알고리즘및실습

[실습 및 과제 1]

과목명/분반	알고리즘및실습/02	담당교수	한연희
학 부 명	컴퓨터공학부	제 출 일	2022/ 03 / 20
학번	2018136121	이름	조원석

INDEX

서론 : Homework의 내용 및 목적

▶ 내용

: search.py과 fibonacci.py 코드상에 정의된 순차검색 함수를 구현하고 이진검색과 피보나치를 각각 반복문과 재귀를 이용해서 구현하는 방법 학습하기

: 재귀함수를 이용한 문제 풀이 중 이진검색을 할 때 사용하는 분할정복과 피보나치 함수를 사용할 때 사용하는 일반 재귀의 차이에 대해 인식하고 각각의 함수가 반복문으로 구현할 때와 어떠한 차이가 있는지 관찰하기

▶ 표면적인 범위에서의 목적

: 검색 알고리즘의 이해 및 효율성 분석

: 반복(DP)와 재귀를 통한 피보나치 수열 탐색

: **알고리즘 효율 분석 방법 기초** 이해

▶범용적인 범위에서의 목적

: 반복과 재귀과 관련된 알고리즘 지식 습득

: 체계적인 사고 훈련의 밑거름

: 문제 코드상의 함수를 구현하며 지적 추상화 레벨 높이기

본론 : 문제 풀이

1> 문제 1. 순차검색과 이진검색

-search.py 분석하기

▶ 내용

: search.py는 수행횟수가 num이고 그 범위가 각각 [0,num]인 랜덤 리스트 s와 검색대상이 되는 key를 이용해서 순차검색을, 그리고 s를 정렬하여 반복과 재귀를 통해 이진검색을 진행합니다.

: 여기서 순차검색과 반복, 재귀로 구현된 이진검색을 진행하기 위해서 함수를 완성하고, 각각의 함수 수행시간을 분석하여 그래프로 그린 후 비교하는 과정을 이번 문제에서 알아보고자 합니다.

-완성되지 않은 세 함수 완성하기

▶ *def sequential_search(s, key):*
#순차검색

▶ *def binary_search(s, key):*
#이진검색(반복_분할정복)

▶ *def recursive_binary_search(s, key, low, high);*
#이진검색(재귀_분할정복)

▶ def sequential_search(s, key):

```
#과정 1. 초기화
# i를 1로 설정한 이유_ IndexError : list index out of range 해결을 위해
num = len(s)                                    # num = s의 길이
location = 0                                    # 초기 위치를 0으로 지정
i = 1                                           # i == 검색 위치
--------------------------------------------------------------------------------
#과정 2. 순차검색 반복문 실행
#key 값이 존재할 경우 위치(i)를 location에 대입 후 반복 탈출
# 0부터 배열의 끝까지 key 값에 대한 순차검색 실행
for i in s:
        if key == s[i-1]:                       #과정 3-1. 순차검색 성공한 경우
                location = i-1
                break
--------------------------------------------------------------------------------
#과정 3-2. 순차검색 실패한 경우
# [주의 및 수행 사항 2번 : location에 변화가 없다면 -1 리턴]
if location == 0:
        location = -1
--------------------------------------------------------------------------------
#과정 4. 최종 결과 리턴
return location
```

def sequential_search(s, key):
즉, 순차검색은 위와 같이
1. 초기화
2. 순차검색 반복문 실행
3. 순차검색을 성공하고 실패한 경우
4. 최종 결과 리턴
을 통해 이루어집니다.

알고리즘의 수행시간 같은 경우에는 '과정 2'의 반복문을 통해 **'과정 3-1'이 핵심연산으로 수행되면서 최악의 경우 num번의 검색이 요구됩니다.**
평균적으로는 (num+1)/2번의 검색이 요구됩니다.

▶ def binary_search(s, key):

```
#과정 1. 초기화
# [주의 및 수행 사항 2번 : location에 변화가 없다면 -1 리턴]
num = len(s)                                    # num = s의 길이
low = 0
high = num - 1
location = -1
-------------------------------------------------------------------------------
#과정 2. 이진검색 반복문 실행
while True:
-------------------------------------------------------------------------------
    #과정 3. 탈출조건 정의 ( low가 high보다 커질 때 )
    if low > high: break
-------------------------------------------------------------------------------
    #과정 4. 검색범위 지정
    mid = round((high + low) / 2)
    #과정 4-1. 같을 때
    if key == s[mid]:
        location = mid
        break
    #과정 4-2. key가 더 클 때
    elif key > s[mid]: low = mid + 1
    #과정 4-3. key가 더 작을 때
    else: high = mid– 1
-------------------------------------------------------------------------------
#과정 5. 최종결과 리턴
return location
```

def sequential_search(s, key): 즉 이진검색(반복_분할정복)같은 경우 위와 같은 형식으로 진행되며 알고리즘의 수행시간 같은 경우에는 **'과정 4'에서 진행된 연산들이 핵심연산으로 수행되면서 최악의 경우** $\log_2 num + 1$**번의 검색이 요구됩니다.**

