R E P O R T

실습 및 과제 1

과목명	알고리즘및실습
분반	2 분 반
교수	한 연 희
학번	2020136129
이름	최 수 연
제출일	2022년 3월 20일 일요일

목차

서론

컴퓨터 과학에서의 검색 알고리즘은 말 그대로 검색 문제를 해결하는 알고리즘을 뜻한다. 검색 알고리즘은 검색 구조에 따라 분류 가능한데, 본 과제에서는 순차검색 알고리즘과 이진검색 알고리즘, 두 종류의 검색 알고리즘을 다룬다.

순차검색 알고리즘이란 순차적으로 비교하면서 검색하는 알고리즘으로 선형 검색 방법이다. 순차검색 알고리즘과 달리 이진검색 알고리즘은 정렬된 상태에서만 검색할 수 있고, 키와 가운데 요소를 비교하는 방식으로 알고리즘이 진행되며 재귀적으로도 이진검색 알고리즘을 구현할 수 있다.

피보나치수열은 제0항을 0, 제1항을 1로 두고 두 번째 항부터는 바로 앞의 두 수를 더한 수를 놓는 방식으로 수를 나열한다. 피보나치수열은 동일한 계산 과정을 반복하는 작업이므로 재귀 알고리즘과 반복 알고리즘 구현이 가능하다.

본 과제는 순차검색과 이진검색, 재귀적 이진검색에 대한 검색 알고리즘 및 피보나치수열에 대한 재귀 알고리즘과 반복 알고리즘을 python 프로그램으로 작성한다. 본 과제를 통해 순차검색 및 이진검색 알고리즘과 피보나치수열에 대한 재귀 및 반복 알고리즘을 이해하고 설명할 수 있다.

본론

문제 1. 순차검색과 이진검색

1. 프로그램 코드

```
import random
import time
import math #floor 함수 사용을 위한 math 모듈 import

def main():
    num = 1_000_000 #num 값을 변경하여 수행시간 비교 및 분석
    s = []

    for value in range(num):
        s.append(random.randint(0, num))
    #배열 s에 0부터 (num-1)까지의 수를 무작위로 (num-1)번 추가

    key = random.randint(0, num)
    #찾고자 하는 key 값을 0부터 (num-1)까지의 수 중 무작위로 선정

    start = time.time()
    location = sequential_search(s, key)
    end = time.time()
    #순차검색 알고리즘 수행시간 측정

    #print(s)
    print("[Sequential Search Result]")
    print("Key value {0}: location {1}".format(key, location))
    print("Elapsed Time: {0:0.8f}ms".format((end - start) * 1_000))
    #1_000을 곱하면 시간 단위가 ms로 바뀜
    print()
    #순차검색 알고리즘 key 값, 위치 및 수행시간 출력

    s.sort() #이진검색 알고리즘을 위한 정렬

    start = time.time()
    location = binary_search(s, key)
    end = time.time()
```

```
    #이진검색 알고리즘 수행시간 측정
    #print(s)
    print("[Binary Search Result]")
    print("Key value {0}: location {1}".format(key, location))
    print("Elapsed Time: {0:0.8f}ms".format((end - start) * 1_000))
    print()
    #이진검색 알고리즘 key 값, 위치 및 수행시간 출력

    start = time.time()
    location = recursive_binary_search(s, key, 0, num - 1)
    end = time.time()
    #재귀적 이진검색 알고리즘 수행시간 측정

    #print(s)
    print("[Recursive Binary Search Result]")
    print("Key value {0}: location {1}".format(key, location))
    print("Elapsed Time: {0:0.8f}ms".format((end - start) * 1_000))
    print()
    #재귀적 이진검색 알고리즘 key 값, 위치 및 수행시간 출력

def sequential_search(s, key): #순차검색 알고리즘 함수
    num = len(s)
    location = 0

    while location < num and s[location] != key:
      location += 1

    if location >= num:
      location = -1

    return location


def binary_search(s, key): #이진검색 알고리즘 함수
    num = len(s)
    low = 0
    high = num - 1
    location = -1

    while low <= high and location == -1:
```

```
        mid = math.floor((low + high) / 2)

        if key == s[mid]:
          location = mid

        elif key < s[mid]:
          high = mid - 1

        else: low = mid + 1

      return location


def recursive_binary_search(s, key, low, high): #재귀적 이진검색 알고리즘 함수
      mid = round((low + high) / 2)

      if low <= high:

        if key == s[mid]:
          return mid

        elif key < s[mid]:
          return recursive_binary_search(s, key, low, mid - 1)

        else:
          return recursive_binary_search(s, key, mid + 1, high)

      else: return -1

if __name__ == "__main__":
      main()
```

2. 프로그램 코드 설명

순차검색 알고리즘 함수

- 순차검색 알고리즘 함수의 매개변수로 리스트 s와 key 값을 받는다.
- 리스트 s의 크기를 받아 num에 저장하고, 위치 location을 0으로 초기화 선언한다.
- while 문을 사용하여 현재 위치인 location이 리스트 s의 크기인 num보다 작고, 리스트 s에서 인덱스 location에 위치한 값이 찾고자 하는 key 값과 같지 않으면 location에 1을 더한다.
- if 문을 사용하여, 만약 location이 num과 같거나 클 경우, location에 정수 -1을 넣고 반환하여, 리스트 s에서는 key 값이 존재하지 않음을 표시한다.
- 만약 location이 num보다 작은데, 리스트 s에서 인덱스 location에 위치한 값이 찾고자 하는 key 값과 같을 경우, while 문과 if 문이 실행되지 않고, 해당 location 값을 바로 반환한다.

이진검색 알고리즘 함수

- 이진검색 알고리즘 함수의 매개변수로 리스트 s와 key 값을 받는다.
- 리스트 s의 크기를 받아 num에 저장하고, 위치 location을 -1로 초기화 선언한다.
- low 값은 리스트 s의 범위 중 가장 작은 인덱스인 0, high 값은 리스트 s의 범위 중 가장 큰 인덱스인 (num-1) 값을 넣어 초기화 선언한다.
- while 문을 사용하여, low가 high보다 작거나 같아야 한다는 전제 조건을 명시하고, 이와 동시에 location 값이 -1일 때만 while 문이 수행되도록 한다.
- 위 조건에 만족할 경우, low와 high를 이용하여 low와 high의 중간값을 구하여 mid에 값을 넣는다. 이때 math 모듈에서 floor 함수를 불러 low와 high의 중간값이 실수일 경우 내림하여 정수를 반환하도록 한다.
- 만약 key 값이 리스트 s에서 인덱스 mid에 해당하는 값과 같을 경우, location에 mid 값을 넣고, 해당 while 문 조건에 부합하지 않으므로 while 문을 빠져나와 location 값을 반환한다.
- 그렇지 않고 만약 key 값이 리스트 s에서 인덱스 mid에 해당하는 값보다 작을 경우, high에 mid에서 1을 뺀 값을 넣어 그다음 key 탐색 범위를 좁히도록 한다.
- 위 두 개의 조건에 모두 해당하지 않을 경우, 즉 key 값이 리스트 s에서 인덱스 mid에 해당하는 값보다 클 경우, low에 mid에서 1을 더한 값을 넣어 그다음 key 탐색 범위를 좁힌다.

재귀적 이진검색 알고리즘 함수

- 재귀적 이진검색 알고리즘의 함수에서는 매개변수로 리스트 s, key, low, high 값을 모두 받는다.(재귀적 이진검색 알고리즘의 수행시간 측정 부분 코드에서 매개변수가 4개로 구성되어 있음.)
- 반올림 함수 round를 사용하여 low와 high의 중간값을 반올림하여 mid에 저장한다.
- 만약 low가 high보다 작거나 같을 때만 if 문이 실행되도록 코드를 작성한다. 만약 low가 high보다 클 경우, 정수 -1을 반환한다.
- 위 if 문 안에 if 문을 또 작성하여 이중 if 문을 만든다. 내부에 있는 if 문은 key 값과 리스트 s에서 인덱스 mid에 해당하는 값을 비교하는 방식으로 작성한다.
- 내부 if 문에서 만약 key 값이 리스트 s에서 인덱스 mid에 해당하는 값과 같을 경우,

mid 값을 반환하고, 그렇지 않고 key 값이 리스트 s에서 인덱스 mid에 해당하는 값보다 작을 경우, 해당 재귀적 이진검색 알고리즘 함수를 다시 호출하는데, 이때 high 값을 (mid - 1)로 바꾸어 해당 함수를 호출한다. 만약 위 두 조건 다 만족하지 않을 때, 즉 key 값이 리스트 s의 인덱스 mid에 해당하는 값보다 클 경우, low 값을 (mid + 1)로 바꾸어 해당 함수를 다시 호출한다.

