**REPORT**

로고이(가) 표시된 사진

자동 생성된 설명

**과목명 : 자료구조 심화학습**

**과제내용 : 1. Divide & Conquer 기본 개념**

**2. 예제프로그램 1 (퀵 정렬, 힙 정렬, 피보나치 수열, 거듭제곱, 팩토리얼)**

**3. 예제프로그램 2 : 카라추바 곱셈**

**4. 비용감소**

**5. 퀵 정렬, 힙 정렬 시간 복잡도**

**교수명 : 심종익 교수님**

**학 년 : 3학년**

**학 번 : 201901366**

**성 명 : 서희준**

**제출일 : 23.07.17.(월)**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**<목차>**

1. **Divide & Conquer**
2. **예제 프로그램 1**
3. **퀵정렬**
4. **힙정렬**
5. **피보나치 수열**
6. **거듭 제곱**
7. **팩토리얼**
8. **예제 프로그램 2 : 카라추바 곱셈**
9. **비용감소 & 시간 복잡도**
10. **비용 감소 방안**
11. **퀵 정렬 시간 복잡도**
12. **힙 정렬 시간 복잡도**
13. **미로 시간 복잡도 분석**
14. **Divide & conqeur ( 분할 정복 ) 의 개념**

분할정복 알고리즘은 문제를 나눌 수 없을 때까지 분할하여 각각의 부분을 해결한 뒤 다시 합병하여 문제의 해를 얻는 기법이다.

이 알고리즘은 주어진 문제를 재귀적으로 2개 이상의 문제로 분할하여 분할된 문제를 비교적 간단하고 쉽게 해결한다. 그리고, 다시 분할된 문제에 대한 해를 병합하여 기존 문제의 해를 구한다. 따라서, 분할 정복 알고리즘은 3가지 부분으로 나눌 수 있다. 첫번째로, divide부분에서 문제를 분할하여 2개 이상의 문제로 나눈다. 이때 분할이 추가적으로 가능하면 처음 과정을 반복한다. 두번째 부분에서는 분할된 부분에서 문제를 해결하는 것이다. 문제를 부분으로 나누어 해결하면, 하나의 문제를 해결하는 것보다 정복하기 수월한 것이 일반적이다. 마지막으로, 정복한 각 부분을 다시 병합하여 원래 문제의 답을 얻는 것이다. 주의해야 할 것은 분할이다. 분할을 어떻게 하느냐에 따라 정복의 난이도가 달라질 수 있다. 또한, 분할 정복 알고리즘은 재귀 알고리즘이 주로 사용되는데, 이때 재귀의 비용이 알고리즘의 효율성을 떨어트릴 수 있다.

분할 정복 알고리즘에 해당하는 알고리즘을 생각하면 크게 퀵정렬, 합병정렬을 떠올릴 수 있다. 하지만, 이원탐색은 분할정복과는 약간의 차이점이 있다. 차이점을 설명하기 위해 퀵정렬과 이원탐색 알고리즘을 설명한다.

우선 퀵정렬 알고리즘은 정렬 알고리즘이다. 피벗 요소를 선택하고 배열 요소들을 재배치하여 선택한 피벗보다 작은 모든 요소들이 피벗의 왼쪽으로 이동하고, 큰 모든 요소들이 피벗의 오른쪽으로 이동시킨다. 이후 피벗 요소를 기준으로 생성된 왼쪽과 오른쪽의 부분배열에 대해 퀵정렬 알고리즘을 재귀적으로 호출한다. 이렇게 정렬된 부분배열을 만들고, 이를 병합하여 하나의 배열로써 해를 구한다. 이진탐색 알고리즘은 탐색 알고리즘이다. 입력된 요소와 배열의 중간 요소 값을 비교해서, 값이 일치하면 찾은 것으로 판단하고 인덱스를 반환한다. 그렇지 않은 경우에는 입력된 요소와 중간요소의 크기 차이를 비교한다. 이때 중간 요소의 오른쪽 부분에 대해 재귀 호출하고, 작으면 왼쪽에 대해 재귀 호출하는 방식이다. 쉽게 말해 중간 요소를 기준으로 두가지 경우 중 하나를 택해 선택지를 좁혀가는 것이다.

분할 정복은 각 단계에서 두 개 이상의 하위문제를 다룬다는 특징이 있다. 분할해서 해결하는 접근법이다. 하지만 이진탐색은 각 단계에서 하나의 하위 문제를 해결하는 탐색법이다. 따라서 이는 감소 정복(reduce & conquer) 알고리즘의 예시라고 할 수 있다.

분할 정복 알고리즘의 장점이자 목적은 어려운 문제를 쉽게 해결할 수 있다는 것이다.

