경로에 대한 방문 노드의 개수를 얻을 수 있음.

```
    int estimate_n_queens (int n) {
        index i, j, col[1..n];
        int m, mprod, numnodes;
        set_of_index prom_children;
        i = 0; numnodes = 1; m=1; mprod = 1;

        while (m != 0 && i!=n) {
            mprod = mprod*m; // m_0 m_1 ...
            numnodes = numnodes + mprod*n; // 전체 노드의 개수

            i++; m = 0; prom_children = 공집합;

            for (j=1; j<=n; j++) {
                col[i] = j;
                if (promising(i)) {
                    m++; // m(유망한 노드 개수) 세기
                    prom_children = prom_children 합집합 {j};
                }
            }
            if (m != 0) {
                j = random selection from prom_children; // 무작위 경로(자식 노드) 선
택
                col[i] = j;
            }
        }
        return numnodes;
    }
```

5.3. Sum of Subset Problem

5.3.1. Sum of Subset Problem

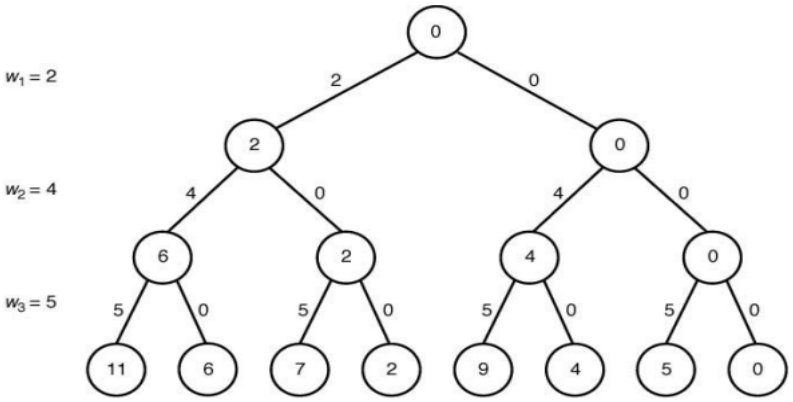
Sum of Subset Problem은 n개의 원소를 가지는 양의 정수 집합($w_1, , \cdots, w_n$)과 정수 W가 주어졌을 때, 원소들을 전부 더했을 때 W가 되는 부분집합을 찾는 문제임.

NP-complete임.

5.3.2. Backtracking

1. 상태공간트리와 유망함의 조건
상태공간트리는 각 원소의 사용 유무에 따라 경로가 나뉘는 이진 트리로 구성할 수 있음. 각 원소에 해당하는 level에서 왼쪽 자식 노드는 해당 원소가 사용되었음을, 오른쪽 자식 노드는 해당 원소가 사용되지 않았음을 의미함. 각 노드는 해당 선택까지의 합계를 값으로 가짐. 루트는 시작점으로, 값을 가지지 않음.

이때 각 원소는 비내림차순(또는 오름차순)으로 구성됨.



n=3, W=6 and $w_1 = 2$; $w_2 = 4$; $w_3 = 5$

유망함의 조건은, 단순히 해당 노드에서의 합이 W를 넘었는지임.

2. Pseudo Code

pseudo code는 아래와 같음. i는 level, weight는 지금까지의 총합, total은 앞으로 가능한 최댓값임.

```
void sum_of_subsets (index i, int weight, int total) {
    if (promising(i)) {
        if (weight == W) { // 정답 발견
            cout << include[1] through include[i];
        }
        else {
            // 다음 level 포함
            include[i+1] = "yes";
            sum_of_subsets(i+1, weight+w[i+1], total-w[i+1]);

            // 다음 level 포함x
            include[i+1] = "no";
            sum_of_subsets(i+1, weight, total-w[i+1]);
        }
    }
}

bool promising (index i) {
    return (weight+total >= W) && (weight == W || weight+w[i+1] <= W);
}
```

promising의 조건이 독특하게 정의되어 있으므로 유의하자.

3. 성능 분석

시간에 대한 성능 분석을 하자. 방문하는 노드의 개수로 분석함.

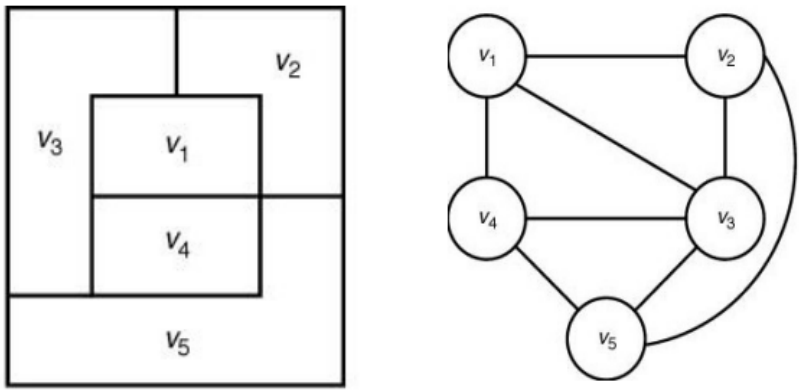
W가 w_n이라면 모든 노드를 탐색해야 하므로 최악의 경우임. 즉, 전체 방문 노드의 개수는 $1+2+2^2+\cdots+2^n = 2^{n+1}-1$이고, $\theta(2^n)$의 성능을 가짐.

5.4. Graph Coloring

5.4.1. Graph Coloring

Graph Coloring(m-coloring)은 m개의 색을 가지고, 인접한 노드가 같은 색을 가지지 않도록 그래프를 색칠하는 문제임.