3. 프로그램 수행화면

num = 1,000

```
[Sequential Search Result]
Key value 420: location -1
Elapsed Time: 0.17976761ms

[Binary Search Result]
Key value 420: location -1
Elapsed Time: 0.01478195ms

[Recursive Binary Search Result]
Key value 420: location -1
Elapsed Time: 0.02074242ms
```

```
[Sequential Search Result]
Key value 975: location 565
Elapsed Time: 0.05841255ms

[Binary Search Result]
Key value 975: location 970
Elapsed Time: 0.00715256ms

[Recursive Binary Search Result]
Key value 975: location 970
Elapsed Time: 0.01001358ms
```

```
[Sequential Search Result]
Key value 279: location 301
Elapsed Time: 0.03170967ms

[Binary Search Result]
Key value 279: location 278
Elapsed Time: 0.00929832ms

[Recursive Binary Search Result]
Key value 279: location 278
Elapsed Time: 0.01358986ms
```

```
[Sequential Search Result]
Key value 155: location 388
Elapsed Time: 0.04363060ms

[Binary Search Result]
Key value 155: location 155
Elapsed Time: 0.00810623ms

[Recursive Binary Search Result]
Key value 155: location 155
Elapsed Time: 0.01549721ms
```

```
[Sequential Search Result]
Key value 88: location -1
Elapsed Time: 0.10609627ms

[Binary Search Result]
Key value 88: location -1
Elapsed Time: 0.00858307ms

[Recursive Binary Search Result]
Key value 88: location -1
Elapsed Time: 0.02026558ms
```

```
[Sequential Search Result]
Key value 788: location -1
Elapsed Time: 0.11920929ms

[Binary Search Result]
Key value 788: location -1
Elapsed Time: 0.01025200ms

[Recursive Binary Search Result]
Key value 788: location -1
Elapsed Time: 0.01716614ms
```

Sequential Search Result의 평균 수행시간 = 0.08980433(ms)

Binary Search Result의 평균 수행시간 = 0.009695688(ms)

Recursive Binary Search Result의 평균 수행시간 = 0.016212465(ms)

num = 10,000

```
[Sequential Search Result]
Key value 4998: location -1
Elapsed Time: 1.08695030ms

[Binary Search Result]
Key value 4998: location -1
Elapsed Time: 0.01573563ms

[Recursive Binary Search Result]
Key value 4998: location -1
Elapsed Time: 0.03361702ms
```

```
[Sequential Search Result]
Key value 3880: location -1
Elapsed Time: 1.09839439ms

[Binary Search Result]
Key value 3880: location -1
Elapsed Time: 0.01192093ms

[Recursive Binary Search Result]
Key value 3880: location -1
Elapsed Time: 0.02241135ms
```

```
[Sequential Search Result]
Key value 9389: location 1648
Elapsed Time: 0.21100044ms

[Binary Search Result]
Key value 9389: location 9383
Elapsed Time: 0.00977516ms

[Recursive Binary Search Result]
Key value 9389: location 9382
Elapsed Time: 0.01955032ms
```

```
[Sequential Search Result]
Key value 5847: location 3669
Elapsed Time: 0.44417381ms

[Binary Search Result]
Key value 5847: location 5829
Elapsed Time: 0.01072884ms

[Recursive Binary Search Result]
Key value 5847: location 5829
Elapsed Time: 0.01978874ms
```

```
[Sequential Search Result]
Key value 4082: location -1
Elapsed Time: 1.27768517ms

[Binary Search Result]
Key value 4082: location -1
Elapsed Time: 0.01120567ms

[Recursive Binary Search Result]
Key value 4082: location -1
Elapsed Time: 0.02336502ms
```

```
[Sequential Search Result]
Key value 808: location 6670
Elapsed Time: 0.67162514ms

[Binary Search Result]
Key value 808: location 762
Elapsed Time: 0.01239777ms

[Recursive Binary Search Result]
Key value 808: location 762
Elapsed Time: 0.02050400ms
```

Sequential Search Result의 평균 수행시간 = 0.798304875(ms)

Binary Search Result의 평균 수행시간 = 0.011960667(ms)

Recursive Binary Search Result의 평균 수행시간 = 0.023206075(ms)

num = 100,000

```
[Sequential Search Result]
Key value 61942: location -1
Elapsed Time: 11.36732101ms

[Binary Search Result]
Key value 61942: location -1
Elapsed Time: 0.01907349ms

[Recursive Binary Search Result]
Key value 61942: location -1
Elapsed Time: 0.02479553ms
```

```
[Sequential Search Result]
Key value 95110: location 33867
Elapsed Time: 3.87358665ms

[Binary Search Result]
Key value 95110: location 95162
Elapsed Time: 0.01955032ms

[Recursive Binary Search Result]
Key value 95110: location 95163
Elapsed Time: 0.03910065ms
```

```
[Sequential Search Result]
Key value 27886: location 40539
Elapsed Time: 4.88138199ms

[Binary Search Result]
Key value 27886: location 27820
Elapsed Time: 0.02098083ms

[Recursive Binary Search Result]
Key value 27886: location 27820
Elapsed Time: 0.01668930ms
```

```
[Sequential Search Result]
Key value 46805: location -1
Elapsed Time: 12.09402084ms

[Binary Search Result]
Key value 46805: location -1
Elapsed Time: 0.01740456ms

[Recursive Binary Search Result]
Key value 46805: location -1
Elapsed Time: 0.02741814ms
```

```
[Sequential Search Result]
Key value 45238: location 45689
Elapsed Time: 5.42569160ms

[Binary Search Result]
Key value 45238: location 44965
Elapsed Time: 0.01955032ms

[Recursive Binary Search Result]
Key value 45238: location 44966
Elapsed Time: 0.02741814ms
```

```
[Sequential Search Result]
Key value 32346: location 22371
Elapsed Time: 2.75468826ms

[Binary Search Result]
Key value 32346: location 32597
Elapsed Time: 0.01549721ms

[Recursive Binary Search Result]
Key value 32346: location 32597
Elapsed Time: 0.02932549ms
```

Sequential Search Result의 평균 수행시간 = 6.732781725(ms)

Binary Search Result의 평균 수행시간 = 0.018676122(ms)

Recursive Binary Search Result의 평균 수행시간 = 0.027457875(ms)

num = 1,000,000

```
[Sequential Search Result]
Key value 865414: location 691077
Elapsed Time: 79.14900780ms

[Binary Search Result]
Key value 865414: location 865977
Elapsed Time: 0.01907349ms

[Recursive Binary Search Result]
Key value 865414: location 865977
Elapsed Time: 0.03600121ms
```

```
[Sequential Search Result]
Key value 592414: location 440175
Elapsed Time: 52.99353600ms

[Binary Search Result]
Key value 592414: location 592296
Elapsed Time: 0.02264977ms

[Recursive Binary Search Result]
Key value 592414: location 592296
Elapsed Time: 0.03314018ms
```

```
[Sequential Search Result]
Key value 105907: location -1
Elapsed Time: 113.65723610ms

[Binary Search Result]
Key value 105907: location -1
Elapsed Time: 0.02861023ms

[Recursive Binary Search Result]
Key value 105907: location -1
Elapsed Time: 0.04506111ms
```

```
[Sequential Search Result]
Key value 763581: location -1
Elapsed Time: 116.11390114ms

[Binary Search Result]
Key value 763581: location -1
Elapsed Time: 0.01955032ms

[Recursive Binary Search Result]
Key value 763581: location -1
Elapsed Time: 0.04386902ms
```

```
[Sequential Search Result]
Key value 464640: location -1
Elapsed Time: 122.93148041ms

[Binary Search Result]
Key value 464640: location -1
Elapsed Time: 0.02837181ms

[Recursive Binary Search Result]
Key value 464640: location -1
Elapsed Time: 0.03600121ms
```

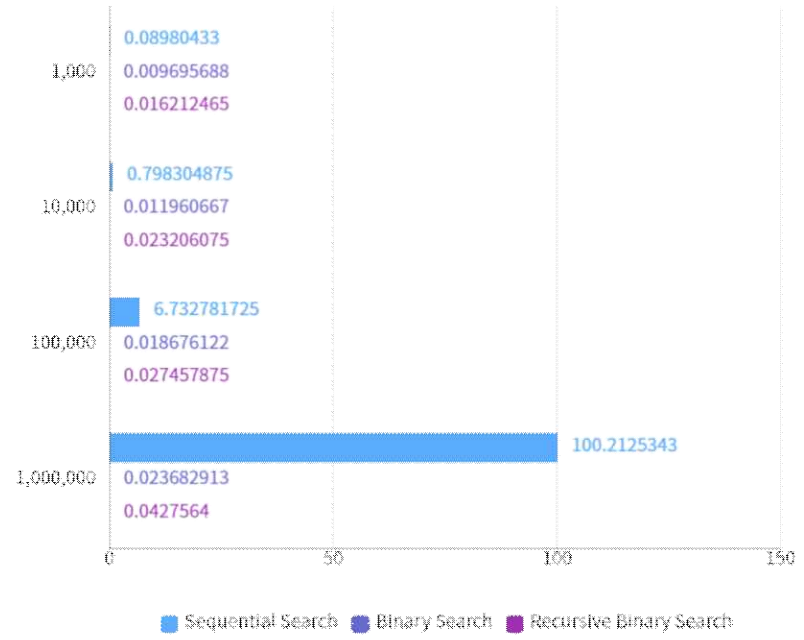
```
[Sequential Search Result]
Key value 342400: location -1
Elapsed Time: 116.43004417ms

[Binary Search Result]
Key value 342400: location -1
Elapsed Time: 0.02384186ms

[Recursive Binary Search Result]
Key value 342400: location -1
Elapsed Time: 0.06246567ms
```

Sequential Search Result의 평균 수행시간 = 100.2125343(ms)

