Algorithms for Physics

DDM

D. C. Kim

May 12, 2023

Table of Contents

Online Interpreter &Reference

1 Introduction

- Visual Studio Code, PyCharm, Spyder과 같은 개발 환경이 있다면 좋겠지만, 당장 필요하다면 온라인 인터프리터를 사용할 수 있다.
- OnlineGDB - Python을 추천한다.
- 파이썬 참고 자료로는 점프 투 파이썬을 추천한다.

What is Algorithm?

- 문제를 해결하기 위해 정해진 일련의 절차
  - 정렬 알고리즘
  - 탐색 알고리즘
  - 그래프 알고리즘
  - 그 밖의 각종 알고리즘(소수 판정, 기하학 등)
- 시간, 공간적 자원의 한계가 있기 때문에 문제를 더 효율적으로 해결하기 위해 연구가 필요하다.

Asymptotic notation

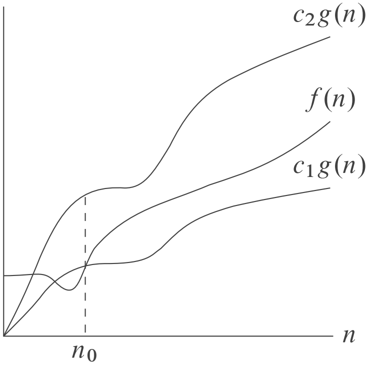
1 Introduction

- 어떤 함수의 증가 양상을 다른 함수와의 비교로 표현하는 방법이다.
- 다음과 같은 다섯 가지 표기법이 주로 사용되며, $\mathcal{O}$가 이 중에서도 많이 쓰인다.
  - $\mathcal{O}(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \ s.t. \forall n \geq n_0, f(n) \leq cg(n)\}$, 쉽게 말해 $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$.
  - $\Omega(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \ s.t. \forall n \geq n_0, cg(n) \leq f(n)\}$, 쉽게 말해 $\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0$.
  - $\Theta(g(n)) = \mathcal{O}(g(n)) \cap (g(n))$, 쉽게 말해 $0 < \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$.
  - $o(g(n)) = \left\{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\right\}$.
  - $\omega(g(n)) = \left\{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\right\}$.
- 예를 들어 $5n^2 = \mathcal{O}(n^2)$, $5n^2 + 3 = \mathcal{O}(n^2)$, $\frac{n^2}{2} - 5 = \mathcal{O}(n^2)$, $5n + 3 = \mathcal{O}(n^2)$이다.

Asymptotic notation

다음 그래프에서 $n \geq n_0$ 에 대해 $c_1 g(n) \leq f(n) \leq c_2 g(n)$ 이므로

- $f(n) = \mathcal{O}(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$.



Example 1

다음 10개의 정수들로 이루어진 수열에서, 연속된 몇 개의 수를 선택해서 구할 수 있는 합 중 가장 큰 합은?

$$2, 1, -4, 3, 4, -4, 6, 5, -5, 1$$

Example 1 - sol1: brute force

- 가능한 모든 연속된 수들을 더하여 답을 찾는다.
- 즉 $1 \le i \le 10$, $i \le j \le 10$에 대해
$$\sum_{k=i}^{j} a_k$$
의 값을 모두 구한 후 그 중 가장 큰 값($= 14$)이 답이다.
- 10개의 수들에 대해 $1 + 2 + \cdots + 9 + 10 = 55$번의 연산이 필요하다.
- n개의 수들이라면 약 n^2번의 연산이 필요할 것이다. $\to \mathcal{O}(n^2)$.

Example 1 - sol2: dynamic programming(dp)

- A_i 를 i 번째 수로 끝나는 가장 큰 연속합이라고 하자.
- 그 다음 A_{i+1} 의 값은 다음과 같이 두 경우로 나누어 결정할 수 있다.
  1. a_{i+1} 을 직전의 연속합에 포함시키는 경우, $A_{i+1} = A_i + a_{i+1}$ 이다.
  2. a_{i+1} 로 시작하는 새로운 연속합을 만들 경우, $A_{i+1} = a_{i+1}$ 이다.
- 가장 큰 연속합을 찾기 위해서 둘 중 더 큰 값을 선택하면 된다. 즉

$$A_{i+1} = \max(A_i + a_{i+1}, a_{i+1})$$

- n 개의 수들을 한 번씩만 보면 되므로 시간 복잡도는 $\mathcal{O}(n)$ 이다.