즉 알고리즘의 효율성을 높일 수 있다. 분할 정복 접근법은 퀵 정렬, 합병 정렬, 고속 푸리에 변환과 같은 효율적인 알고리즘의 핵심이다. 이러한 예에서 분할 정복 접근법이 해결의 성능을 개선한다. 특히, 기본 문제가 상수 시간에 해결되는 경우, 문제를 분할하고 부분 솔루션을 결합하는 작업은 문제의 크기 n에 비례하며 각 단계에서 크기 n/p의 유한 개수 p의 하위 문제가 생성된다. 그러면 분할 정복 알고리즘의 비용은 O(n log n)이 된다. 병렬성도 무시할 수 없다. 일반적으로 분할 정복 알고리즘은 여러 프로세서를 사용하는 공유 메모리 시스템에서 사용된다. 이로 인해 프로세서 간의 데이터 통신을 계획하는 추가적인 비용없이 각각의 하위 문제는 하나의 프로세서로써 실행될 수 있습니다. 메모리 캐시 측면에서도 이점이 있다. 하위 문제의 크기가 메모리 캐시에서 해결될 만큼 작기 때문에 캐시를 효율적으로 사용할 수 있다. 이러한 알고리즘을 캐시 자동화(cache oblivious) 알고리즘이라고도 한다.

반대로 분할 정복 알고리즘의 단점은 재귀를 포함하므로 때때로 느릴 수 있다는 것이다.

함수를 계속 호출하는 방식이기 때문에 시간 복잡도 측면에서 좋지 않다. 새로운 프로세스로써 재귀 호출한 부분에서 CPU의 “context switching”이 일어나고, 이때 프로세스에 대한 정보를 PCB에 저장하는 과정도 반복된다. 각 재귀 파트를 실행하는 효율도 로직의 구현에 따라 크게 달라질 수 있다. 계속적인 로컬 변수 공간 할당과 리턴 주소 저장 등 공간 복잡도 측면에서도 상대적으로 좋지 않을 수 있다. 또한, 하위 문제들이 상호 의존적이고 특정한 순서로 해결되어야 할 때 더욱 구현이 어려워질 수 있다. 쉽게 병렬화되지 않는 경우가 있으며, 이는 계산 자원의 비효율화로 이어질 수 있다. 상황에 따라서 대용량 데이터를 처리할 때 하위 문제의 중간 결과를 저장하는 메모리 공간도 제한될 수도 있다.

따라서, 앞서 설명한 것과 같이 구현하는 로직이 장, 단점에 큰 영향을 미친다. 분할 정복을 사용하기에 적절한 경우를 분석하고, 문제의 특성과 제약을 고려하여 분할정복기법을 이용하는 것이 중요하다.

1. **Divide & conqeur ( 분할 정복 ) 예제 프로그램 1**
2. **quickSort**

퀵 정렬은 평균 성능이 매우 좋다. 평균 연산시간이 O(n log n)으로, 정렬되지 않은 다량의 데이터를 처리하기 유리하다. 이는 divide가 거의 비슷한 두 크기로 나누어졌을 때의 시간복잡도이다. 만약 대부분의 데이터가 정렬되어 있는 경우 O(n^2)의 시간복잡도를 가지는 비효율을 보여준다. 아래 프로그램에서는 예제 프로그램 1에 해당하는 모든 함수를 구현했다. 따라서, sort에 해당하는 정의와 선언을 sort.c와 sort.h로 독립적으로 관리한다.

**텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명**

수 정렬을 위해 정렬되지 않은 임의의 숫자열을 randomCreate 함수로 생성한다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Option 매개변수를 따로 유지하는 이유는 heapsort때문이다. Heap은 트리 특성상 [0]인덱스가 아닌 [1]를 첫번째 인덱스이자 root 노드이기 때문에 인덱스 1부터 값을 넣어야 하며, 10개의 숫자를 다루기 위해 [10+1]까지 공간을 확장하여 할당받는다. Quicksort는 이에 해당하지 않기 때문에 옵션 1을 전달 받아 0부터 100사이의 숫자를 임의로 10개 할당받는다. 임의의 숫자열을 생성할 때 line 57과 같이 출력문을 포함하여 생성된 임의의 숫자열을 확인할 수 있도록 한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