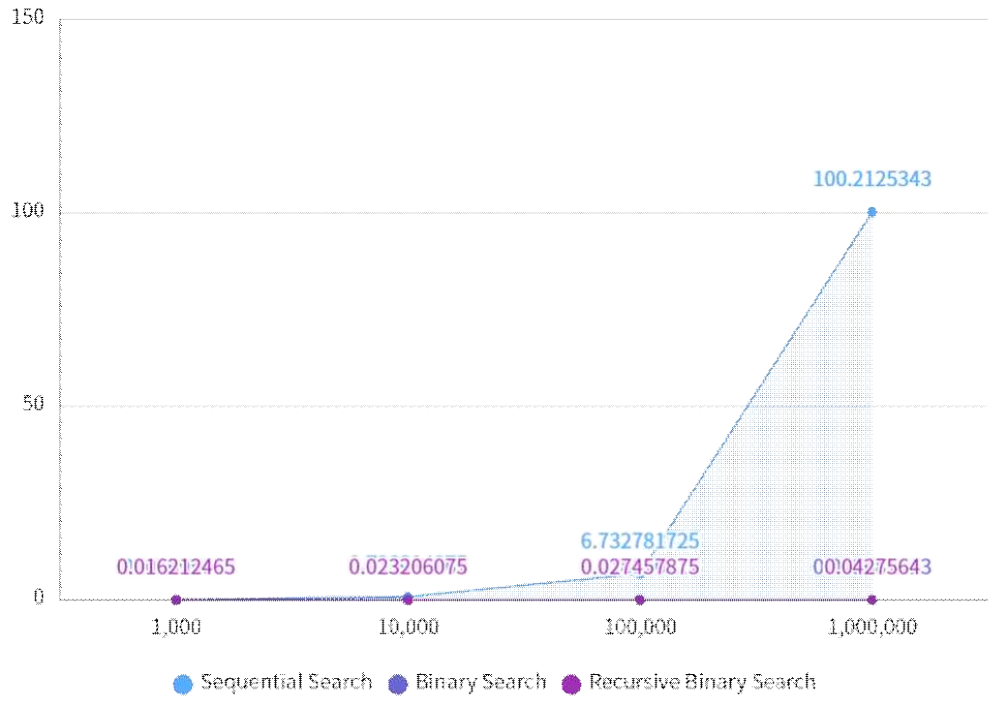
Binary Search Result의 평균 수행시간 = 0.023682913(ms)

Recursive Binary Search Result의 평균 수행시간 = 0.0427564(ms)

4. 수행시간 그래프

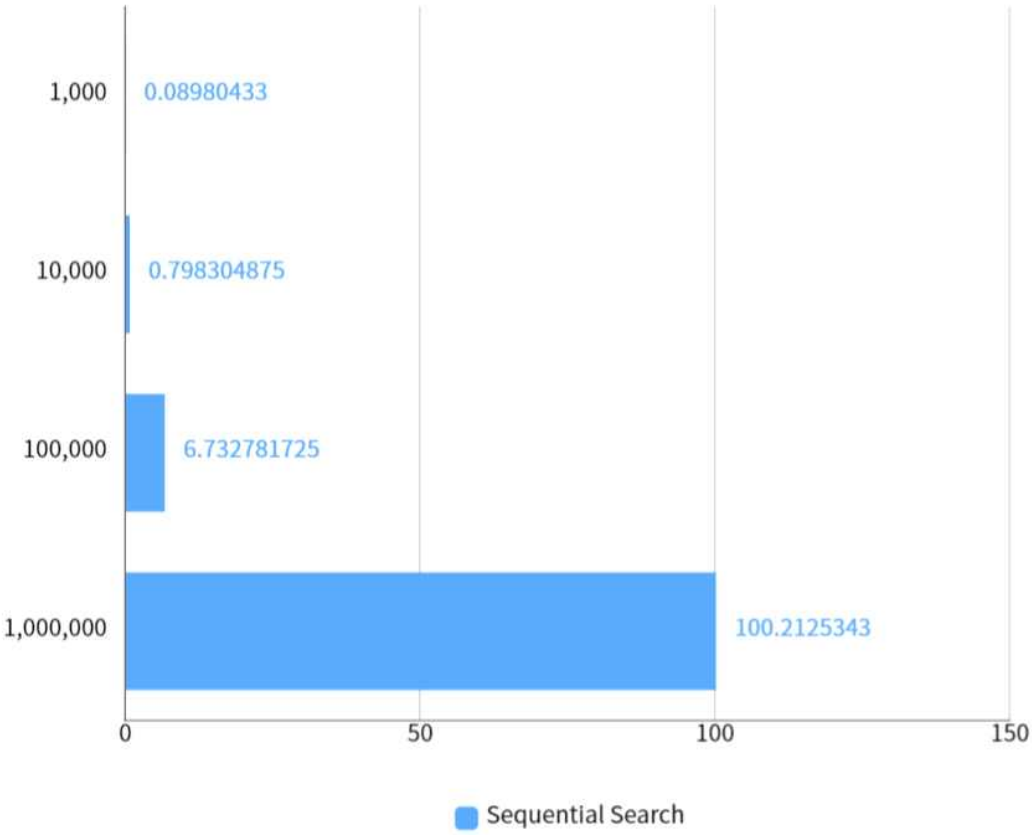


표[1-1]

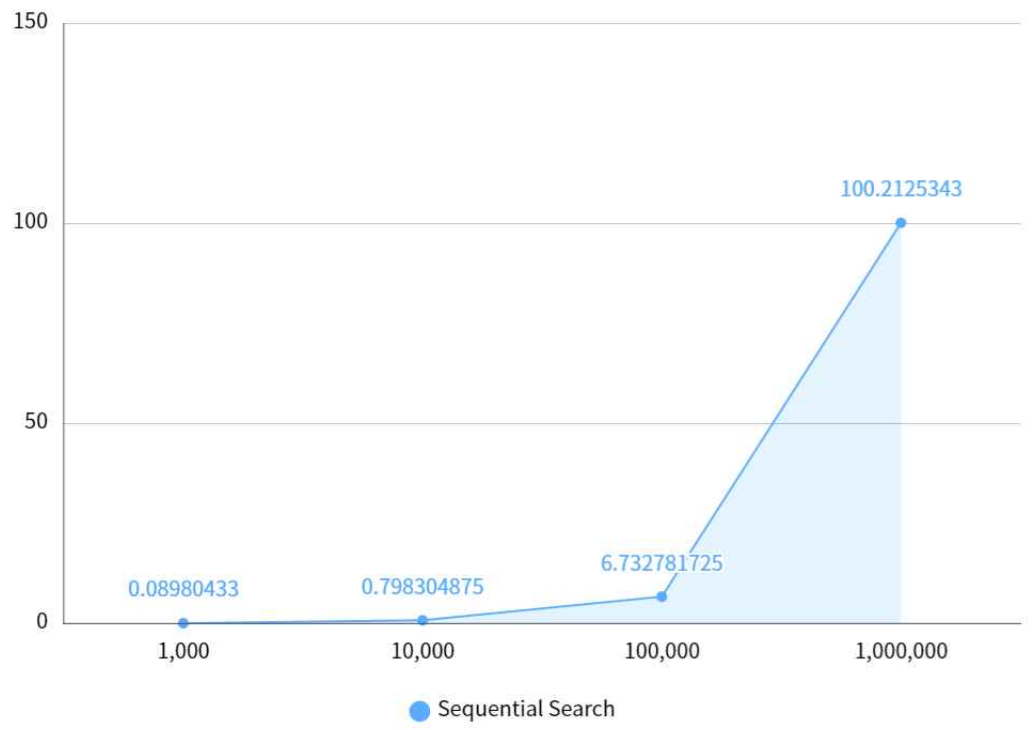


표[1-2]

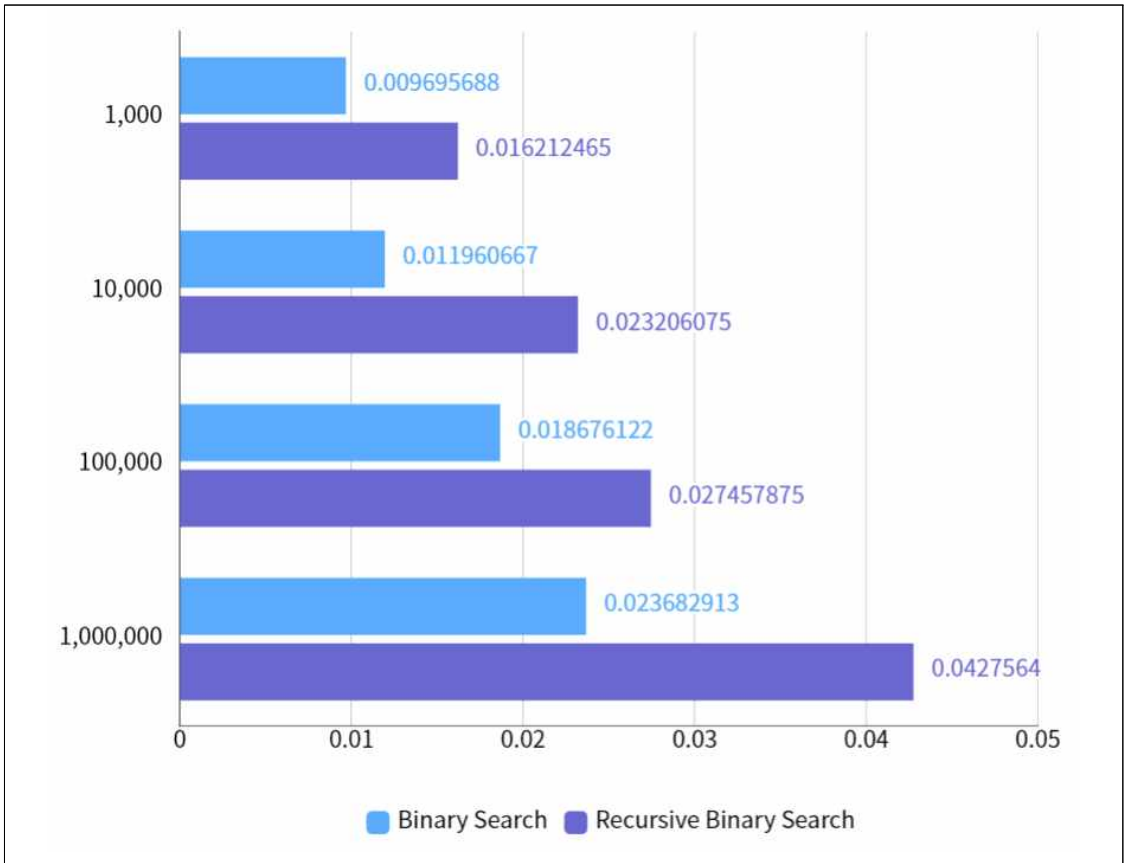
순차검색 알고리즘, 이진검색 알고리즘, 재귀적 이진검색 알고리즘을 여러 번 수행한 뒤 각각의 수행시간에 대한 평균을 내어 그래프로 나타낸 것이다. 본 그래프는 num이 각각 1,000, 10,000, 100,000, 1,000,000 일 때의 각각 수행시간이다. 순차검색 알고리즘의 값이 이진검색 및 재귀적 이진검색 알고리즘의 값에 비해 큰 결괏값이 도출되어 그래프의 가독성이 떨어지는 점을 보완하여 따로 나눈 그래프를 추가로 첨부하였다.