Example 1 - sol2: dynamic programming(dp)

파이썬으로 구현해보자.

```python
a = [2, 1, -4, 3, 4, -4, 6, 5, -5, 1]  # 기존 수열
A = [a[0]]  # i번째 수로 끝나는 최대 연속합

for i in range(1, len(a)):
A.append(max(A[i - 1] + a[i], a[i]))  # 수를 기존의 연속합에 이어 붙이거나 새로운 연속합을 만드는 경우 중 최대

print(A, max(A))  # A와 A의 최댓값
```

Example 2

다음 행렬 A에 대해, A^{16}은?

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad A^{16} = ?$$

Example 2 - sol1

단순히 곱하면 된다. 15번의 행렬곱이 필요하다.

$$A^1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},\ A^2 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix},\ A^3 = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix},\ A^4 = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix},\ A^5 = \begin{pmatrix} 3 & 5 \\ 5 & 8 \end{pmatrix}$$

$$A^6 = \begin{pmatrix} 5 & 8 \\ 8 & 13 \end{pmatrix},\ A^7 = \begin{pmatrix} 8 & 13 \\ 13 & 21 \end{pmatrix},\ A^8 = \begin{pmatrix} 13 & 21 \\ 21 & 34 \end{pmatrix},\ A^9 = \begin{pmatrix} 21 & 34 \\ 34 & 55 \end{pmatrix},\ A^{10} = \begin{pmatrix} 34 & 55 \\ 55 & 89 \end{pmatrix},$$

$$A^{11} = \begin{pmatrix} 55 & 89 \\ 89 & 144 \end{pmatrix},\ A^{12} = \begin{pmatrix} 89 & 144 \\ 144 & 233 \end{pmatrix},\ A^{13} = \begin{pmatrix} 144 & 233 \\ 233 & 377 \end{pmatrix},\ A^{14} = \begin{pmatrix} 233 & 377 \\ 377 & 610 \end{pmatrix}$$

$$A^{15} = \begin{pmatrix} 377 & 610 \\ 610 & 987 \end{pmatrix},\ A^{16} = \begin{pmatrix} 610 & 987 \\ 987 & 1597 \end{pmatrix}$$

Example 2 - sol2: Divide And Conquer

$A^{16} = (A^8)^2 = ((A^4)^2)^2 = (((A^2)^2)^2)^2$ 이므로 4번의 행렬곱만으로도 충분하다.

$$A^1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},\ A^2 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix},\ \cancel{A^3 = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}},\ A^4 = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix},\ \cancel{A^5 = \begin{pmatrix} 3 & 5 \\ 5 & 8 \end{pmatrix}}$$

$$\cancel{A^6 = \begin{pmatrix} 5 & 8 \\ 8 & 13 \end{pmatrix}},\ \cancel{A^7 = \begin{pmatrix} 8 & 13 \\ 13 & 21 \end{pmatrix}},\ A^8 = \begin{pmatrix} 13 & 21 \\ 21 & 34 \end{pmatrix},\ \cancel{A^9 = \begin{pmatrix} 21 & 34 \\ 34 & 55 \end{pmatrix}},\ \cancel{A^{10} = \begin{pmatrix} 34 & 55 \\ 55 & 89 \end{pmatrix}},$$

$$\cancel{A^{11} = \begin{pmatrix} 55 & 89 \\ 89 & 144 \end{pmatrix}},\ \cancel{A^{12} = \begin{pmatrix} 89 & 144 \\ 144 & 233 \end{pmatrix}},\ \cancel{A^{13} = \begin{pmatrix} 144 & 233 \\ 233 & 377 \end{pmatrix}},\ \cancel{A^{14} = \begin{pmatrix} 233 & 377 \\ 377 & 610 \end{pmatrix}}$$

$$\cancel{A^{15} = \begin{pmatrix} 377 & 610 \\ 610 & 987 \end{pmatrix}},\ A^{16} = \begin{pmatrix} 610 & 987 \\ 987 & 1597 \end{pmatrix}$$

Example 2 - sol2: Divide And Conquer

- A^{20} 과 같은 경우는?

$$\begin{aligned} 20 &= 2 \times (10) \\ &= 2 \times (2 \times 5) \\ &= 2 \times (2 \times (2 \times 2 + 1)) \end{aligned}$$

  이므로

$$A^1 \to A^2 \to A^4 \to A^5 \to A^{10} \to A^{20}$$

  을 차례로 계산하면 된다.