▶ def recursive_binary_search(s, key, low, high):

```
#과정 1. 탈출조건 정의 ( low가 high보다 커질 때 )
# [주의 및 수행 사항 2번 : location에 변화가 없다면 –1 리턴]
if(low > high): return -1
--------------------------------------------------------------------------------
#과정 2. 검색범위 지정
mid = round((low + high) / 2)
#과정 2-1. 같을 때
if key == s[mid]:
        return mid
#과정 2-1. key가 더 클 때
elif key > s[mid]:
    return recursive_binary_search(s, key, mid + 1, high)
#과정 2-3. key가 더 작을 때
else:
    return recursive_binary_search(s, key, low, mid - 1)
```

def recursive_binary_search(s, key, low, high): 즉 이진검색(재귀_분할정복) 같은 경우 위와 같은 형식으로 진행되며 알고리즘의 수행시간 같은 경우에는 **'과정 4'에서 진행된 연산들이 핵심연산으로 수행되면서 최악의 경우** $\log_2 num + 1$**번의 검색이 요구됩니다.**

분할정복이란?

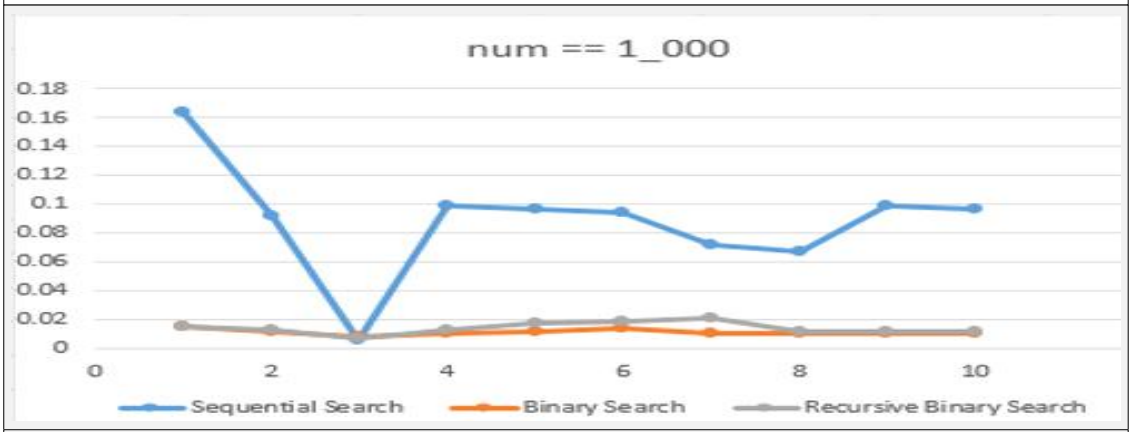
>그대로 해결할 수 없는 문제를 작은 문제로 분할하여 문제를 해결하는 방법이나 알고리즘

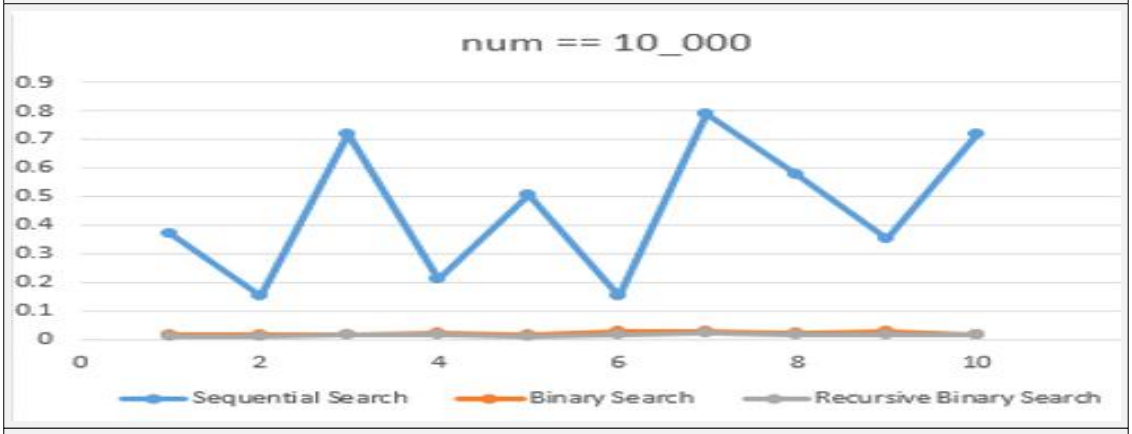
ex)