1,000
0.08980433
10,000
0.798304875
100,000
6.732781725
1,000,000
100.2125343
0
50
100
150
Sequential Search

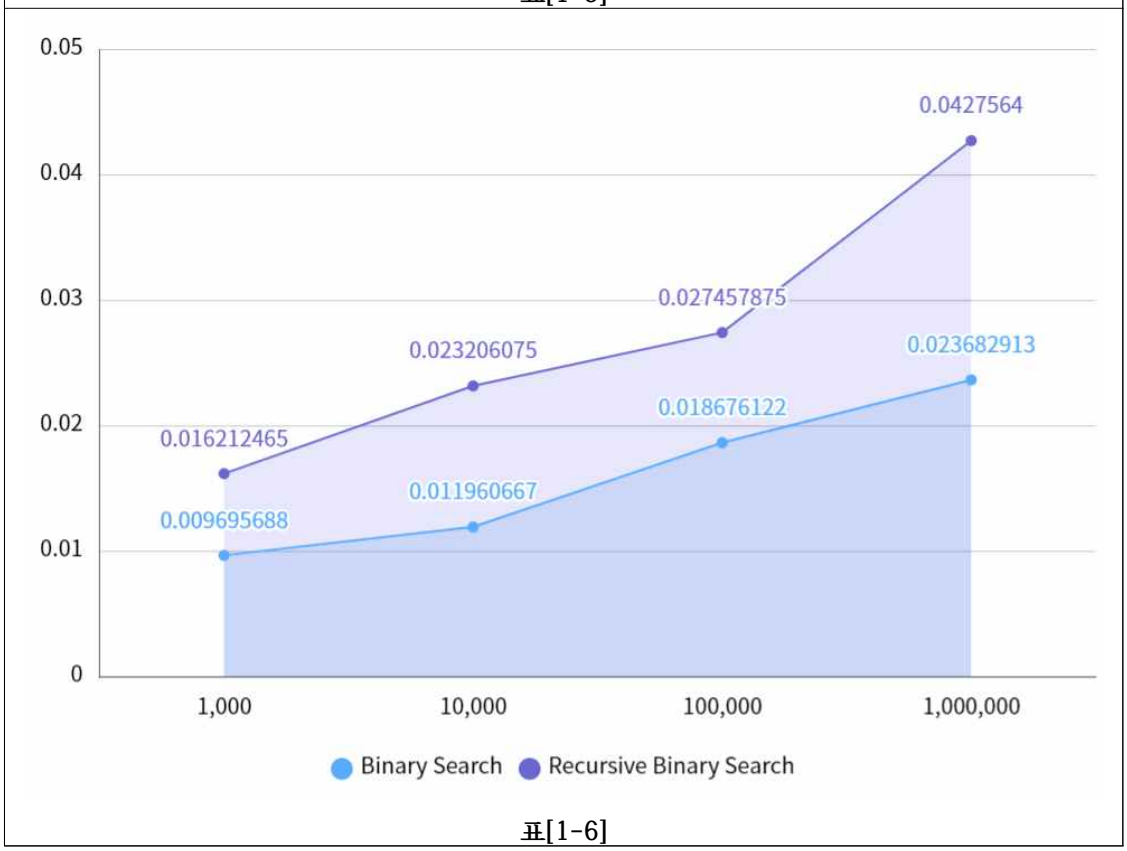
표[1-3]


150
100
50
0
100.2125343
6.732781725
0.08980433
0.798304875
1,000
10,000
100,000
1,000,000
Sequential Search

표[1-4]


1,000
10,000
100,000
1,000,000
0.009695688
0.016212465
0.011960667
0.023206075
0.018676122
0.027457875
0.023682913
0.0427564
0
0.01
0.02
0.03
0.04
0.05
Binary Search
Recursive Binary Search

표[1-5]


0.05
0.04
0.03
0.02
0.01
0
0.0427564
0.027457875
0.023206075
0.023682913
0.018676122
0.016212465
0.011960667
0.009695688
1,000
10,000
100,000
1,000,000
Binary Search
Recursive Binary Search

표[1-6]

5. 알고리즘별 수행시간에 대한 비교 및 기술적 고찰

num이 1,000일 때, 수행시간의 정확성을 높이기 위해 총 6번의 수행 결과, 순차검색 알고리즘의 경우 수행시간 평균이 0.08980433(ms)가 나왔고, 이진검색 알고리즘의 경우 수행시간 평균이 0.009695688(ms), 재귀적 이진검색 알고리즘의 경우 수행시 평균이 0.016212465(ms)가 나왔다. num이 1,000일 때, 세 알고리즘의 수행시간이 크게 차이는 나지 않으나, 그나마 가장 수행시간이 높은 알고리즘은 순차검색 알고리즘이다.

그러나 순차검색 알고리즘의 경우, 표[1-3], 표[1-4]를 보면 알 수 있듯이, 평균 수행시간이 num이 10,000일 때는 0.798304875(ms), num이 100,000일 때는 6.732781725(ms), num이 1,000,000일 때는 100.2125343(ms)으로, num이 10배가 늘어날 때마다 수행시간도 대략 10배씩 같이 증가하는 것을 알 수 있다. 따라서 순차검색 알고리즘은 num 값이 n배씩 늘어날 때마다 수행시간이 n만큼 동시에 늘어나므로 본 순차검색 알고리즘은 num 값과 수행시간이 비례하다는 것을 알 수 있다. 따라서 항목의 개수만큼 검색 시간이 늘어나므로 최악의 경우를 고려하였을 때, 시간복잡도는 O(n)이다.

이와 달리 이진검색 알고리즘의 경우, 표[1-6]을 통해 알 수 있듯이 num의 값이 10배씩 증가하여도 수행시간이 크게 증가하지 않는 것을 알 수 있다. 최악의 경우를 고려해보았을 때, while 문을 한번 수행할 때마다 해당 리스트의 크기가 절반으로 감소하기 때문에 시간복잡도는 O(logn)이다. 단, 이진검색은 순차검색과 달리 숫자가 오름차순으로 정렬되어있어야 하는 조건이 있다.

재귀적 이진검색 알고리즘의 경우, 기존 이진검색 알고리즘을 변형하여 재귀적 호출을 하여 해당 함수를 계속 순환해 사용할 수 있도록 만든 알고리즘이다. 재귀적 이진검색 알고리즘 또한 다시 함수가 순환 호출될 때, 해당 리스트의 크기가 계속 절반으로 감소하며 호출되기 때문에 기존 반복으로 구현한 이진검색 알고리즘과 마찬가지로 시간복잡도는 O(logn)이다.

표[1-2]를 통해 전체적으로 세 알고리즘을 비교하면, 이진검색 및 재귀적 이진검색 알고리즘의 수행시간은 큰 변화는 없으나, 순차검색 알고리즘은 도드라지게 num이 증가할수록 수행시간이 빠르게 늘어나는 것을 알 수 있다. 따라서 이진검색 알고리즘이 가장 효율적이라고 볼 수 있지만, 두 알고리즘의 리스트 항목의 정렬 여부가 다르므로 무조건 이진검색 알고리즘이 가장 효율적이라고 볼 수는 없다. 실제로 정렬되어있지 않은 상태에서는 순차검색 알고리즘보다 빠른 알고리즘을 찾을 수 없기 때문이다.

문제 2. 피보나치수열

1. 프로그램 코드

```
import time

def main():
    num = 10 #num값을 변경하여 수행시간 비교 및 분석

    start = time.time()
    result1 = iterative_fibonacci(num)
    end = time.time()
    #피보나치수열에 대한 반복 알고리즘 수행시간 측정

    print("[Iterative Fibonacci]")
    print("Num {0} : Fibonacci Number {1}".format(num, result1))
    print("Elapsed Time: {0:08f}s".format((end - start)))
    print()
    #반복 알고리즘의 num 값과 num 번째 해당하는 피보나치수열의 수 및 수행시간 출력

    start = time.time()
    result2 = recursive_fibonacci(num)
    end = time.time()
    #피보나치수열에 대한 재귀 알고리즘 수행시간 측정

    print("[Recursive Fibonacci]")
    print("Num {0} : Fibonacci Number {1}".format(num, result2))
    print("Elapsed Time: {0:08f}s".format((end - start)))
    print()
    #재귀 알고리즘의 num 값과 num 번째 해당하는 피보나치수열의 수 및 수행시간 출력

def iterative_fibonacci(num): #피보나치수열에 대한 반복 알고리즘 함수
    fibo = list(0 for i in range(0, num + 1))
    if num > 0:
      fibo[1] = 1
      for i in range(2, num + 1):
        fibo[i] = fibo[i-1] + fibo[i-2]
    return fibo[num]

def recursive_fibonacci(num): #피보나치수열에 대한 재귀 알고리즘 함수
    if num <= 1:
```

```
        return num
    else:
        return recursive_fibonacci(num - 1) + recursive_fibonacci(num - 2)

if __name__ == "__main__":
    main()
```

2. 프로그램 코드 설명

피보나치수열에 대한 반복 알고리즘 함수
- 설정한 num 값을 해당 함수 매개변수로 받는다. - 0번째부터 num번째까지의 피보나치 수를 구하기 위해 리스트 fibo를 0부터 num까지의 리스트 크기로 for 문을 사용해 선언하여 값을 0으로 모두 초기화한다. - 만약 num 값이 0보다 클 경우, fibo의 num이 1인 1번째 값에는 1을 넣는다. - 그리고 for 문을 사용하여 2부터 num까지 반복하며, fibo의 i번째 값에 리스트 fibo의 (i-1)번째와 (i-2)번째의 값을 더한 값을 넣는다. - 해당 for 문이 종료되면 if 문도 빠져나와 리스트 fibo의 num번째 값을 반환한다.
피보나치수열에 대한 재귀 알고리즘 함수
- 설정한 num 값을 해당 함수 매개변수로 받는다. - 만약 num 값이 1보다 작거나 같은 경우, 즉 0과 1일 경우에는 num 그대로 반환한다. - 그렇지 않고, 만약 num 값이 1보다 큰 경우, 해당 재귀 알고리즘 함수를 다시 호출하는데, 이때 피보나치수열의 수를 도출하기 위해 각 (num-1) 번째와 (num-2) 번째 값을 매개변수로 넣은 두 함수를 더한 값을 반환하도록 한다.