- 충분히 큰 n 에 대해 약 $\log_2 n$ 번의 행렬곱으로 해결할 수 있으므로 $n = 10^6 \approx 2^{20}$ 일 때도 대략 20 번의 행렬곱이면 충분하다.

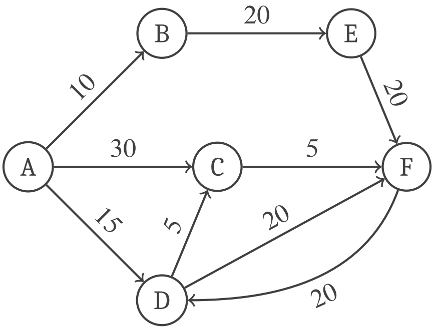
Example 2 - sol2: Divide And Conquer

파이썬으로 구현해보자.

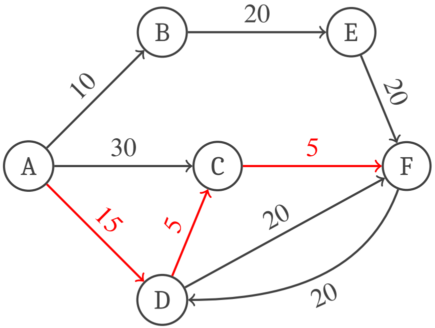
```python
def matrix_product(A, B):  # 행렬곱 함수, 두 행렬 A, B를 입력받아 AB를 리턴
    return [[sum(A[j][i] * B[i][k] for i in range(len(B))) for k in range(len(B[0]))] for j in range(len(A))]

def fast_power(A, n):  # 분할정복을 이용한 빠른 행렬 거듭제곱, A^n리턴
    if n == 1:  # n이 1이면 그냥 A를 리턴
        return A
    if n % 2 == 0:  # n이 짝수이면 n을 2로 나누어 A^(n/2)를 계산하고 곱하여 리턴
        B = fast_power(A, n // 2)
        return matrix_product(B, B)
    else:  # 그외, 즉 n이 홀수이면 n에서 1을 빼주어 짝수로 만들어 재귀
        return matrix_product(A, fast_power(A, n - 1))

A = [[0, 1], [1, 1]]  # 행렬
print(fast_power(A, 16))
```

Shortest Path: Dijkstra

A에서 F까지 가는 경로 중에서 가장 빠른 경로는?



다익스트라(using heap) 알고리즘을 통해 해결할 수 있다. 지도에서 길찾기 알고리즘의 기초가 된다.



- 링크: acmicpc.net
- 약 25000개 이상의 문제들이 있다.
- 단계별로 풀어보기, 문제집 등이 있어 자신에게 맞는 문제들을 쉽게 찾아볼 수 있다.
- 질문 게시판이 있어 모르는 부분에 대한 질문이 가능하다.
- 랭킹에서 맞은 문제 수에 대한 랭킹을 확인할 수 있다.
- 이메일 인증을 통해 학교 등록을 하면 중앙대 랭킹에도 아이디가 보인다.
- 다음 페이지의 solve.ac와 잘 연동되어있다.

solved.ac

- 링크: solved.ac
- 회원가입 후 백준과 연동할 수 있다.
- 푼 문제에 따라 자신의 티어가 결정되어 게임을 하는 듯한 느낌을 준다.
- 프로그래밍 언어 사용에 쉽게 익숙해질 수 있는 문제들인 새싹 티어 문제,
- 문제해결에서 자주 마주하게 되는 주제들과 트릭을 단계별로 선별한 CLASS를 유용하게 이용해보자.
- 레이팅에 따른 전체 랭킹, 중앙대 랭킹도 볼 수 있다.