어떤 지도에서 이웃한 영역이 같은 색을 가지지 않도록 색칠하는 것과 같음. 임의의 지도는 대응되는 평면 그래프(Planar Graph)로 바꾸어 접근할 수 있음.

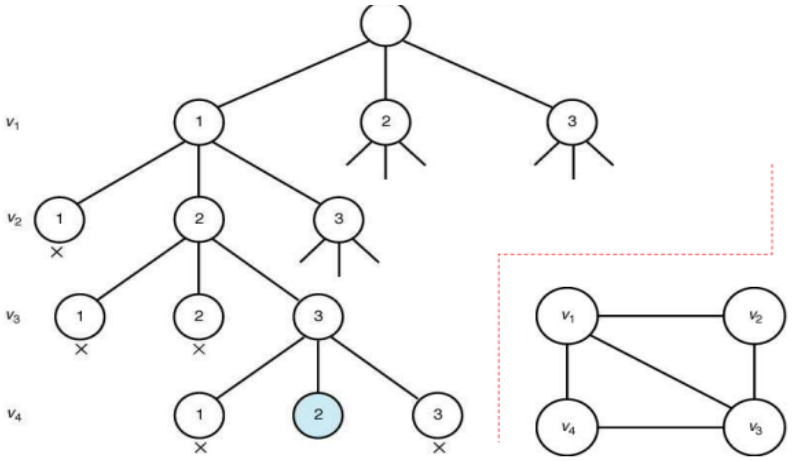


NP-complete임.

5.4.2. Backtracking

1. 상태공간트리와 유망함의 조건

상태공간트리는 각 정점 별로 level을 가지도록 하고, 해당 level에 존재하는 노드(트리의 노드)는 m 개이며 각각 해당 색의 사용을 나타냄. 즉, 아래와 같음.



유망함의 조건은 단순함. 해당 노드와 그 이전 노드 전부르 비교하여, 그래프 상에 연결이 존재하고 색이 같으면 유망하지 않은 것으로, 그렇지 않다면 유망한 것으로 함.

2. Pseudo Code

pseudo code는 아래와 같음. n은 정점의 개수, m은 색의 개수, 배열 W는 그래프의 인접 행렬임.

```
void m_coloring (index i) {
    int color;
    if (promising(i)) {
        if (i == n) {
            cout << vcolor[1] through vcolor[n];
        }
        else
        for (color = 1; color<=m; color++) {
            vcolor[i+1] = color;
            m_coloring(i+1);
        }
    }
}

bool promising(index i) {
    int j; bool switch;
    switch = TRUE;
    j = 1;

    while (j<i && switch) {
        if (W[i][j] && vcolor[i]==vcolor[j]) switch = FALSE;
        j++;
    }
    return switch;
}
```

3. 성능 분석
시간에 대한 성능 분석을 하자. 방문하는 노드의 개수로 분석함.

m이 2이고, v_n이 모든 다른 정점과 연결되어 있고, v_{n-1}과 v_{n-2}이 연결되어 있고, 다른 정점들은 연결되어 있지 않은 경우는 상태공간트리의 모든 노드를 방문하게 되는 최악의 경우임. 상태공간트리의 최대 노드의 개수는 $1 + m + m^2 + \cdots + m^n = \frac{m^{n+1}-1}{m-1}$ 임.

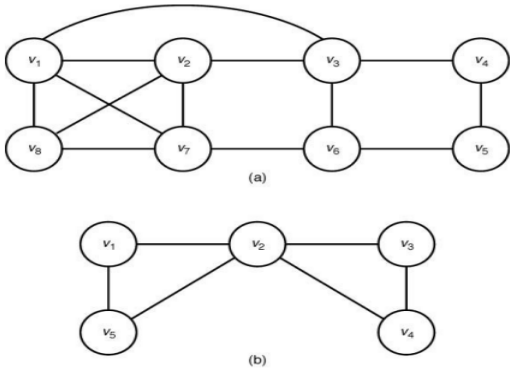
물론 monte carlo를 활용하여 성능을 분석할 수도 있음.

5.5. Hamiltonian Circuit Problem

5.5.1. Hamiltonian Circuit Problem

Hamiltonian Circuit Problem(HCP)은 그래프에서 hamiltonian circuit을 구하는 문제임.

hamiltonian circuit은 특정 정점에서 출발하여 그래프 내의 모든 정점을 한 번씩만 방문하고 출발한 정점으로 돌아오는 경로임. 즉, tour임. TSP는 optimal tour를 구했다면, HCP는 단순히 모든 tour를 찾음. 물론 또는 하나만 찾고 종료하도록 하거나, 모든 답을 비교해서 optimal한 결과를 도출하도록 할 수도 있음.



NP-complete임.

5.5.2. Backtracking

1. 상태공간트리와 유망함의 조건

상태공간트리는 간단히 구성할 수 있음. 각 level에서는 해당 순서에 선택하는 정점을 노드로 나타냄. 이 때 출발 정점(v_1)은 방문 시에 고려하지 않으므로 0번째 정점으로 하고, 나머지 n-1개의 정점을 $1 \sim n-1$ 번째로 함. 루트는 0번째 정점을 나타냄.

유망함의 조건으로는 아래의 3가지를 사용함.

1) i번째 정점은 i-1번째 정점과 이웃해야 함.
2) n-1번째 정점은 0번째 정점과 이웃해야 함.
3) i번째 정점은 앞쪽 $1 \sim i-1$ 번째 정점과 겹치면 안됨.