3. 프로그램 수행화면

num = 10

```
[Iterative Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time: 0.000012s

[Recursive Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time: 0.000046s
```

num = 15

```
[Iterative Fibonacci]
Num 15 : Fibonacci Number 610
Elapsed Time: 0.000011s

[Recursive Fibonacci]
Num 15 : Fibonacci Number 610
Elapsed Time: 0.000283s
```

num = 20

```
[Iterative Fibonacci]
Num 20 : Fibonacci Number 6765
Elapsed Time: 0.000016s

[Recursive Fibonacci]
Num 20 : Fibonacci Number 6765
Elapsed Time: 0.003105s
```

num = 25

```
[Iterative Fibonacci]
Num 25 : Fibonacci Number 75025
Elapsed Time: 0.000012s

[Recursive Fibonacci]
Num 25 : Fibonacci Number 75025
Elapsed Time: 0.031691s
```

num = 30

```
[Iterative Fibonacci]
Num 30 : Fibonacci Number 832040
Elapsed Time: 0.000015s

[Recursive Fibonacci]
Num 30 : Fibonacci Number 832040
Elapsed Time: 0.372307s
```

num = 35

```
[Iterative Fibonacci]
Num 35 : Fibonacci Number 9227465
Elapsed Time: 0.000016s

[Recursive Fibonacci]
Num 35 : Fibonacci Number 9227465
Elapsed Time: 4.078217s
```

num = 40

```
[Iterative Fibonacci]
Num 40 : Fibonacci Number 102334155
Elapsed Time: 0.000018s

[Recursive Fibonacci]
Num 40 : Fibonacci Number 102334155
Elapsed Time: 44.630917s
```

num = 45

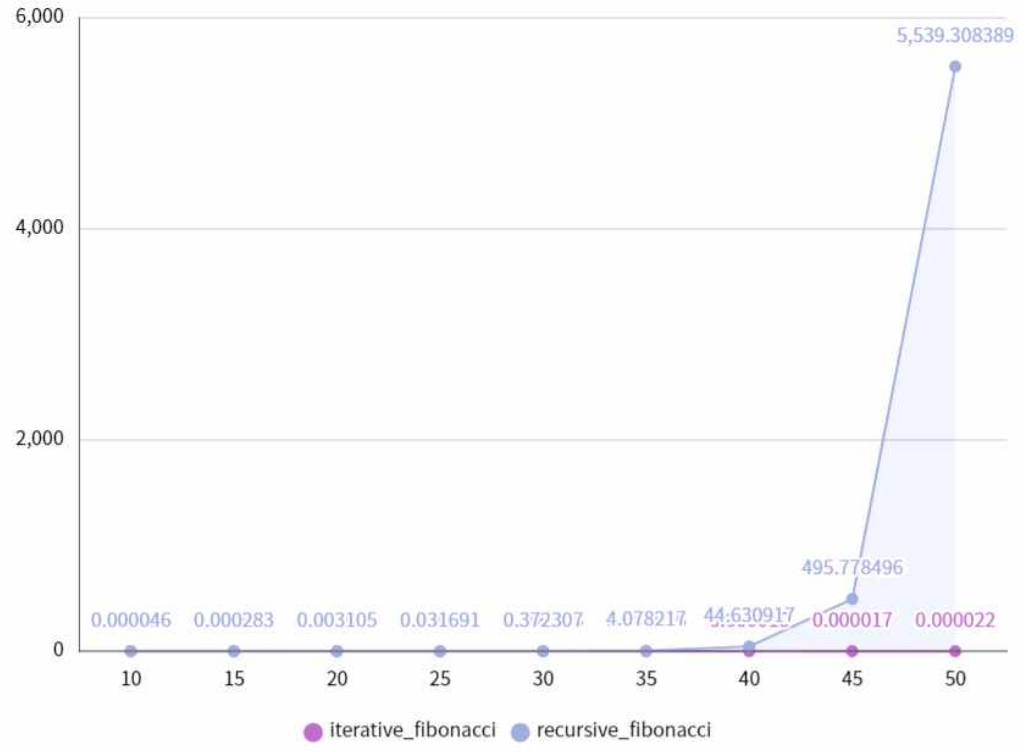
```
[Iterative Fibonacci]
Num 45 : Fibonacci Number 1134903170
Elapsed Time: 0.000017s

[Recursive Fibonacci]
Num 45 : Fibonacci Number 1134903170
Elapsed Time: 495.778496s
```

num = 50

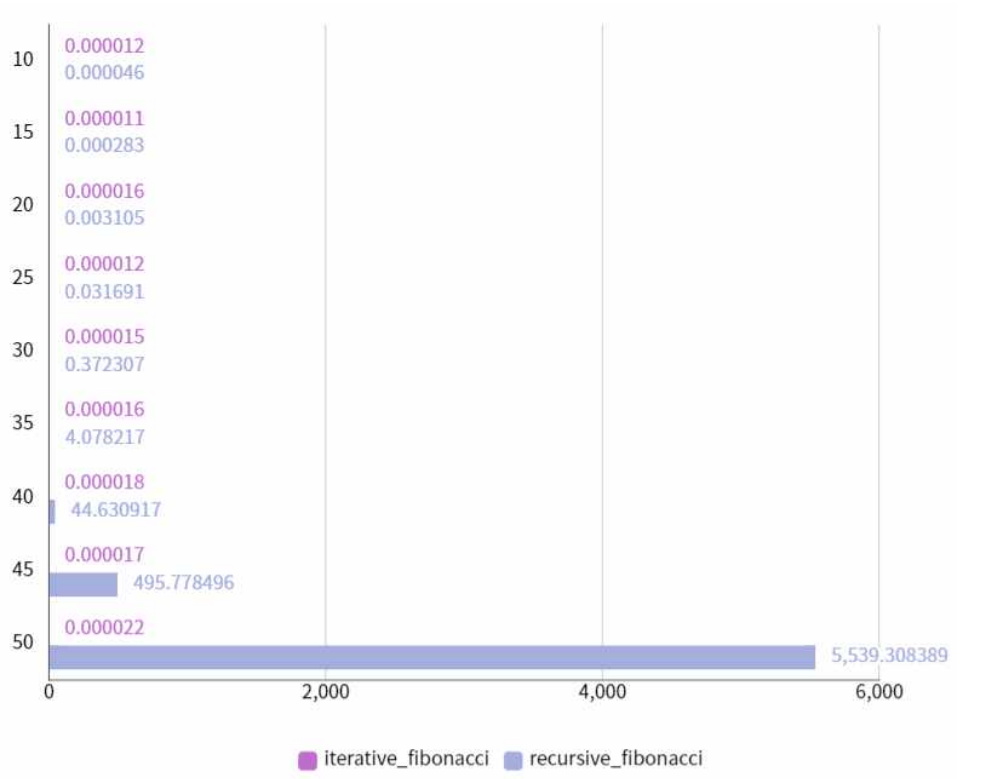
```
[Iterative Fibonacci]
Num 50 : Fibonacci Number 12586269025
Elapsed Time: 0.000022s

[Recursive Fibonacci]
Num 50 : Fibonacci Number 12586269025
Elapsed Time: 5539.308389s
```

4. 수행시간 그래프

전체 그래프

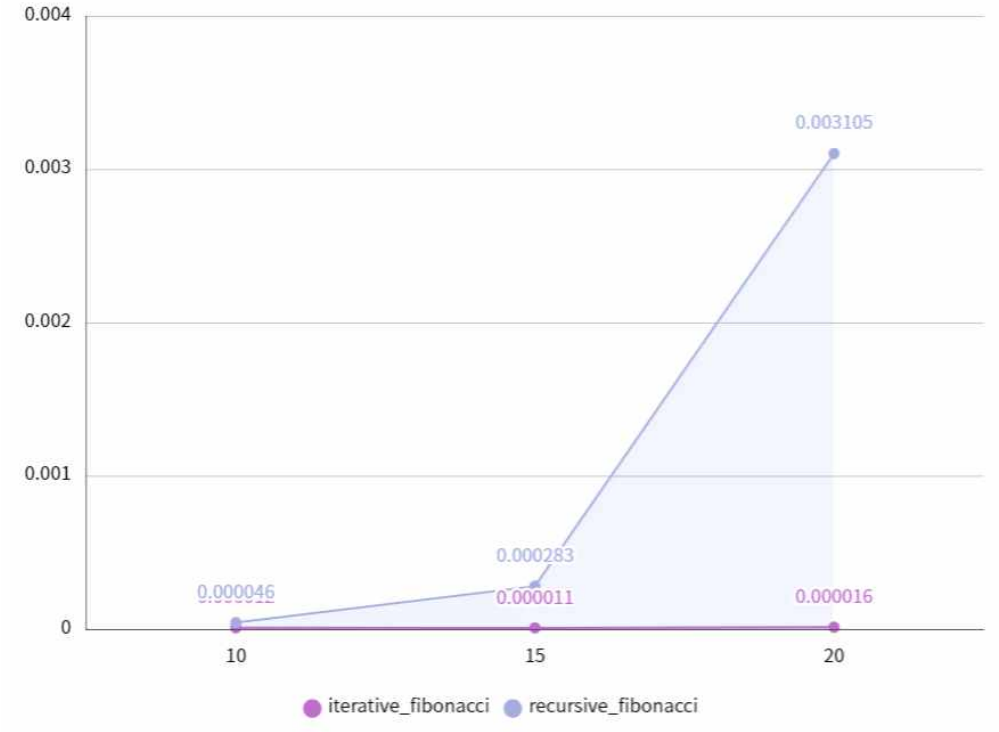


표[2-1]