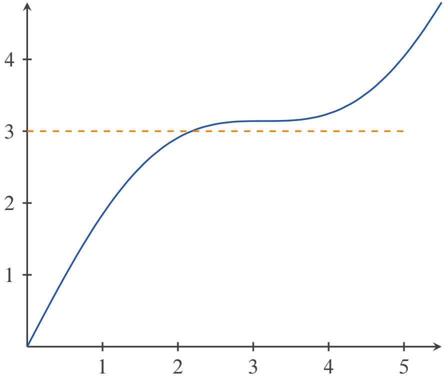
Table of Contents

Polynomial + trigonometric?

2 Find numerical solution

다음 방정식의 해를 찾아보자.

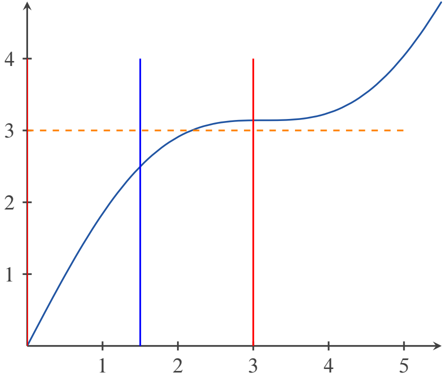
$$x + \sin x = 3$$



Binary search

2 Find numerical solution

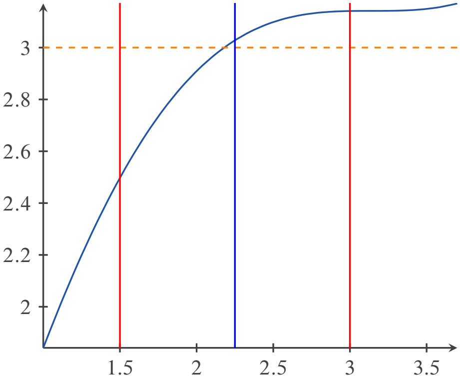
- 해가 $x = 0$과 $x = 3$ 사이에 존재하는 것은 확실해보인다.
- 그렇다면 그 절반인 $x = 1.5$를 기준으로 해는 어디에 존재할까?
- $f(x) = x + \sin x$에 대해 $f(1.5) < 3$이므로 오른쪽 구간에 존재할 것이다.



Binary search

2 Find numerical solution

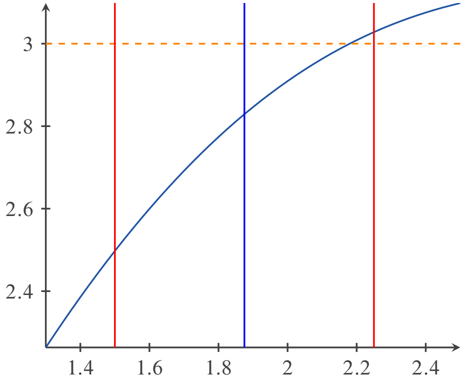
- 이제 $1.5 < x < 3$에 대해서 같은 작업을 반복해보자.
- 두 구간의 절반인 $x = 2.25$에서 $f(2.25) > 3$이므로 해는 왼쪽 구간에 존재할 것이다.



Binary search

2 Find numerical solution

- 이제 구간은 $1.5 < x < 2.25$가 된다.
- 두 구간의 절반인 $x = 1.875$에서 $f(1.875) < 3$이므로 해는 오른쪽 구간에 존재할 것이다.
- 이와 같은 작업을 충분히 반복하면 해가 약 $x \approx 2.17975706648003001294$임을 알 수 있을 것이다.



Binary search

- 처음 구간 $[0, 3]$ 에서 $[1.5, 3]$, $[1.5, 2.25]$ 와 같이 해가 존재하는 구간의 길이는 $\frac{1}{2}$ 배가 된다.
- 만약 임의의 x 에 대해 $f(x) = x + \sin x$ 의 값을 충분히 정확하게 계산할 수 있고, 이와 같은 작업을 n 번 반복한다면 구간의 길이는 처음의 2^{-n} 배가 된다.
- 단지 20 번만 반복하더라도 약 10^{-6} 배가 된다.
- 만약 구간을 10^{-6} 개로 나누어 각 점에서 $f(x)$ 를 계산 후, 해와 가장 가까운 점을 찾는다면 10^{6} 번 계산해야 하므로 엄청난 차이임을 알 수 있다.