2. Pesudo Code

pesudo code는 아래와 같음. n은 정점의 수, W는 그래프의 인접 행렬임. i는 i번째 정점을 말하고, 배열 vindex에는 1번째부터 n-1번째의 정점을 저장함. vindex[0]=1임.

```
void hamiltonian(index i) {
    index j;
    if (promising(i)) {
        if (i == n-1) {
            cout << vindex[1] through vindex[n-1];
        }
        else {
            for (j = 2; j<=n; j++) { // Try all vertices as
                vindex[i+1] = j; // next one.
                hamiltonian (i+1);
            }
        }
    }
}

bool promising(index i) {
    int j; bool switch;
    if (i== n-1 && !W[vindex[n-1]][vindex[0]]) switch = FALSE; // 2번
    else if (i>0 && !W[vindex[i-1]][vindex[i]]) switch = FALSE; // 1번
    else {
        // 3번
        switch = TRUE;
        j= 1;
        while (j<i && switch) {
            if (vindex[i] == vindex[j]) switch = FALSE;
            j++;
        }
    }
    return switch;
}
```

3. 성능 분석

시간에 대한 성능 분석을 하자. 노드의 방문 횟수로 분석함.

각 노드가 n-1개의 자식 노드를 가질 수 있고 총 n개의 level이 있으므로, 상태공간트리가 가지는 최대 노드 개수는 $1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{n-1} = \frac{(n-1)^n - 1}{n-2}$ *임.*

물론 노드 전체를 방문하는 경우는 존재하지 않음. 최악의 상황을 가정해 보자. v_1*은* v_2*하고만 연결되어 있고,* v_1*을 제외한 나머지 정점들은 모두 연결되어 있다고 하자. 이 경우 n-1번째에 도달해서야 경로가 존재하지 않는 것을 판단할 수 있으므로, 이전에 방문한 것을 피해 방문하면 아래와 같은 노드 방문 횟수를 가지게 됨.*

$$1 + (n-1) + (n-1) + (n-2)(n-1) + (n-3)(n-2)(n-1) + \cdots + (n-1)(n-2)\cdots(n-(n-2))$$

즉, 최악의 경우 지수보다 더 안좋은 성능을 가짐.

6. Branch and Bound

6.1. Branch and Bound

6.1.1. Branch and Bound

분기한정법(Branch and Bound)는 상태공간트리를 기반으로 최적화 문제를 푸는 접근 방식임. 이는 backtracking과 유사하지만, 유망함을 판단할 때 bound라는 개념을 사용하고, 상태공간트리를 탐색하는 방법이 제한되어 있지 않고, 최적화 문제에만 사용됨.

bound는 끝까지 탐색을 하지 않고 현재의 상태로도 계산할 수 있는 한계치로, 해당 위치로부터 더 탐색했을 때 나올 수 있는 최적의 상황에 대한 값임. 해당 경로로 끝까지 갔을 때 실제로 확인한 최적의 값은 bound보다 더 최적일 수 없음. 만약 실제로 탐색 중에 발견된 값보다 특정 노드의 bound 값이 더 좋지 않다면, 해당 노드는 유망하지 않은 것으로 판단함.

branch and bound에서는 깊이 우선 탐색 뿐만 아니라, 너비 우선 탐색 등 여러 탐색 기법을 사용함. 그 중 뒤에서 설명할 최고 우선 탐색을 주로 사용함. 참고로, 너비우선탐색의 경우 주로 queue와 반복문으로 구현함.

6.2. Knapsack Problem

6.2.1. Knapsack Problem

Knapsack Problem은 가치와 무게에 대한 값이 존재하는 물건들을 담을 수 있는 최대 무게가 정의된 배낭에 넣을 때, 가치가 최대가 되도록 담는 방법을 찾는 최적화 문제임.

knapsack problem에는 두 가지 종류가 있음.

1) 0-1 knapsack problem : 각 물건을 선택하거나, 선택하지 않는 선택지만 존재하는 knapsack problem.

2) fractional kanpsack problem : 각 물건의 일부분만 쪼개서 선택할 수 있는 knapsack problem.

knapsack problem은 NP-complete임.

6.2.2. 0-1 Knapsack Problem : Brute Force

물건이 n개라면 그 부분집합의 개수는 2^n개이므로 성능은 $\Theta(2^n)$임.

6.2.3. Knapsack Problem : Greedy

1. 0-1 Kanpsack Problem
0-1 knapsack problem을 greedy로 시도한다면 최대 가치 또는 최대 무게 당 가치를 갖는 물건부터 채우는 것을 greedy criterion으로 할 수 있는데, 이 경우 항상 최적임이 보장되지는 않음(쉽게 반례를 찾을 수 있음.). 즉, 0-1 knapsack problem에는 greedy를 적용할 수 없음.

2. Fractional Knapsack Problem
최대 무게 당 가치를 갖는 물건부터 채우는 것을 greedy criterion으로 해서 풀 수 있음. 당연하게도 마지막 물건은 필요하다면 잘라서 넣으면 됨. 이 경우 항상 최적임.

6.2.4. 0-1 Knapsack Problem : DP

1. 점화식
n개의 물건(1번째 ~ n번째)이 존재하고, 각 가치를 p_i, 무게를 w_i라 하자. i번째까지 담은 무게가 w를 넘지 않을 때의 최대 이익을 P[i][w]라고 하면, 그 점화식은 아래와 같음. 즉, i번째 물건의 무게가 w보다 크다면 넣지 않고, w보다 작다면 넣지 않는 것과 넣은 것 중 더 큰 가치를 가지는 것을 선택함.

$$P[i][w] = \begin{cases} max(P[i-1][w], p_i + P[i-1][w-w_i]) \ (if \ w_i \le w) \\ P[i-1][w] \ (if \ w_i > w) \end{cases}$$

P[0][w]=P[i][0]=0임.