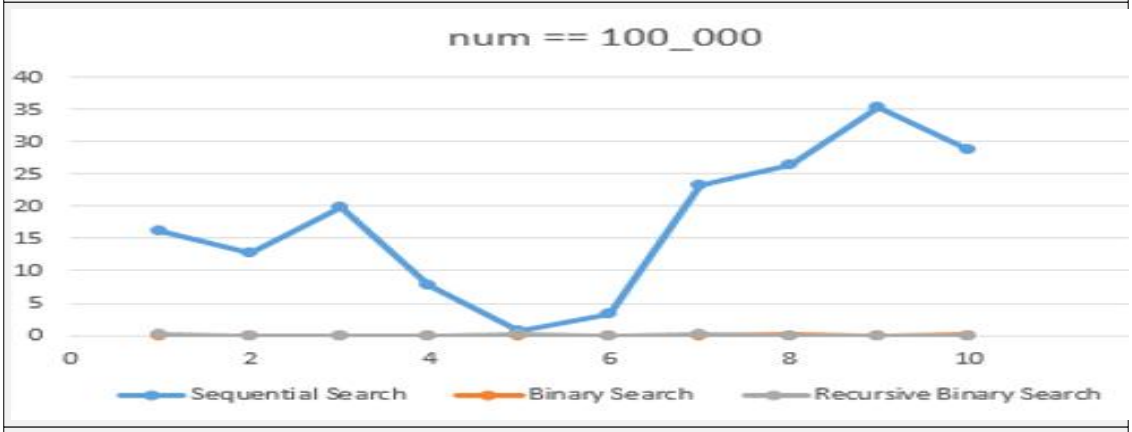
분할 - mid를 기준으로 왼쪽, 오른쪽으로 분할, key와 s[mid]가 같다면, mid 값을 반환한다.

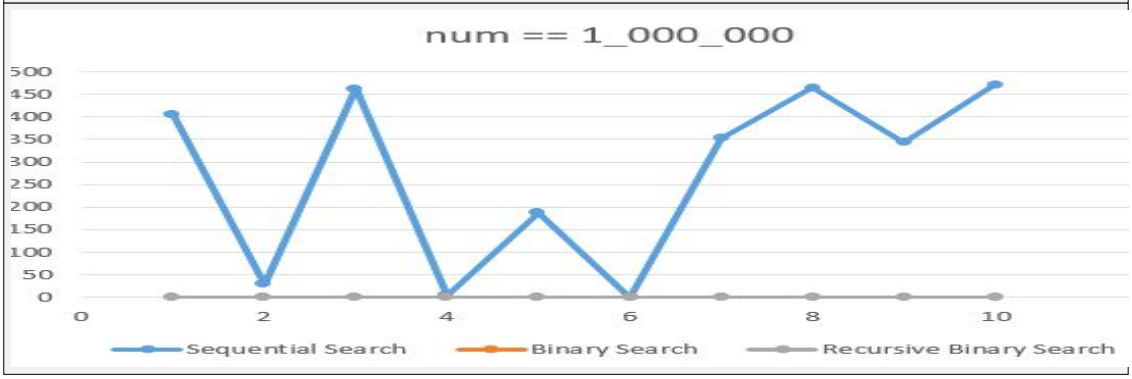
정복 – mid가 s[mid]보다 크거나 작을 때를 기준으로 각각 우측 검색, 좌측 검색을 진행한다.