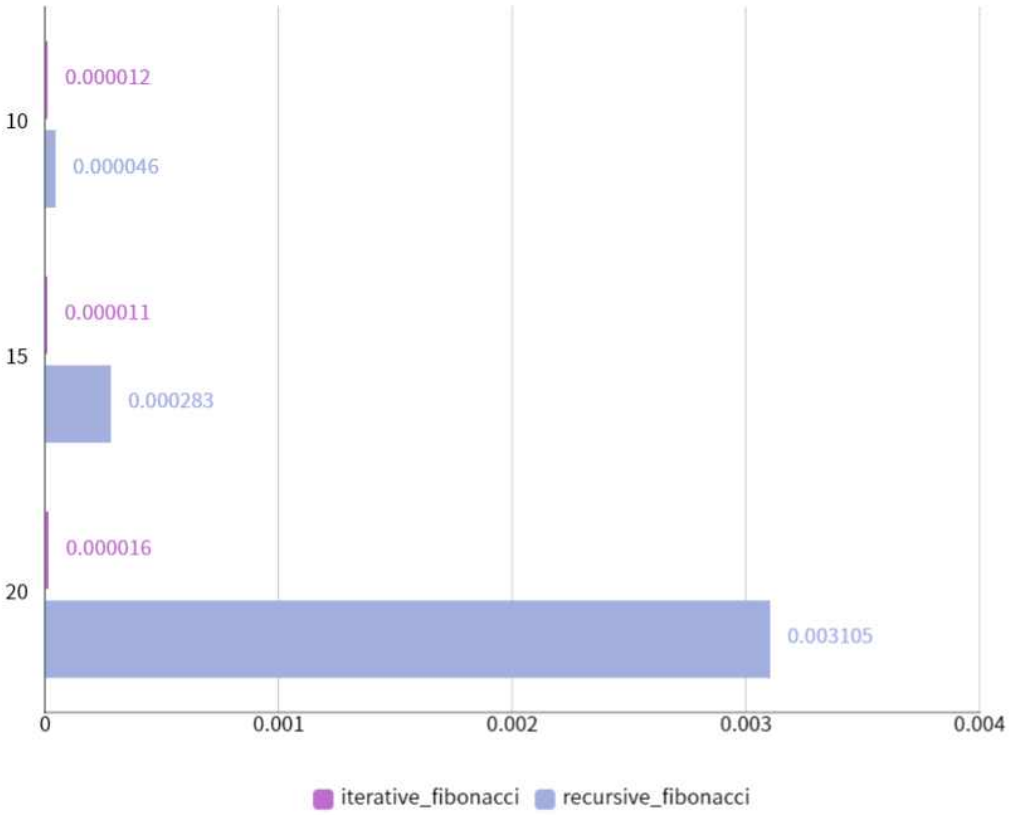


표[2-2]

num = 10, 15, 20일 때

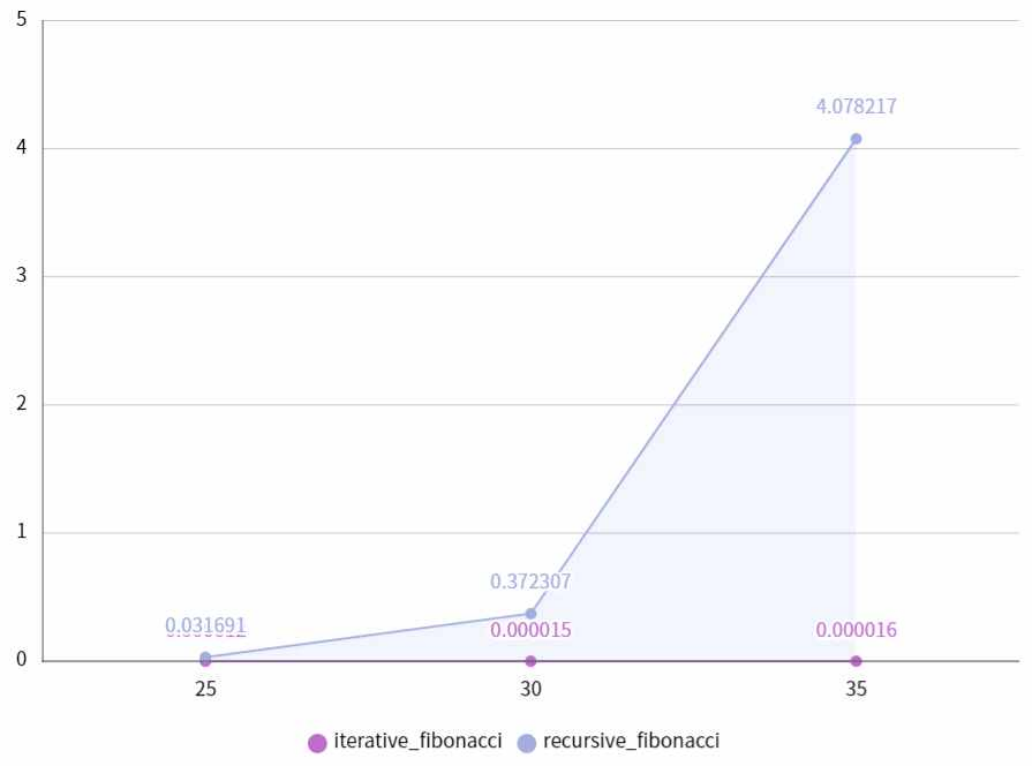


표[2-3]

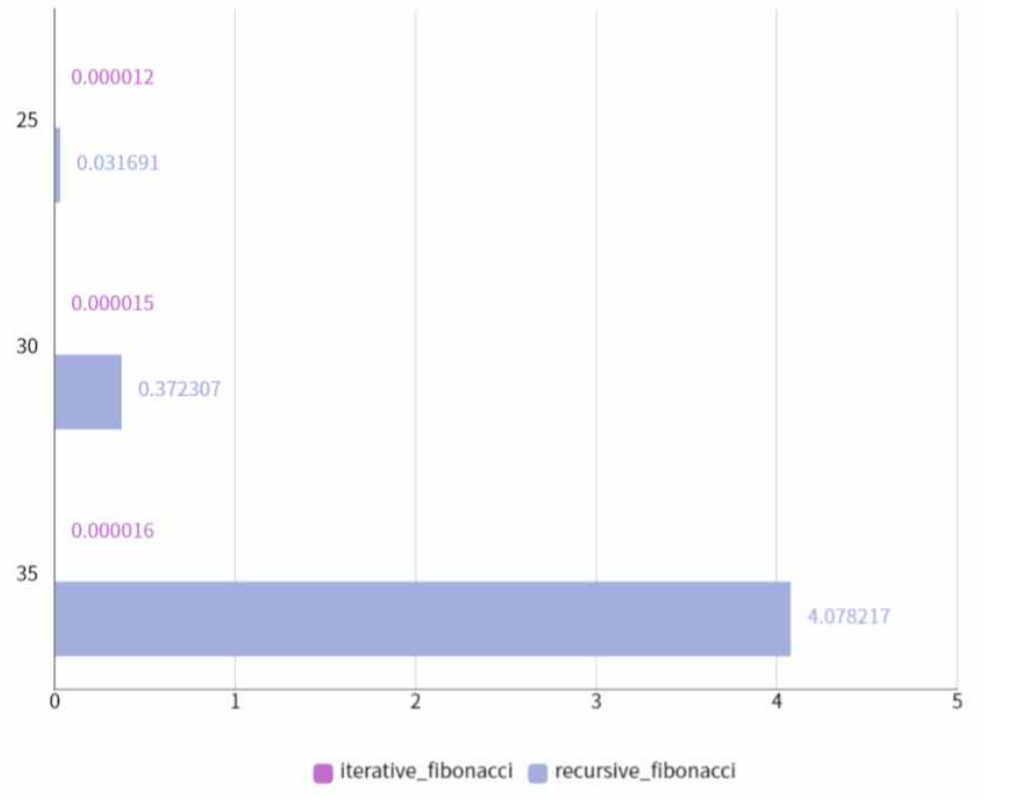


표[2-4]