최대 무게가 W라면, 위의 점화식을 활용하여 P[n][W]를 구해야 함. i번째까지를 사용한 값을 구하려면 i-1번째까지 사용한 값이 필요하므로, 사용하는 물건의 개수를 하나씩 늘려가며 계산하면 됨. 이때 각

개수에 대해서는, w의 값을 0부터 W까지 늘려가며 계산하면 됨.

		Capacity remaining g=0	g=1	g=2	g=3	g=4	g=5	g=6	g=7	g=8
k=0	f(0,g) =	0	0	0	0	0	0	0	0	0
k=1	f(1,g) =	0	**15**	15	15	15	15	15	15	15
k=2	f(2,g) =	0	**15**	15	15	15	15	25	25	25
k=3	f(3,g) =	0	15	15	15	**24**	24	25	25	25
k=4	f(4, g) =	0	15	15	15	24	24	25	25	**29**

2. 성능 분석 및 개선

1) bottom-up

배열 P를 전부 구해야 하고, n과 W는 상관관계가 없으므로, 각 항에 대한 계산을 기본연산으로 한다면 모든 경우에 대해 $\Theta(nW)$임.

2) top-down으로 개선

bottom-up으로 P[n][W]를 구하려면 모든 원소를 전부 계산해야 함. 반면 P[i][w]를 구하는 데에는 P[i-1][w]와 P[i-1][w-w_i]의 값만이 필요하므로 top-down으로 계산할 수도 있음. 즉, 이를 분할정복으로 계산할 수 있음.

분할정복을 사용한다면, 각 n값에 대해 2번의 분할이 존재하므로 총 $1+2+2^2+\cdots+2^{n-1}=2^{n-1}-1$번의 연산이 발생하고, $\Theta(2^n)$의 성능을 가짐.

3) 최종 성능

최악의 경우에는 top-down으로 계산한 것도 $O(nW)$의 성능을 가짐. 즉, 개선한 DP의 최종 성능은 최악의 경우 $O(min(2^n, nW))$임.

6.2.5. 0-1 Knapsack Problem : Backtracking

0-1 knapsack problem을 backtracking으로 풀어봄. 즉, 매번 정답이 나올 때마가 기록해 두고, 모든 정답을 확인하며 최적의 답을 찾음.

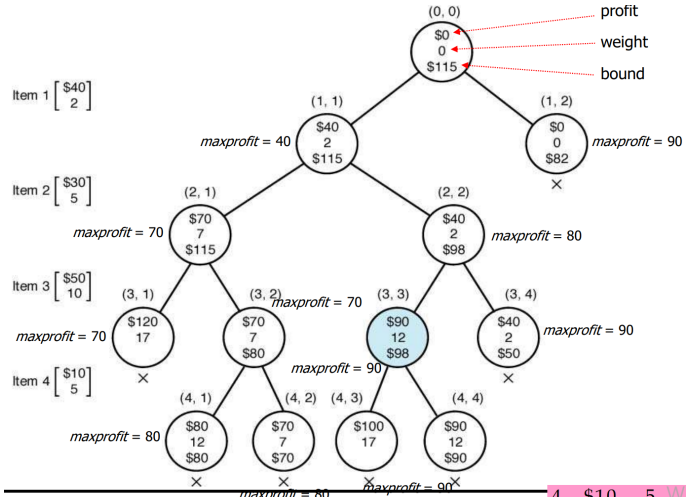
물건의 개수는 n, 담을 수 있는 최대 무게를 W, 각 가치와 무게를 $p_1 \sim p_n$, $w_1 \sim s_n$으로 함.

사실 bound를 사용하므로 branch and bound이고, depth-first로 생각할 수 있음.

1. 상태공간트리

상태공간트리에서 각 level은 특정 물건에 대한 선택을 나타내고, 해당 물건을 선택했으면 왼쪽 자식으로, 선택하지 않았으면 오른쪽 자식으로 가는 것으로 함. 루트는 아무것도 선택하지 않은 상태임.

이때 상태공간트리를 구성하는 물건의 순서는 무게 당 가치 $\frac{p_i}{w_i}$ 값을 기준으로 비오름차순(내림차순)으로 구성함.



2. 유망함의 조건

유망함을 따지기 위해 profit, weight, bound, maxprofit 변수를 사용함. profit과 weight은 해당 노드의 상태까지의 가치의 합과 무게의 합임. bound는 fractional knapsack problem임을 가정했을 때 현재 위치에서 계산할 수 있는 최대 가치임. maxprofit은 현재까지 나온 profit 중 최댓값(최적)임.

현재 노드가 i이고, 해당 경로에서 계속 진행하면 k일 때 처음으로 $weight > W$가 된다고 하자. bound는 아래와 같이 계산할 수 있음. 즉, k 이전까지는 단순히 가치를 전부 더하고, k에서는 fractional이라고 가정하여 더함. 실제로는 fracional이 아니므로 결과로 도출된 정답은 bound보다 클 수 없음.

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

$weight < W$이고, $bound > maxprofit$이면 해당 노드는 유망함. 즉, 무게가 초과하지 않았고, maxprofit 보다 최적일 가능성이 존재함.