▶ 그래프를 통해 확인하기

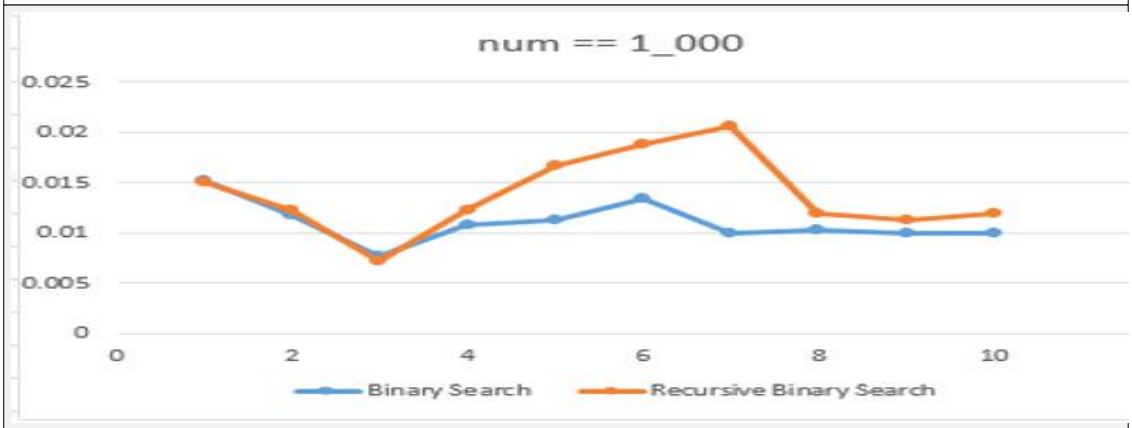

num == 1_000
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
2
4
6
8
10
Sequential Search
Binary Search
Recursive Binary Search


num == 10_000
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
Sequential Search
Binary Search
Recursive Binary Search

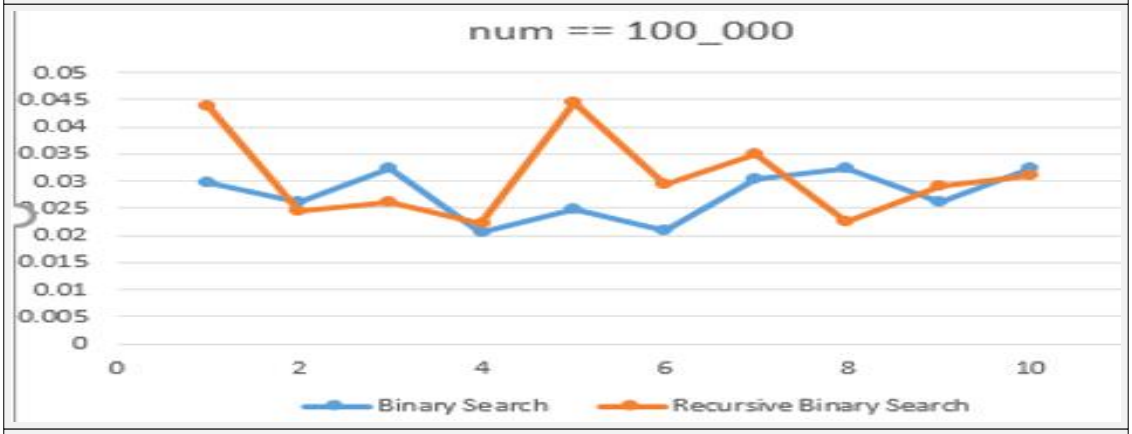

num == 100_000
40
35
30
25
20
15
10
5
0
0
2
4
6
8
10
Sequential Search
Binary Search
Recursive Binary Search

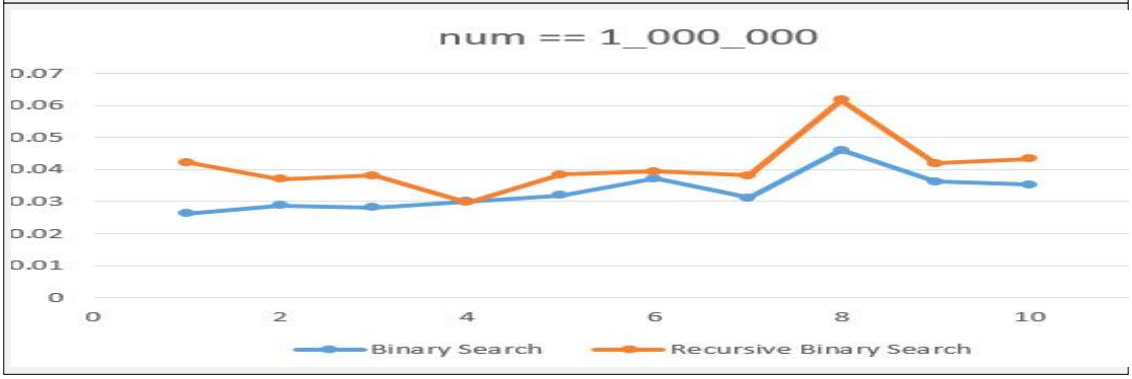

num == 1_000_000
500
450
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
Sequential Search
Binary Search
Recursive Binary Search

▶ 그래프를 통해 확인하기


num == 1_000
0.025
0.02
0.015
0.01
0.005
0
0
2
4
6
8
10
Binary Search
Recursive Binary Search


num == 10_000
0.03
0.025
0.02
0.015
0.01
0.005
0
0
2
4
6
8
10
Binary Search
Recursive Binary Search


num == 100_000
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
2
4
6
8
10
Binary Search
Recursive Binary Search


num == 1_000_000
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
2
4
6
8
10
Binary Search
Recursive Binary Search

▶ 그래프를 통해 확인하기

COLAB이라는 컴파일러를 통해 각각의 수행시간을 num당 10회씩 비교해본 결과 다음과 같은 그래프들이 나타났습니다.