- 앞선 방식을 이분 탐색이라고 한다.
- $f(x) = 0$의 수치해를 찾는 다른 방법으로 newton-raphson method, secant method 등이 있고, 일반적으로 이분 탐색보다 더 빠르게 해에 수렴하지만, 특수한 상황에서는 적용할 수 없다는 단점도 가지고 있다.
- $f(x) = 0$의 해를 찾는 이분 탐색과 비슷한, 볼록성이 알려진 함수의 극댓값이나 극솟값을 찾는 삼분 탐색도 존재한다.

Binary search

2 Find numerical solution

파이썬으로 구현해보자.

```python
import math  # sin 함수 사용을 위한 라이브러리
def f(x):  # f(x) 정의
    return x + math.sin(x)
left = 0  # 해가 존재하는 구간의 왼쪽 끝
right = 3  # 해가 존재하는 구간의 오른쪽 끝
for i in range(100):  # 100번 반복
    mid = (left + right) / 2  # 구간의 중간값
    if f(mid) > 3:  # 3보다 크면 구간의 오른쪽을 mid로
        right = mid
    else:  # 그렇지 않으면 구간의 왼쪽을 mid로 한다.
        left = mid
print(left)
```

Table of Contents

Fast Fourier Transform

- 알고리즘을 설명하는 데 이것을 빼놓기는 힘들 것이다.
- 여기서 FFT의 역사와 간단한 원리를 굉장히 잘 설명하고 있다.
- 먼저 DFT(Discrete Fourier Transform)는 다음과 같이 정의된다. 여기서 x_n은 $0 \le n \le N-1$, 즉 N개의 신호를 의미한다.

$$f_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad k = 0, 1, \cdots, N-1$$

- 이를 이용해 간단한 예시로 직접 DFT를 수행해보자.

DFT

3 Fast Fourier Transform

- $x_n = [1, 3, 2]$이라 하자.
- $$f_0 = \sum_{n=0}^{2} x_n e^{-\frac{2\pi i}{3} 0\cdot n} = \sum_{n=0}^{2} x_n = 6$$
- $$f_1 = \sum_{n=0}^{2} x_n e^{-\frac{2\pi i}{3} 1\cdot n} = \sum_{n=0}^{2} x_n e^{-\frac{2\pi n i}{3}} = 1 + 3e^{-\frac{2\pi i}{3}} + 2e^{-\frac{4}{3}\pi i} = -\frac{3}{2} - \frac{\sqrt{3}}{2} i$$
- $$f_2 = \sum_{n=0}^{2} x_n e^{-\frac{2\pi i}{3} 2\cdot n} = \sum_{n=0}^{2} x_n e^{-\frac{4\pi n i}{3}} = 1 + 3e^{-\frac{4\pi i}{3}} + 2e^{-\frac{8\pi i}{3}} = -\frac{3}{2} + \frac{\sqrt{3}}{2} i$$

Using Divide and Conquer

- 일반적으로 N 개의 수를 모두 DFT하기 위해 N^2 번의 계산이 필요하다. $\rightarrow \mathcal{O}(n^2)$
- 그런데 크기 N 인 수열을 크기 $\frac{N}{2}$ 인 두 개의 수열로 나누어 계산 후 합쳐도 된다면, $\mathcal{T}(n)$ 은 다음과 같이 쓸 수 있다.

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

- 예를 들어 크기 64인 수열을 단순 DFT를 통해 계산한다면 $64^2 = 4096$ 번의 계산이 필요하지만, 크기 32인 수열 두 개로 나누어 계산한다면 $2 \cdot 32^2 + 64 = 2048 + 64$ 번의 계산이 필요하다.
- 그런데 크기 32인 수열 또한 마찬가지로 크기 16인 수열 두 개로 나누어 계산할 수 있으므로 반복하면 단지 $\mathcal{O}(n \log_2 n)$ 안에 계산할 수 있다.
- 예를 들어 크기 64인 수열은 대략 $64 \log_2 64 = 64 \cdot 6 = 384$ 번의 계산으로 푸리에 변환을 할 수 있다.