3. Pseudo Code

n, W, $p_1 \sim p_n$, $w_1 \sim s_n$를 입력으로 받음.

```
void knapsack (index i, int profit, int weight) {
    if (weight <= W && profit > maxprofit) { // best so far
        maxprofit = profit;
        numbest = i;
        bestset = include;
    }
    if (promising(i)) {
        include[i+1] = "YES"; // Include w[i+1]
        knapsack(i+1, profit+p[i+1], weight+w[i+1]);
        include[i+1] = "NO"; // Not include w[i+1]
        knapsack(i+1, profit, weight);
    }
}

bool promising(index i) {
    index j, k;
    int totweight;
    float bound;

    if (weight >= W) {
        return FALSE;
    }
    else {
        j = i+1;
        bound = profit;
        totweight = weight;
        while ((j <= n) && (totweight +w[j] <= W)) {
            totweight = totweight + w[j];
            bound = bound + p[j];
            j++
        }

        k=j;
        if (k <= n) {
            bound = bound +(W-totweight)*p[k]/w[k];
        }

        return bound > maxprofit;
    }
}
```

매 노드에서의 물건 구성은 include 배열에 저장하고, 최적인 경우의 include 값은 bestset에 따로 저장함.

어떤 노드가 유망하면, 그 자식 노드들을 전부 확인하게 됨.

4. 성능 분석

상태공간트리는 $2^{n+1}-1$개의 노드를 가짐(등비수열의 합).

물건이 n개이고 W가 n인 $p_i = i, w_i = i$인 경우를 생각해 보면, 최적의 답은 n번째 물건 하나만을 선택하는 것이므로 상태공간트리의 가장 오른쪽까지 갔다가 마지막에 왼쪽 노드를 선택하게 됨.

즉, 최악의 경우 $2^{n+1}-2$개의 모든 노드를 탐색함. 시간의 관점에서 각 노드의 방문을 기본연산으로 했을 때, worst case에 대한 order는 $\Theta(2^n)$임.

DP는 $O(min(2^n, nW))$이고 backtracking은 $\Theta(2^n)$이므로 order만 보고서는 어느 알고리즘이 더 낫다고 이야기하기 어려움. 하지만 monte carlo를 사용해서 확인하면 backtracking이 DP보다 일반적으로는 더 좋은 성능을 보장한다고 함.

6.2.6. 0-1 Knapsack Problem : Branch and Bound(breadth-first)

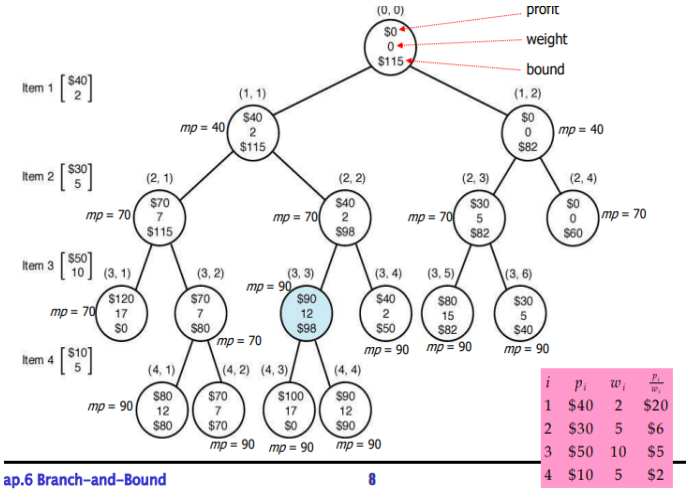
1. Bread-first

bread-first 탐색을 사용하여 branch and bound를 적용함. 이를 위해 queue를 사용하고, queue에 데이터를 넣기 위한 구조체를 사용함. 즉, 아래와 같은 구조를 사용함.

```
void breadth_first_branch_and_bound(state_space_tree T, number& best){
    queue_of_node Q;
    node u, v;
    initialize(Q); // Initialize Q to be empty
    v = root of T; // Visit root
    enqueue(Q,v);
    best = value(v);

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

상태공간트리는 앞에서 다룬 backtracking의 방식과 동일함. 루트는 시작점으로 함.



2. 성능 분석

단순히 노드 방문 횟수를 세어 보면, 깊이우선탐색보다 노드 방문이 더 많은 상황이 존재함. 이는 탐색 초반에 max profit으로 큰 값이 지정되면 덜 방문하는데 너비우선탐색에서는 뒤쪽에 결정되는 상황이기 때문임. 즉, 탐색 시에 큰 max profit 값을 빠르게 찾을 수 있어야 하는데, 이를 반영한 것이 best-first 임.

6.2.7. 0-1 Knapsack Problem : Branch and Bound(best-first)

1. Best-first

Best-first 탐색에서는 최적의 bound값을 가진 노드를 우선적으로 탐색하기 위해서 단순 queue 대신 priority queue(heap)를 사용하여 구현함.

즉, 아래와 같이 단순히 queue를 priority queue로 바꿔주기만 하면 성능이 개선됨.