알고리즘의 수행시간은 num이 증가함에 따라
각각 **순차검색 > 이진검색(재귀) >= 이진검색(반복)** 순으로 높게 나타났고, **수행횟수가 커질수록 순차검색의 효율이 가장 좋지 않았음을 눈으로 확인해볼 수 있었습니다**

그리고 분할정복이라는 같은 알고리즘을 사용하였지만, 이번 문제에서 평균적으로 **반복으로 구현하는 것보다 재귀로 구현했을 때 수행시간이 더 짧은 것**을 확인해볼 수 있었습니다.

각각의 그래프는 엑셀을 통해 나타냈고 각각의 수행횟수에 따른 수행시간의 자세한 값은 확인 용도로 보고서와 함께 추가로 첨부하겠습니다.

▶ 수행결과 모음

num == 1_000

```
[Sequential Search Result]
Key value 467: location -1
Elapsed Time: 0.16307831ms

[Binary Search Result]
Key value 467: location 462
Elapsed Time: 0.01525879ms

[Recursive Binary Search Result]
Key value 467: location 462
Elapsed Time: 0.01502037ms
```

	Sequential Search	Binary Search	Recursive Binary Search
1	0.16307831	0.01525879	0.01502037
2	0.09179115	0.01168251	0.01215935
3	0.00596046	0.00762939	0.00715256
4	0.09870529	0.01072884	0.01215935
5	0.09679794	0.01120567	0.0166893
6	0.0936985	0.01335144	0.01883507
7	0.07104874	0.01001358	0.020504
8	0.06628036	0.010252	0.01192093
9	0.09822845	0.01001358	0.01120567
10	0.09608269	0.01001358	0.01192093
	0.088167189	0.011014938	0.013756753

num == 1_000_000

```
[Sequential Search Result]
Key value 183858: location -1
Elapsed Time: 318.95089149ms

[Binary Search Result]
Key value 183858: location -1
Elapsed Time: 0.03743172ms

[Recursive Binary Search Result]
Key value 183858: location -1
Elapsed Time: 0.03981590ms
```

	Sequential Search	Binary Search	Recursive Binary Search
1	404.8936367	0.02622604	0.04220009
2	29.73437309	0.02884865	0.03695488
3	462.6457691	0.02813339	0.03814697
4	4.72307205	0.03004074	0.02980232
5	187.3135567	0.03194809	0.03838539
6	0.041008	0.0371933	0.03933907
7	352.3693085	0.03123283	0.03814697
8	464.1151428	0.04601479	0.06175041
9	343.1603909	0.03623962	0.04196167
10	471.5743065	0.03528595	0.04339218
	272.0570564	0.03311634	0.041007995

2> 문제 2. 피보나치 수열

-fibonacci.py 분석하기

▶ 내용

: fibonacci.py는 두 개의 함수를 이용해서 num번째 피보나치를 구하는 문제입니다.
하나는 Dynamic Programing(DP : 동적 계획법)을 이용한 반복의 형태로, 다른 하나는 일반적인 재귀를 이용해서 구현합니다.

-완성되지 않은 두 함수 완성하기

▶ def Iterative_fibonacci(num):
피보나치(반복)
▶ def recursive_fibonacci(num):
피보나치(재귀)

▶ def Iterative_fibonacci(num):

```
#과정 1. 초기화 실행
    i = 2
    s = []
    s.append(1)                    # s[0] = 1
    s.append(1)                    # s[1] = 1
--------------------------------------------------------------------------------
    #과정 2. DP를 이용한 피보나치 초기화 실행
    while i < num:
        s.append(s[i - 1] + s[i - 2])
        i += 1
--------------------------------------------------------------------------------
    #과정 3. 결과값 리턴
    return s[num-1]
```

def Iterative_fibonacci(num): 즉, 피보나치(반복)은 위와 같이 반복을 통해 이루어지게 되는데 0번째와 1번째 피보나치 수는 각각 1이므로 사전에 초기화[1]를 진행하고 fibonacci(n) = fibonacci(n-1) + (n-2)[2]이므로 num이하의 모든 자연수 i에 대해 i번째 피보나치 수를 리스트에 저장해주고 그 저장된 값을 통해 그다음에 올 피보나치 수를 다음 인덱스에 저장하는 식으로 구현했습니다. **'과정2'는 알고리즘의 핵심연산으로 사용되어 반복문 순환에 사용된 i+=1을 제외하고 평균적으로 num번의 연산을 진행합니다.**