$$
\begin{aligned}
\mathcal{T}(n) &= 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n) \\
&= 2\left(2\mathcal{T}(\frac{n}{2^2}) + \mathcal{O}(\frac{n}{2})\right) + \mathcal{O}(n) \\
&= 2^2\mathcal{T}\left(\frac{n}{2^2}\right) + \mathcal{O}(2n) \\
&\ \ \vdots \\
&= 2^k\mathcal{T}\left(\frac{n}{2^k}\right) + \mathcal{O}(kn), \quad (n = 2^k) \\
&= n\mathcal{T}(1) + \mathcal{O}(n\log_2 n) \\
&= \mathcal{O}(n\log_2 n)
\end{aligned}
$$

Using Divide and Conquer

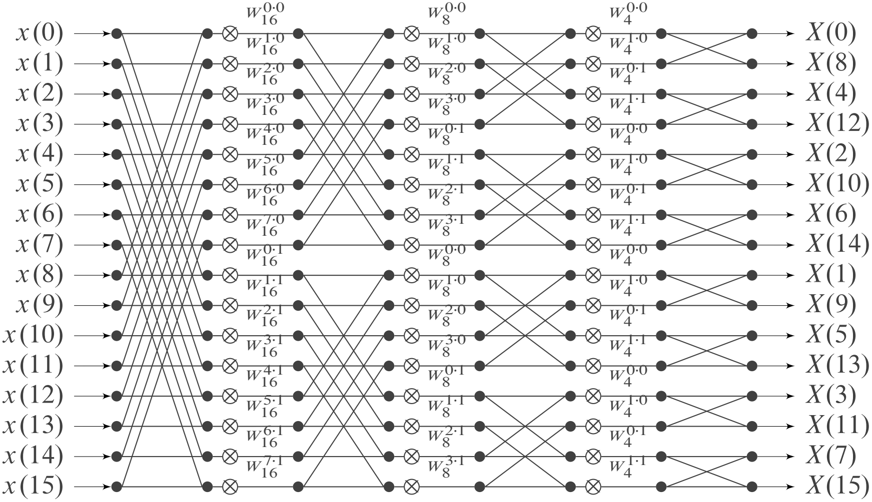
- 그렇다면 어떻게 두 개의 수열로 나눌 수 있을까? 다음 정의를 조금 풀어 써보자.

$$
\begin{aligned}
f_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn}, \quad k = 0, 1, \cdots, N-1 \\
&= x_0 e^{-\frac{2\pi i}{N}0k} + x_1 e^{-\frac{2\pi i}{N}1k} + x_2 e^{-\frac{2\pi i}{N}2k} + x_3 e^{-\frac{2\pi i}{N}3k} + \cdots \\
&= \left(x_0 e^{-\frac{2\pi i}{N}0k} + x_2 e^{-\frac{2\pi i}{N}2k} + \cdots \right) + \left(x_1 e^{-\frac{2\pi i}{N}1k} + x_3 e^{-\frac{2\pi i}{N}3k} + \cdots \right) \\
&= \left(x_0 e^{-\frac{2\pi i}{N/2}0k} + x_2 e^{-\frac{2\pi i}{N/2}k} + \cdots \right) + e^{-\frac{2\pi i}{N}k} \left(x_1 e^{-\frac{2\pi i}{N}0k} + x_3 e^{-\frac{2\pi i}{N}2k} + \cdots \right) \\
&= \left(x_0 e^{-\frac{2\pi i}{N/2}0k} + x_2 e^{-\frac{2\pi i}{N/2}k} + \cdots \right) + e^{-\frac{2\pi i}{N}k} \left(x_1 e^{-\frac{2\pi i}{N/2}0k} + x_3 e^{-\frac{2\pi i}{N/2}k} + \cdots \right) \\
&= f_{\text{even}} + e^{-\frac{2\pi i}{N}k} f_{\text{odd}}
\end{aligned}
$$

- 짝수항과 홀수항으로 나누어 각각 재귀적으로 FFT를 쓰면 해결할 수 있다.

Butterfly diagram

FFT diagram. 마치 나비 모양과 같다고 해서 butterfly diagram이라고도 부른다.