```
void best_first_branch_and_bound (state_space_tree T,number best) {
    priority_queue_of_node PQ;
    node u,v;
    initialize(PQ); // initialize PQ to empty
    v = root of T;
    best = value(v);
    insert(PQ,v);

    while(!empty(PQ)) { // Remove node with best bound
        remove(PQ,v);
        if(bound(v) is better than best) { // Check if node is still
            // promising
            for(each child u of v) {
            if(value(u) is better than best)
                best = value(u);
                if(bound(u) is better than best) // promising 검사
                    insert(PQ,u);
            }
        }
    }
}
```

이때 queue에서 꺼냈을 때 유망함을 다시 검사함. bound값은 항상 동일한데, best 값은 바뀔 수 있기 때문임.

6.3. Traveling Salesperson Problem

6.3.1. Traveling Salesperson Problem

Traveling Salesperson Problem(TSP)는 그래프에서 optimal tour를 구하는 문제임.

tour는 모든 노드를 방문하고 원래의 노드로 돌아오는 path(cycle)을 말함. 즉, TSP는 모든 노드를 방문하고 원래 노드로 돌아오는 최적의 tour를 구하는 문제임.

여기에서는 최단경로의 값과 실제 최단경로를 구하는 알고리즘을 살펴봄.

TSP는 NP-complete임. 즉, TSP에 대해 지수보다 성능을 개선한 알고리즘이 없고, 그런 알고리즘이 존재할 수 없다는 것이 증명되지도 않음.

6.3.2. Brute Force

모든 노드가 서로 edge로 연결된 n개의 노드가 존재하는 그래프를 생각하자. 모든 노드가 연결되어 있으므로 첫 번째 노드를 제외한 나머지 노드를 단순히 나열하기만 하면 됨. 즉, 이 그래프에 존재하는 tour의 수는 $(n-1)!$임. DP에서의 최적화 문제는 기본적으로 각 경우를 전부 비교하여 최적의 값을 찾으므로, brute force의 경우 성능은 $O((n-1)!)$임.

6.3.3. DP

1. 점화식
n개의 노드가 있다고 하자. 출발지/도착지는 첫번째 노드로 함. 모든 노드에 대한 집합을 V라 함. 그래프는 인접 행렬 W로 나타내고, 경로가 존재하지 않는 원소에는 무한대(충분히 큰 수)가 값으로 사용됨. 그래프는 방향성이든 비방향성이든 상관 없음.

D[i][A]는 i번째 노드부터 집합 A에 해당하는 노드들을 전부 한 번씩 거쳐 첫번째 노드로 가는 최단경로

(값)임.

이때 D[i][공집합]=W[i][1]임.

결과적으로 구해야 하는 값은 아래와 같음. 첫번째 노드부터, j번째와 첫번째를 제외한 노드를 거쳐 다시 첫번째 노드로 돌아오는 값 중 최솟값을 구함.

$$D[1][V-v1] = min_{2 \le jleqn}(W[1][j] + D[j][V-v1, vj]);$$

이를 구하기 위한 점화식은 아래와 같음. i번째 노드부터 출발하여 A에 속하는 노드들을 거쳐 첫번째 노드로 돌아오는 값은, i에서 j번째로 가는 가중치와 j번째에서 j번째를 제외한 A의 노드들을 거쳐 첫번째로 가는 값을 더한 것 중 최솟값을 구함. 즉, i번째 바로 다음이 j번째인 경우를 비교하는 것.

$$D[i][A] = min_{2 \le j \le n}(W[i][j] + D[j][A - \{v_j\}])$$

P[i][A]는 i번째 노드부터 집합 A에 해당하는 노드들을 전부 한 번씩 거쳐 첫번째 노드로 가는 최단경로에서의 i번째 바로 다음으로 오는 노드임. D가 결정될 때 해당 노드의 값을 동일한 인덱스의 P에 저장하면 됨. 최단경로 자체는 P[i][C]=a, $P[a][C - \{v_a\}] = b$ 이런 식으로 C에 해당되는 집합이 공집합이 될 때까지 반복해서 구할 수 있음.

2. Pseudo Code

인접 행렬 W를 입력으로 받음.

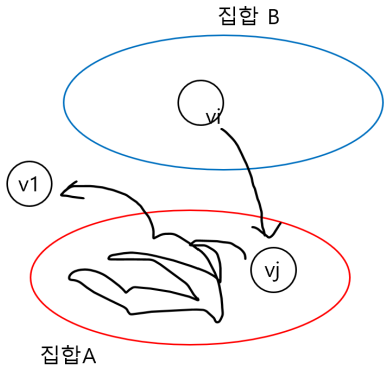
```
void travel(int n, const number W[], index P[][], number& minlength){
    index i, j, k;
    number D[1..n][subset of V-{v1}];
    for(i=2; i<=n; i++) {
        D[i][공집합] = W[i][1];
    }

    // A의 크기 지정(A 크기 고정)
    for(k=1; k<= n-2; k++) {
        // 해당 크기의 각 부분집합(A 집합 고정)
        for(all subsets A in V-{v1} containing k vertices) {
            // B의 원소를 하나씩 확인(B 원소 고정)
            for(i such that i != 1 and vi is not in A){
                // A의 원소 중 최솟값을 가지는 것 선택
                D[i][A] = min_{j : vj in A}(W[i][j]+D[j][A-{vj}]);
                P[i][A] = value of j that gave the minimum;
            }
        }
    }