DP(Dynamic Progaming)은 복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법을 말합니다. 위에서 언급한 것처럼 리스트에 이전 결과들을 저장하는, 즉 메모이제이션(Memoization)이라는 방법을 사용해서 결과를 구하는데 이 문제에서는 일차원 리스트를 이용해서 구현했습니다.

1) 과정1
2) (단, n >= 2의 자연수), 과정2

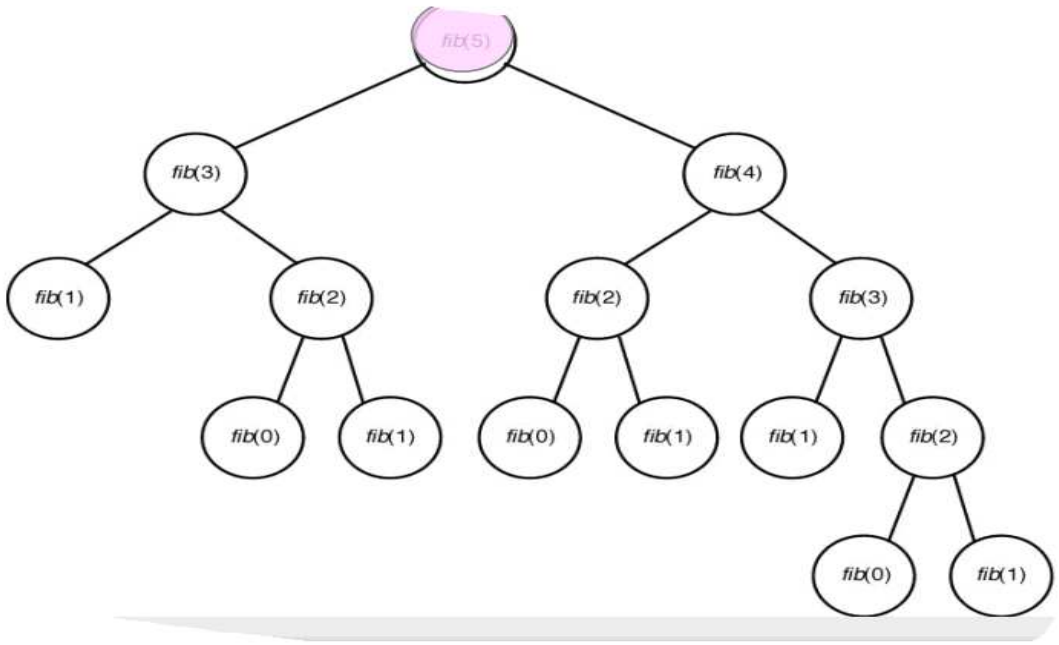
▶ def recursive_fibonacci(num):

#과정 1. 탈출조건 정의

if 0 <= num < 2: return num

#과정 2. 피보나치(재귀) 실행

return recursive_fibonacci(num-2) + recursive_fibonacci(num-1)



def recursive_fibonacci(num):, 즉 피보나치(재귀)는 원리 자체는 피보나치(반복)과 동일하지만 '과정2'에서 재귀를 통해 이전의 결과를 호출하는 과정을 통해 **핵심연산의 횟수, 즉 num에 따른 함수의 호출횟수가 위와 같은 재귀트리의 마디 개수[3]와 같은 수준**으로 증가하며