num = 25, 30, 35일 때

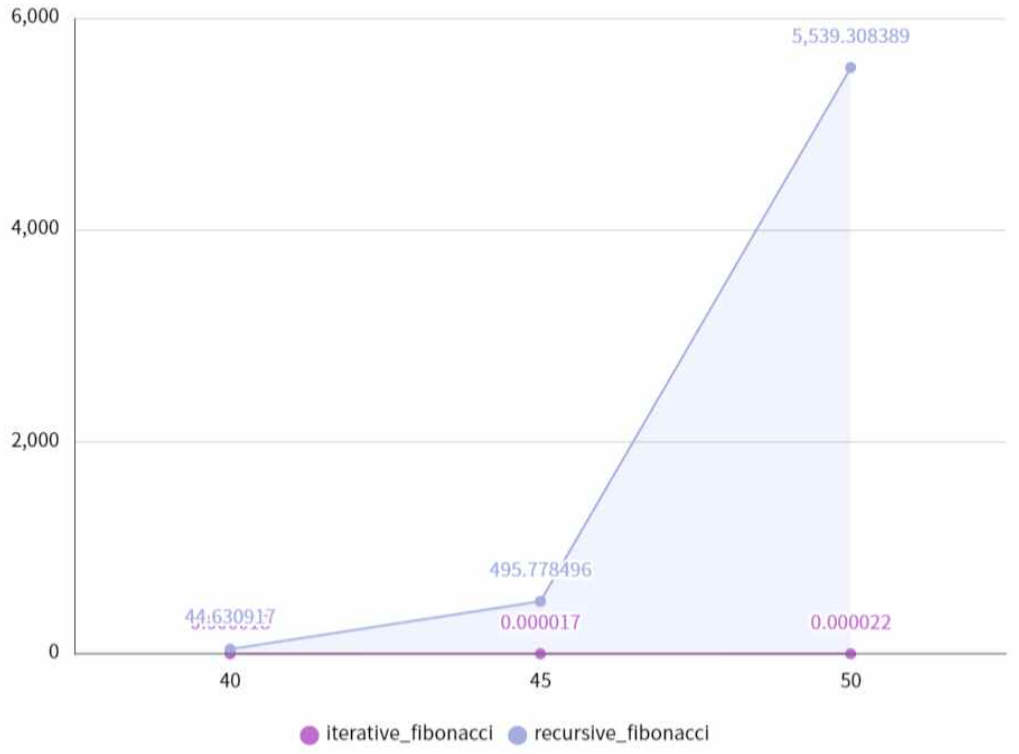


표[2-5]

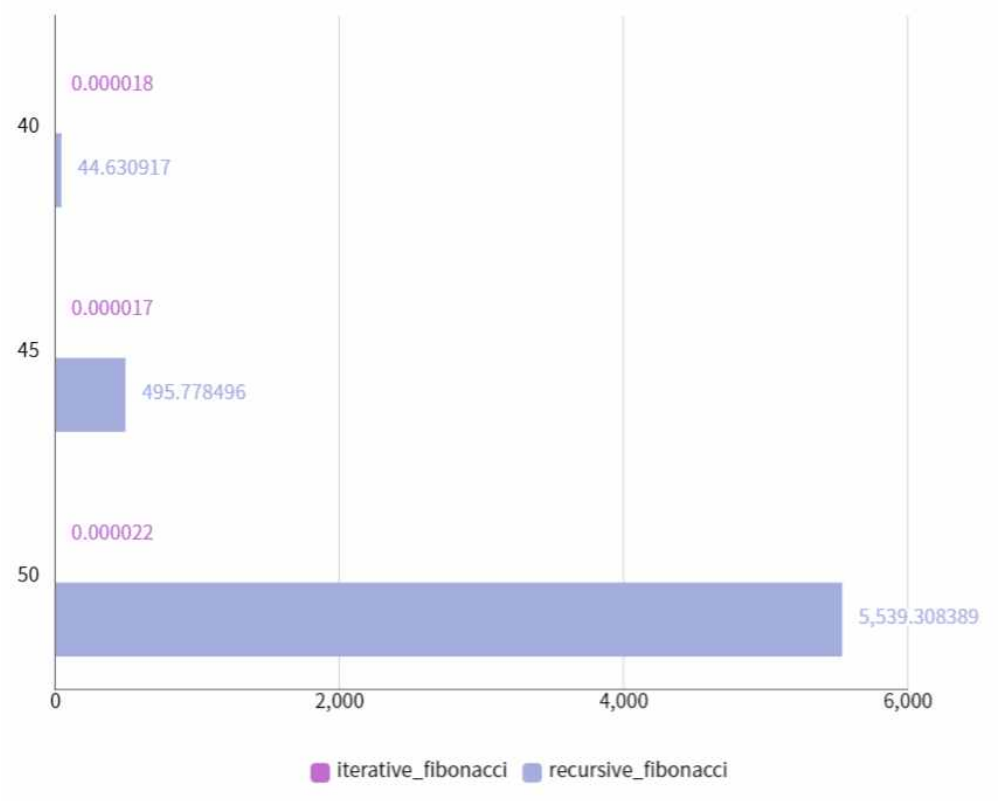


표[2-6]

num = 40, 45, 50일 때

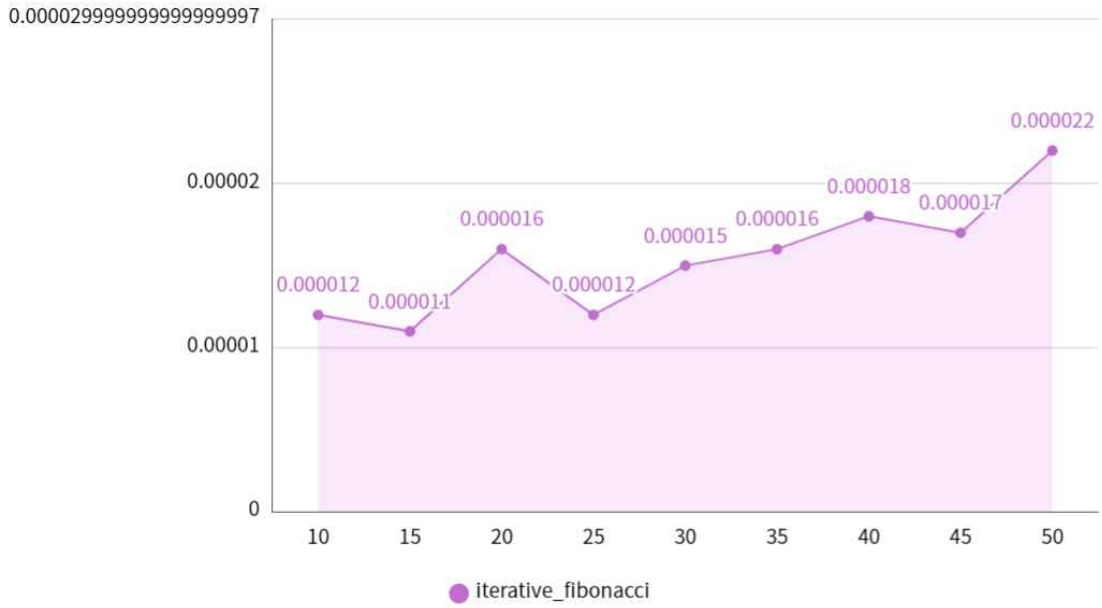


표[2-7]

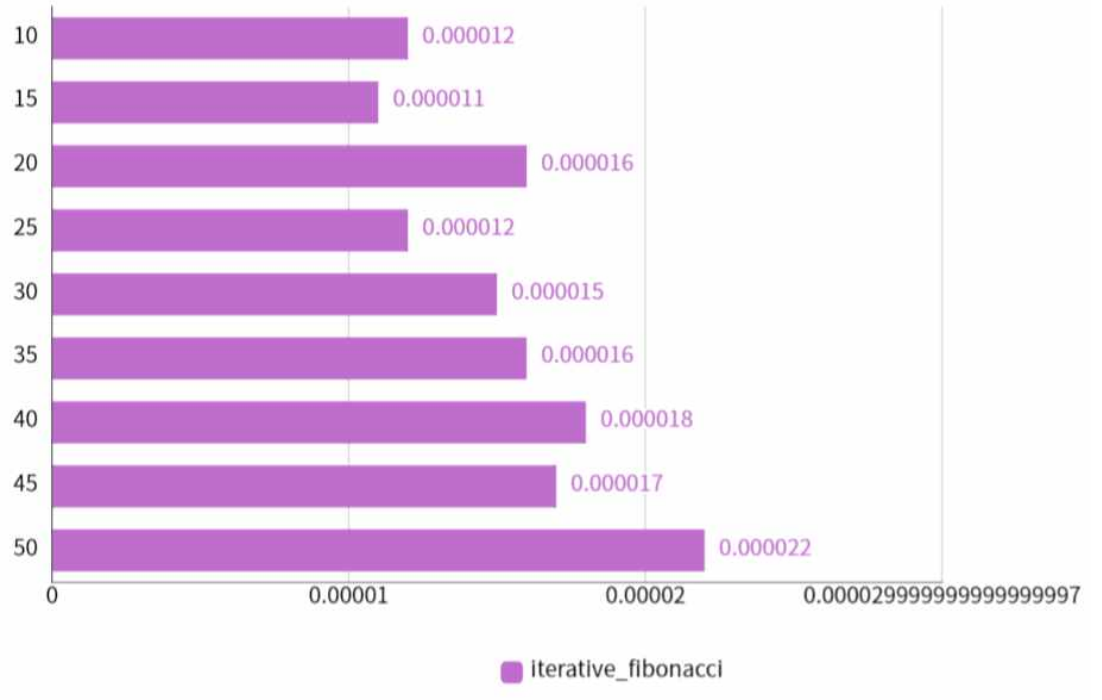


표[2-8]