    D[1][V-{v1}] = min_{j : vj in A}(W[1][j]+D[j][V-{v1,vj}]);
    P[1][V-{v1}] = value of j that gave the minimum;

    minlength = D[1][V-{v1}];
}
```

본 알고리즘에서는 총 3가지 집합으로 노드를 나눌 수 있음. 첫번째 노드와, A와, 이 둘을 제외한 나머지임(이하 B). 위의 코드는 A의 크기를 1부터 n-2까지 지정함(n-1은 동작이 다름.). 해당 크기의 A와, 그때 존재하는 B의 각 원소에 대해 D와 P를 구함. min에서는 고정된 A 집합과 B의 원소 하나에 대해, B 바로 다음으로는 A 중 어떤 원소가 와야 최단경로인지를 최솟값으로 정함.



정리하면, 공집합에서 시작해 A의 크기를 하나씩 늘려가며 각 크기에서의 최적의 값을 찾음. k 루프에서는 원소의 개수가 k-1개인 D[i][A-vj]로 원소의 개수가 k개인 D[i][A]를 채움.

마지막에 D와 P를 구할 때는 A의 원소가 n-1개이고, B는 공집합이므로 해당 루프에서는 처리하지 못하고, 따로 계산함. 이는 마지막 연산과 점화식이 분리되어 있는 것과 같은 이유임.

3. 성능 분석
1) 시간복잡도 분석
시간에 대한 복잡도를 분석하자. every case에 대해 구할 수 있음.

D와 P를 계산하는 부분을 기본연산으로 함. k = n-2, A = $\binom{n-1}{k}$, i = n-1-k, j = k이므로 정리하면 아래와 같음.

$$T(n) = \sum_{k=1}^{n-2} \binom{n-1}{k} (n-1-k)k$$

아래의 정리를 사용하면 계산이 간단함. 모든 $1 \leq n$에 대해서 성립함.

$$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$$

정리하면 아래와 같음.

$$T(n) = \sum_{k=1}^{n-2} \binom{n-1}{k} (n-1-k)k = \sum_{k=1}^{n-2} \binom{n-2}{k} (n-1)k = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$

2) 공간복잡도 분석
dominant factor는 D와 P임. 이 둘의 크기는 같음.

n개의 노드가 있다고 하면, $V-\{v_1\}$의 부분집합은 총 2^{n-1}개임. 즉, 공간복잡도는 $2n2^{n-1} \in \Theta(n2^n)$임.

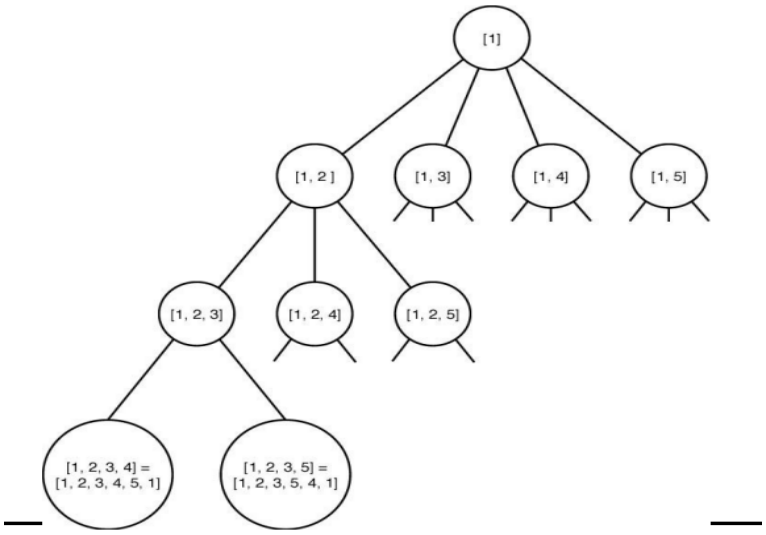
즉, DP를 사용하면 시간복잡도가 $\Theta(n^2 2^n)$이고 공간복잡도는 $\Theta(n2^n)$임. 성능이 안 좋지만, brute force 보다는 빠름.

인덱스 안의 집합 A는 각 원소의 유무를 1과 0으로 나타내서 비트 연산 등으로 구현할 수 있을 것 같음.

6.3.4. Branch and Bound

best-first 탐색을 사용한 branch and bound로 TSP를 해결해 보자.

1. 상태공간트리
상태공간트리에서 각 노드는 방문하는 정점을 나타내고, 해당 정점까지의 경로를 값으로 가짐. 루트는 첫 번째 정점(v_1)을 나타냄. 이때 leaf 노드에서 도달하면 경로가 확정되므로 bound 등을 계산하지 않음.



2. bound

각 노드에서 bound는 모든 정점을 고려하여 계산함. 이때 정점은 이미 지나온 노드, 현재 방문 중인 노드, 아직 방문하지 않은 노드에 따라 값이 다르게 계산됨. 쉽게 말하면, 해당 정점에서 나갈 수 있는 간선 중에서 최솟값을 선택하는 것임.

1) 이미 지나온 노드 : 단순히 해당 경로에서의 가중치를 값으로 함.

2) 현재 방문 중인 노드 : 이미 지나온 노드로 가는 경로와, v_1으로 가는 경로를 제외한 것들 중 최솟값을 값으로 함. leaf에서는 bound를 계산하지 않으므로, v_1으로 가는 경로를 고려하는 경우는 존재하지 않음.

3) 아직 방문하지 않은 노드 : 이미 지나온 노드로 가는 경로와, 현재 방문 중인 노드로 가는 경로를 제외한 것들 중 최솟값을 값으로 함.

이렇게 정점별로 계산된 값을 전부 더하면 bound임.

3. Pseudo Code

pesudo code는 아래와 같음. 마지막 정점과 도착지는 자동으로 결정되므로 level이 n-2이면 leaf임. min-length는 leaf에서만 갱신됨.