T(n)은 재귀 트리 상의 마디의 개수라고 할 때

T(0) = T(1) = 1, n>=2일 때 T(n-1) > T(n-2)이므로

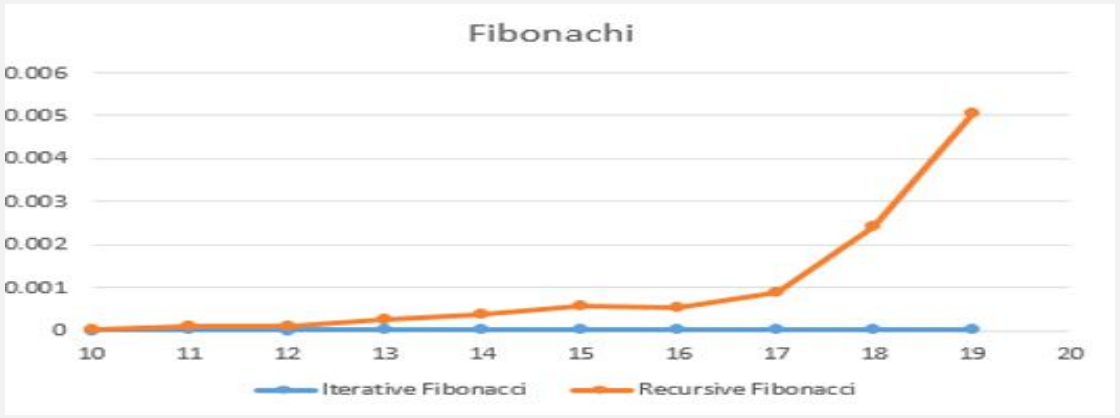
T(n) = T(n-1) + T(n-2) > 2*T(n-2)

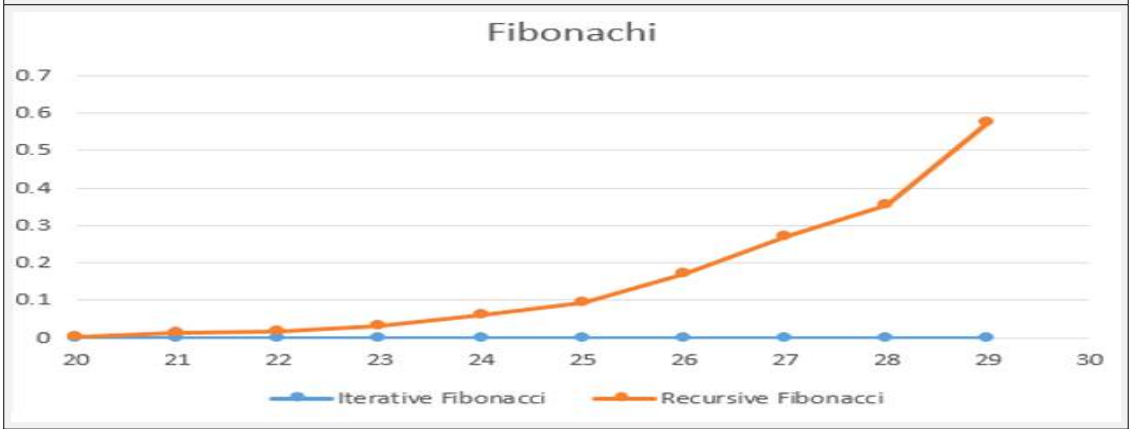
> 2*2 * T(n-4) > ... > $2^{n/2}$이고, 즉 T(n) > $2^{n/2}$이므로

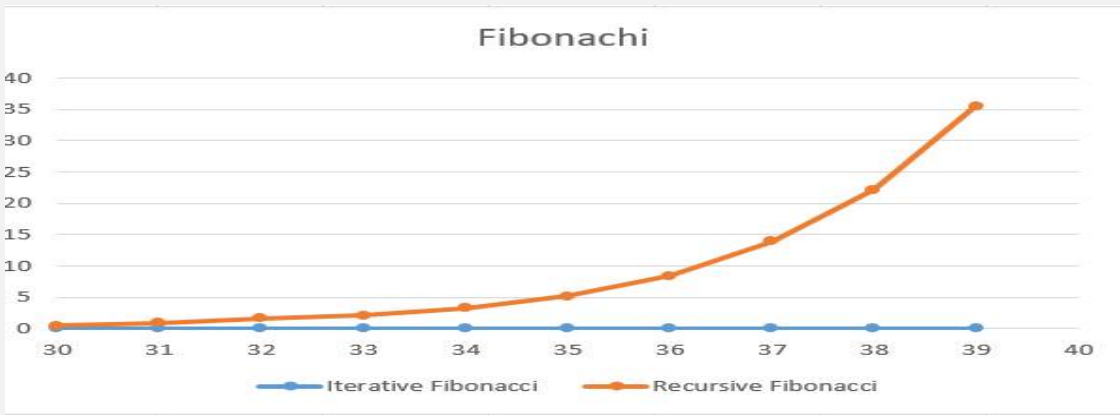
T(n), 즉 **num에 따른 핵심연산의 호출횟수는 최소 $2^{num/2}$회보다 많습니다.**

3) 알고리즘 설계와 분석의 기초 강의자료 참고

▶ 그래프를 통해 확인하기







위의 그래프는 num의 범위가 각각 [10,19], [20,29], [30,39]인 케이스를 나타냈습니다. 위의 예제를 실행시키는 동안 피보나치(반복)는 각각의 실행시간이 거의 차이가 없었지만, 피보나치(재귀)의 경우 실행시간이 기하급수적으로 증가하는 추세를 보였습니다.