Iterative_fibonacci

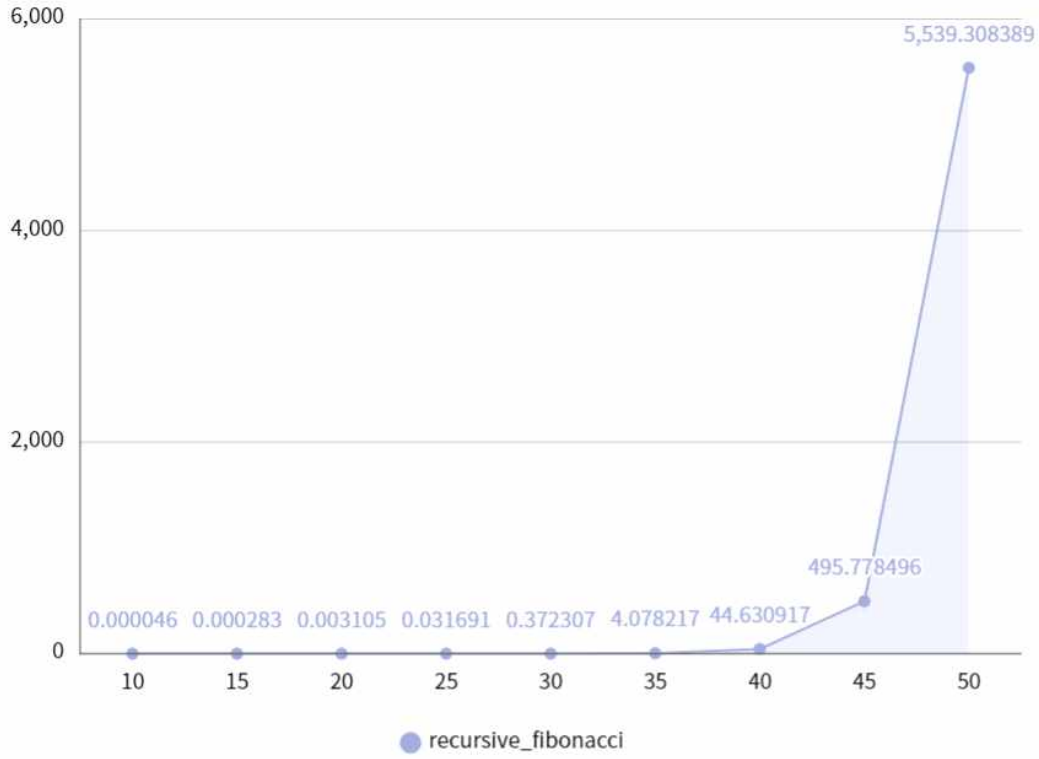


표[2-9]

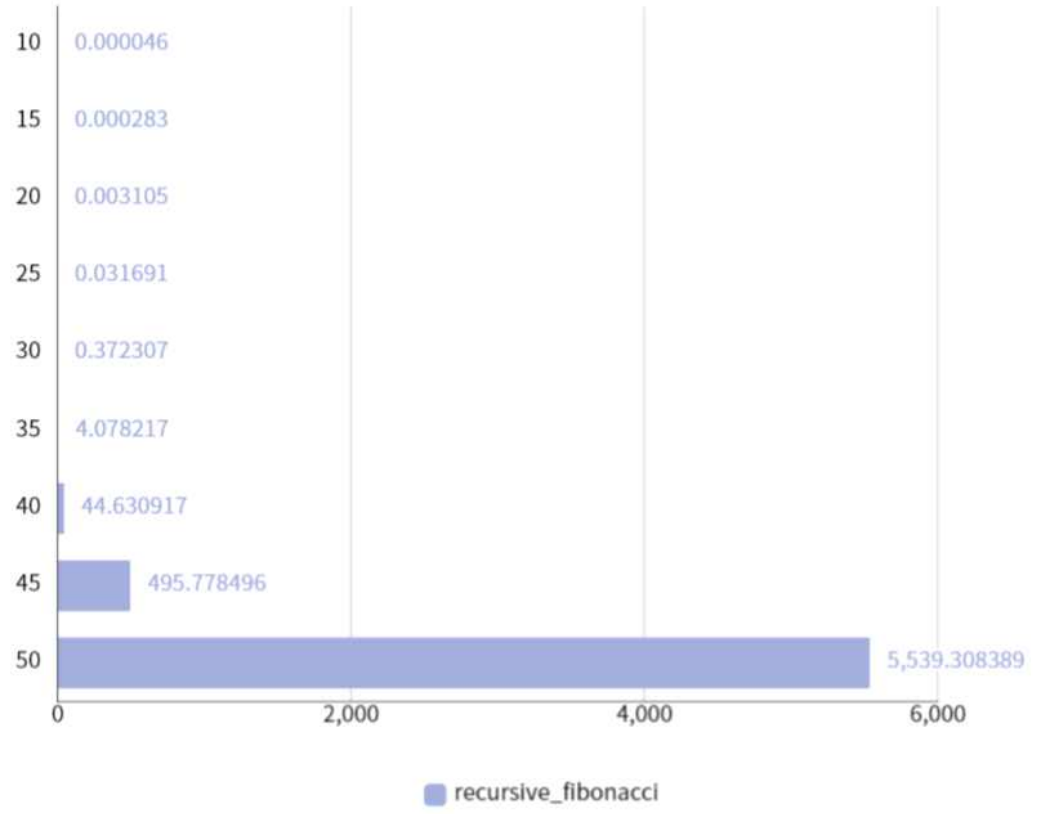


표[2-10]

Recursive_fibonacci



표[2-11]

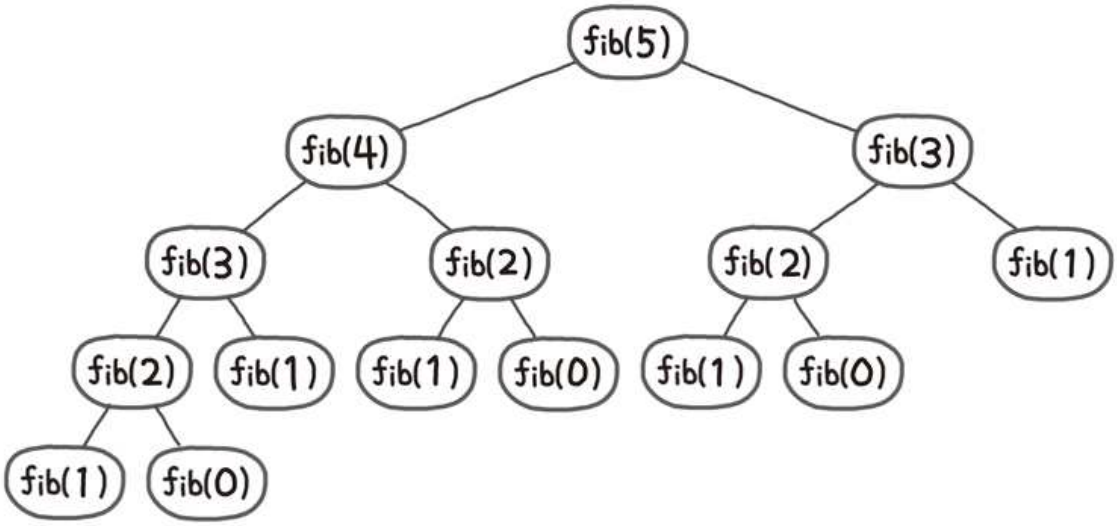


표[2-12]

5. 알고리즘별 수행시간에 대한 비교 및 기술적 고찰

표[2-1]와 표[2-2]를 통해 두 함수를 비교해보면 재귀 알고리즘의 수행시간 증가율에 의해 반복 알고리즘의 그래프 변화가 거의 없는 것처럼 나타났다. 이를 통해 num이 증가할수록 재귀 알고리즘의 수행시간이 훨씬 빠르게 증가한다는 것을 알 수 있다. 본 그래프로는 반복 알고리즘의 수행시간 변화율을 쉽게 측정하기 어려워 표[2-9]와 표[2-10]을 통해 num에 대한 반복 알고리즘의 수행시간 그래프를 따로 추출하였다. 표[2-9] 그래프를 살펴보면, num이 증가할수록 수행시간이 조금씩 증가하는 것을 알 수 있다. 그러나 재귀 알고리즘에 비해 num 값의 증가율이 수행시간 변화에 크게 영향을 미치지 않았다. 두 알고리즘의 시간복잡도에 대해 정리해보자면 다음과 같다.

피보나치수열에 대한 재귀 알고리즘의 경우, num에 대한 피보나치수를 구하기 위해서는 해당 함수가 recursive_fibonacci(num)이라고 했을 때, recursive_fibonacci(num-1)과 recursive_fibonacci(num-2)를 모두 알아야 한다. 다시 말해, 한 번 해당 함수를 호출하면 두 번의 함수를 다시 호출하게 된다.



[그림1][1)]

따라서 위 [그림1]과 같은 형태로 순환 호출을 하게 되므로 피보나치 수열을 계산하는 재귀 알고리즘의 시간복잡도는 O(2^n)이 된다.

피보나치수열에 대한 반복 알고리즘의 경우, for 문을 통해 반복하는 알고리즘이므로 fibo[0]부터 fibo[num]까지 한 번씩만 계산하므로, 즉 n에 대한 반복이 n번만 일어나므로 해당 알고리즘의 시간복잡도는 O(n)인 것을 알 수 있다. 이러한 반복 알고리즘은 중복되는 계산이 없기 때문에 재귀 알고리즘보다 훨씬 더 수행시간이 빠르다는 것을 알 수 있다.

---

1) 그림 출처: https://velog.io/@kongyb/210824-25-CodeStates-27-28%EC%9D%BC%EC%B0%A8

결론

지난 학기 ‘자료구조및실습’ 수업에서 순차 탐색 및 이진탐색 알고리즘과 피보나치수열에 대한 반복 및 재귀 알고리즘을 배웠었는데, 이번 학기 ‘알고리즘및실습’ 수업을 통해 그래프를 그려보면서 다시 복습할 수 있어서 좋았다. 그러나 이번 과제를 표지, 서론, 본론, 결론에 맞추어 작성하려다 보니 평소보다 더 과제에 시간 투자를 많이 하게 되는 느낌이 들었다. 어떻게 보면 내용 측면에서는 더 기억에 오래 남을 수 있긴 하지만, 약간은 부담되기도 했다. 그래도 보고서를 잘 작성해서 파일에 잘 보관해두면 나중에 알고리즘 관련 공부를 하거나 자소서를 준비할 때 충분히 도움 될 것 같다.