```
void travel2(int n, const number W[][], ordered_set &opttour,
number &minlength) {
    priority_queue_of_node PQ;
    node u, v;
    initialize(PQ);
    v.level =0;
    v.path = [1];
    v.bound = bound(v);
    minlength = INFINITE;
    insert(PQ, v);

    while (!empty(PQ)) {
        remove(PQ, v);
        if (v.bound < minlength) {
            u.level = v.level + 1;
            for ((all i such that 2< i< n) && (i is not in v.path)) {
                u.path = v.path;
                put i at the end of u.path;
                if (u.level == n-2) {
                    put index of only vertex
                    not in u.path at the end of u.path;
                    put 1 at the end of u.path;
                    if (length(u)<minlength) {
                        minlength = length(u);
                        opttour = u.path;
                    }
                }
                else {
                u.bound = bound(u);
                if (u.bound < minlength) insert(PQ, u);
                }
            }
        }
    }
}
```

4. 성능 분석
이 방식은 성능을 개선하지만, 여전히 지수보다 개선하지는 못함.

참고로, 성능을 더 개선하기 위해 근사 알고리즘 등을 사용하기도 함. 근사 알고리즘은 최적에 대한 보장은 못하지만, 거의 최적에 가까운 해답을 도출하는 알고리즘임.

7. 기타 알고리즘들

7.1. 순차 탐색

7.1.1. 순차 탐색

1. 순차 탐색
순차 탐색은 배열의 요소 전부를 한쪽에서부터 끝까지 하나씩 확인하여 탐색하는 방식임.

pseudo code는 아래와 같음.

```
void seqsearch(int n, // 입력
        const keytype S[],
        keytype x,
        index& location) { // 출력

    location = 1;
    while (location <= n && S[location] != x) location++;
    if (location > n) location = 0;
}
```

2. 성능 분석
시간에 대한 복잡도를 구해보자. 비교 연산을 기본 연산으로 한다면 최악의 경우 $\Theta(n)$, 최선의 경우 $B(n) = 1$임.

만약 각 찾으려는 대상이 배열에 존재하고(존재하지 않는다면 존재하지 않는 경우에 대한 확률도 고려해야 함.) 각 위치에 존재할 확률이 전부 같다면(이 가정은 필수적임.), 평균적 경우는 $A(n) = \frac{n+1}{2}$ 임.

7.2. 피보나치 수 구하기

7.2.1. 분할정복

1. 분할정복
피보나치 수를 분할정복으로 구할 수 있음. 참고로 0번째 첫번째 수는 1임.

pseudo code는 아래와 같음.

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

2. 성능 분석
시간에 대한 복잡도를 구해보자. 함수 호출을 기본 연산으로 함.

복잡도는 every case에 대해 $T(n) = T(n-1)+T(n-2)+1$임. 아래의 수식으로 order를 구할 수 있음.

T(0), T(1). n이 짝수라고 가정함. $T(n) = T(n-1) + T(n-2) > 2T(n-2) > \cdots > 2^{\frac{n}{2}}T(0) = 2^{\frac{n}{2}}$. 즉, T(n)은 $2^{\frac{n}{2}}$보다 성능이 떨어짐. 이 정도만 알아도 DP로 우회하게 되므로 더 정리할 필요가 없음.

3. 검증
복잡도의 성능을 귀납적으로 구했으므로, 검증해야 함. 수학적 귀납법을 사용해 연역적으로 검증할 수 있음. 임의의 $n \geq 2$에 대해서 $T(n) > 2^{\frac{n}{2}}$ 임을 보이면 됨.

induction base : 대입해 보면 T(2), T(3)에서 성립함.

induction hypothesis : $2 \leq m < n$인 모든 m에 대해서 $T(m) > 2^{\frac{m}{2}}$가 성립한다고 가정함.

induction step : $T(n) = T(n-1) + T(n-2) > 2^{\frac{n-1}{2}} + 2^{\frac{n}{2}} + 1 > \cdots > 2^{\frac{n}{2}}$로 정리할 수 있음. 즉, 모든 n에 대해 성립함.

7.2.2. DP

1. DP
피보나치 수를 DP로 구할 수 있음.

pseudo code는 아래와 같음.

```
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];
}
```

2. 성능 분석
시간에 대한 복잡도를 구해 보자. 대입 연산을 기본 연산으로 한다면 복잡도는 $T(n) = n + 1$임.

분할정복에 비해 성능이 월등히 향상되었음.

7.3. 정렬

7.3.1. 교환 정렬

1. 교환 정렬
교환 정렬(Exchange sort)은 전체 키에 대해서, 특정 위치의 키와 해당 키 뒷쪽의 모든 키를 비교하여 순서를 바꾸는 정렬 방법임.

비내림차순(값이 같거나 오름차순인 경우)으로 정렬하는 pseudo code는 아래와 같음.

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

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

2. 성능 분석
시간에 대한 복잡도를 구해 보자.

비교 연산을 기본 연산으로 한다면 every case에 대해 생각할 수 있고, 복잡도 함수는 $T(n) = \frac{n(n-1)}{2}$ 임.

교환 연산을 기본 연산으로 한다면 최선/최악/평균에 대해 생각할 수 있음. 최악의 경우 복잡도 함수는 $W(n) = \frac{n(n-1)}{2}$ 임.

기본 연산을 비교로 하는 것은 간단하다. 이러면 every case이다. 근데 이렇게 끝내버리면 안된다. 기본 연산을 교환으로 해보자. worst, best, average case가 나오게 된다.

즉, 알고리즘의 성능을 이야기할 때는 어떤 기본 연산을 기준으로 했는지를 항상 고려해야 한다. 시/공간, 기본 연산, case를 모두 이야기해야 한다.