파이썬으로 구현해보자.

```python
from cmath import exp, pi
def fft(a, inv = 0):  # 수열 a를 fft한다. inv가 1이면 역변환이다.
        N = len(a)  # a의 길이이다.
        if N == 1: return a  # 길이가 1이면 그냥 리턴한다.
        A_even = fft(a[0::2], inv)  # 0부터 2단위로 -> 짝수항을 나눠 fft 한다.
        A_odd = fft(a[1::2], inv)  # 홀수항을 나눠 fft한다.
        w_N = [exp(-2j * pi * n / N * (1 - 2 * inv)) for n in range(N//2)]  # w_N을 계산한다. 1-2*inv는 inv가 0이면 1, 1이면 -1이다.
        return [A_even[n] + w_N[n] * A_odd[n] for n in range(N//2)] + [A_even[n] - w_N[n] * A_odd[n] for n in range(N//2)]  # 각각을 합쳐 리턴한다.
```

- 컴퓨터 과학에서 FFT의 대표적인 활용 예는 곱셈이다.
- 일반적으로 두 n 자리수의 곱셈의 시간 복잡도는 $\mathcal{O}(n^2)$ 이다.
- 그런데 FFT를 이용하면 이를 $\mathcal{O}(n \log_2 n)$ 으로 줄일 수 있다.
- (참고) 카라추바 알고리즘은 1962년 공개된 빠른 곱셈 알고리즘으로, $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.585})$ 의 시간 복잡도를 가진다.
- 파이썬의 libmpdec 라이브러리는 적당히 큰 수의 곱에 대해 카라추바 알고리즘을, 매우 큰 수의 곱에 대해 NTT(Number Theoretic Transform)를 사용한다.

파이썬으로 구현해보자. 앞선 FFT 코드 바로 아래에 작성해야 한다.

```
a = [4, 3, 2, 0, 0, 0, 0, 0]  # 4+3x+2x^2
b = [8, 7, 6, 0, 0, 0, 0, 0]  # 8+7x+6x^2
A = fft(a, 0)  # a를 fft
B = fft(b, 0)  # b를 fft
C = [A[i] * B[i] for i in range(8)]  # fft한 결과끼리 곱셈
D = fft(C, 1)  # 역변환
for i in range(len(D)):
        D[i] = round(D[i].real / 8)  # 실수부분을 길이 8로 나누고 반올림(오차)
print(D)  # 32+52x+61x^2+32x^3+12x^4
```

Application - All Possible Sum

- FFT를 이용해서 할 수 있는 재밌는 것으로 두 수열의 원소끼리의 덧셈으로 가능한 모든 가능한 값과 순서쌍의 갯수를 구하는 것이다.
- $a_i = [1, 4, 6]$, $b_i = [1, 3, 5]$라 하자. 모든 i, j에 대해 $a_i + b_j$로 가능한 값을 구해보자.

Application - All Possible Sum

- $a_i = [1, 4, 6], b_i = [1, 3, 5]$
- $2 = 1 + 1, 4 = 1 + 3, 5 = 4 + 1, 6 = 1 + 5, 11 = 6 + 5$
- $7 = 4 + 3 = 6 + 1, 9 = 4 + 5 = 6 + 3$
- 따라서 2, 4, 5, 6, 7, 9, 11이 가능하며, 7, 9는 각각 2쌍이 존재한다.
- 그런데 $(x + x^4 + x^6)(x + x^3 + x^5)$를 계산해보면 놀랍게도 $x^2 + x^4 + x^5 + x^6 + 2x^7 + 2x^9 + x^{11}$이다.
- 각 원소를 지수로 하는 다항식을 만든 후, FFT를 이용하여 곱셈하면 빠르게 셀 수 있다.

Application

3 Fast Fourier Transform

파이썬으로 구현해보자. 앞선 코드에서 조금만 수정하면 된다.

```
a = [0, 1, 0, 0, 1, 0, 1, 0] + [0] * 8  # x + x^4 + x^6
b = [0, 1, 0, 1, 0, 1, 0, 0] + [0] * 8  # x + x^3 + x^5
A = fft(a, 0)  # a를 fft
B = fft(b, 0)  # b를 fft
C = [A[i] * B[i] for i in range(16)]  # fft한 결과끼리 곱셈
D = fft(C, 1)  # 역변환
for i in range(len(D)):
        D[i] = round(D[i].real / 16)  # 실수부분을 길이 16로 나누고 반올림 (오차)
print(D)  # x^2 + x^4 + x^5 + x^6 + 2x^7 + 2x^9 + x^11
```