▶ 실행결과 확인

```
[Iterative Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time: 0.000005s

[Recursive Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time: 0.000032s
```

```
[Iterative Fibonacci]
Num 20 : Fibonacci Number 6765
Elapsed Time: 0.000007s

[Recursive Fibonacci]
Num 20 : Fibonacci Number 6765
Elapsed Time: 0.003722s
```

```
[Iterative Fibonacci]
Num 30 : Fibonacci Number 832040
Elapsed Time: 0.000010s

[Recursive Fibonacci]
Num 30 : Fibonacci Number 832040
Elapsed Time: 0.470573s
```

```
[Iterative Fibonacci]
Num 40 : Fibonacci Number 102334155
Elapsed Time: 0.000022s

[Recursive Fibonacci]
Num 40 : Fibonacci Number 102334155
Elapsed Time: 63.657812s
```

3> 결론 : 고찰

▶ 고찰

▶ 어려웠던 점

: 실행시간 분석 측면에서는 **파이참**이라는 컴파일러를 이용했을 때 **이진검색의 실행시간이 잘 표시되지 않았던 것**에 어려움이 있었지만 교수님의 도움으로 **Colab을 이용해 수행시간**을 분석하면서 문제를 해결할 수 있었습니다. (아래는 해당 오류 화면입니다.)

```
[Sequential Search Result]
Key value 85681: location 43313
Elapsed Time: 6.16860390ms

[Binary Search Result]
Key value 85681: location 85420
Elapsed Time: 0.00000000ms

[Recursive Binary Search Result]
Key value 85681: location 85420
Elapsed Time: 0.00000000ms
```

그리고 **순차검색**을 진행할 때 **i의 인덱스를 고려하지 못해 IndexError : list index out of range**라는 오류가 간헐적으로 발생하여 난항을 겪다가 **i의 값을 바꿔주면서** 해결하게 되었습니다. (아래는 해당 오류 화면입니다.)

```
64      #과정 2. 순차검색 반복문 실행
65      for i in s:                         # 0부터 배열의 끝까지 key 값에 대한 순차검색 실행
66          #과정 3-1. 순차검색 성공한 경우
67          if key == s[i]:                 # key 값이 존재할 경우 위치(i)를 location에 대입 후 반복 탈출
68              location = i
69              break
70      #과정 3-2. 순차검색 실패한 경우
71      if location == 0:                   # [주의 및 수행 사항 2번 : location에 변화가 없다면 -1 리턴]
72          location = -1
73
74      #과정 4. 최종 결과 리턴
75      return location
76
77  # 2. 이진검색(반복 분할정복) 함수 본문
```

sequential_search() › for i in s

Variables

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

__exception__ = {tuple: 3} (<class 'IndexError'>, IndexError('list index out of range'), <traceback object at 0x0000019749BFA340>)
i = {int} 1000
key = {int} 560
location = {int} 0
num = {int} 1000
s = {list: 1000} [993, 765, 979, 388, 161, 273, 725, 235, 168, 32, 142, 625, 747, 496, 199, 570, 697, 919, 253, 167, 973, 618, 808, 10, 939, 95, 526, 460, 28

▶ 고찰

▶ 배운 점

: 과제를 진행하면서 **①DP와 분할 정복 등에 대한 알고리즘 관련 지식**과 **②최악의 경우, 평균의 경우의 핵심연산 수행 횟수를 구하는 방법**, 그리고 **③순차검색과 이진검색, 재귀와 반복의 실행시간 차이** 등에 대해 고민해보는 시간을 가질 수 있었습니다.

▶ 느낀 점

: 과제 외적으로는 파이썬을 기존에 사용해본 경험이 많이 없어 당황하긴 했었지만 예제 코드가 매우 깔끔해서 기분이 좋았습니다. 나중에 리펙터링 기법들을 학습하게 되면 저런 코딩을 할 수 있게 되는 건지 궁금해지기도 했습니다.
과제 관련해서는 알고리즘 수행시간 분석이 중요한 이유를 직접 시간을 띄워가며 확인해보니 눈으로 체감할 수 있어서, 그리고 다음 과제인 정렬 관련해서도 어떤 걸 하게 될지 기대가 되고 좋았습니다.

과제는 이상으로 마치겠습니다. 감사합니다.