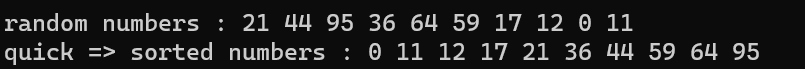
자동 생성된 설명

**텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

퀵 정렬은 피벗 요소를 좌측 배열 요소로 선택하고 배열 요소들을 피벗과 비교한다. 비교할 때 피벗보다 작은 모든 요소들이 피벗의 왼쪽으로 이동하고, 큰 모든 요소들이 피벗의 오른쪽으로 이동시켜 정렬한다. 따라서 line 12~13과 같이 i,j 인덱스를 따로 유지하여 i를 좌측부터, j는 우측부터 비교 배치한다. 이때 i,j 인덱스가 크거나 작아지며, i<j의 상황이 발생할 때 피벗의 위치가 정해진다. 피벗을 기준으로 큰 값이 오른쪽, 작은 값이 왼쪽으로 배치된 이후 i가 커지고 j가 작아지면, j의 인덱스는 i가 이전에 이미 비교한 요소를 재확인하는 불필요한 상황이 발생한다. 이러한 경우 SWAP한 요소를 다시 SWAP해야 한다. 따라서, i<j인 상황이 발생할 때 do-while문을 탈출하여, 피벗의 위치를 확정한다. 이후 피벗 요소를 기준으로 생성된 왼쪽과 오른쪽의 부분배열에 대해 퀵정렬 알고리즘을 재귀적으로 호출한다. 이러한 방식으로 피봇을 하나씩 정렬하면서, 정렬해간다. 하나의 배열의 두개로 나누면서 피벗을 정렬하는 과정으로, 분할정복의 대표적인 예이다.

<결과>

****

1. **heapsort**

힙정렬은 주어진 배열을 최대 힙구조로 만들어 루트 노드인 target[1]의 위치의 요소를 뽑아 배열의 가장 뒤에 배치하는 식으로 정렬한다. 루트에 있는 숫자를 배열의 뒤부터 배치할 때 SWAP 매크로를 사용한다. 따라서, target[1]에는 제일 작은 값은 아니지만, 최대 힙이 될 수 없는 임의의 숫자가 배치된다. 여기서 다시 adjust()함수를 호출하여, tartget[1]에 있는 던 값이 최대 힙에 맞는 위치로 배치되도록 하는 것이다. 최대 힙을 만드는 adjust정렬이 된 숫자는 정렬해야 할 숫자열에서 제외된 것을로 간주하여, 루트 노드에 들어갈 새로운 노드를 찾는 과정이 요구된다. 이는 숫자가 정렬될 때마다 나머지 숫자들로 최대 힙을 만드는 과정을 반복하는데, 트리의 깊이가 n이면 for루프는 최대 d번 수행된다. 교환과 히프크기 축소, 새로운 최대 힙 생성을 n-1 반복하면, 배열 a[1]부터 a[n]까지 정렬이 완료된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

메인 함수에서 힙에 들어갈 숫자를 임의로 10개 생성하는 함수를 호출하고, 그것을 정렬한 뒤 출력한다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

정렬하기 앞서, line96과 같이, 힙을 최대 힙으로 정렬하는 과정을 거친다. 이때 n/2인 이유는 리프노드를 가진 노드부터 탐색하기 위해서이다. 현재 생성된 트리는 완전 이진트리라고 가정하기 때문에 마지막 노드 n의 n/2는 n의 부모노드인 것을 알 수 있다. 따라서, 리프노드를 가진 부모노드부터 차례로 최대 힙이 되는지 비교하며, 조정한다. 이때 루트로 오는 노드가 line 100에서 스왑되는 것이다. 항상 최대 힙이 유지된다면, 루트는 매번 남아있는 숫자들 중 제일 큰 숫자일 것이다. 따라서, i를 배열의 끝 인덱스부터 감소하며 큰 숫자를 뒤로 넣는 오름차순 정렬을 진행한다. 이때 인덱스 1, 즉 루트노드에 대하여 adjust함수를 호출하는 이유는 앞서, 설명한 것과 같이 스왑과정에서 리프노드와 교환되기 때문이다. 리프노드에 있던 숫자가 제일 작다고 할 수는 없지만, 리프에 있었기 때문에 부모노드보다 작은 것은 분명하다. 이 숫자의 위치를 찾아주면서 다시 최대 힙을 만들면, 다른 노드들을 건드릴 필요없이 다시 최대힙을 생성할 수 있다. 노드가 힙에서 삭제되었기 때문에 버블 다운 방식으로 노드를 재조정한다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

adjust함수는 설명한대로, 최대힙을 만들기 위해 힙의 노드들 재조정하는 함수이다. 처음 힙 정렬을 시작하기 전에 루트로 받은 값은 인덱스 값 기준으로 가장 큰 부모 노드일 것이다. 버블 다운 방식으로 반복하며, 루트로 받은 값이 자식 노드보다 큰 위치를 찾는다. 이때 자식은 오른쪽과 왼쪽 둘일 수 있는데, 그 중 항상 큰 값과 비교한다. 만일 교환이 일어나더라도, 최대 힙이 유지되기 때문이다. 위치를 찾았을 때 탈출하고 line 87과 같이 비교했던 자식의 부모위치로 들어가게 된다.

**<결과>**



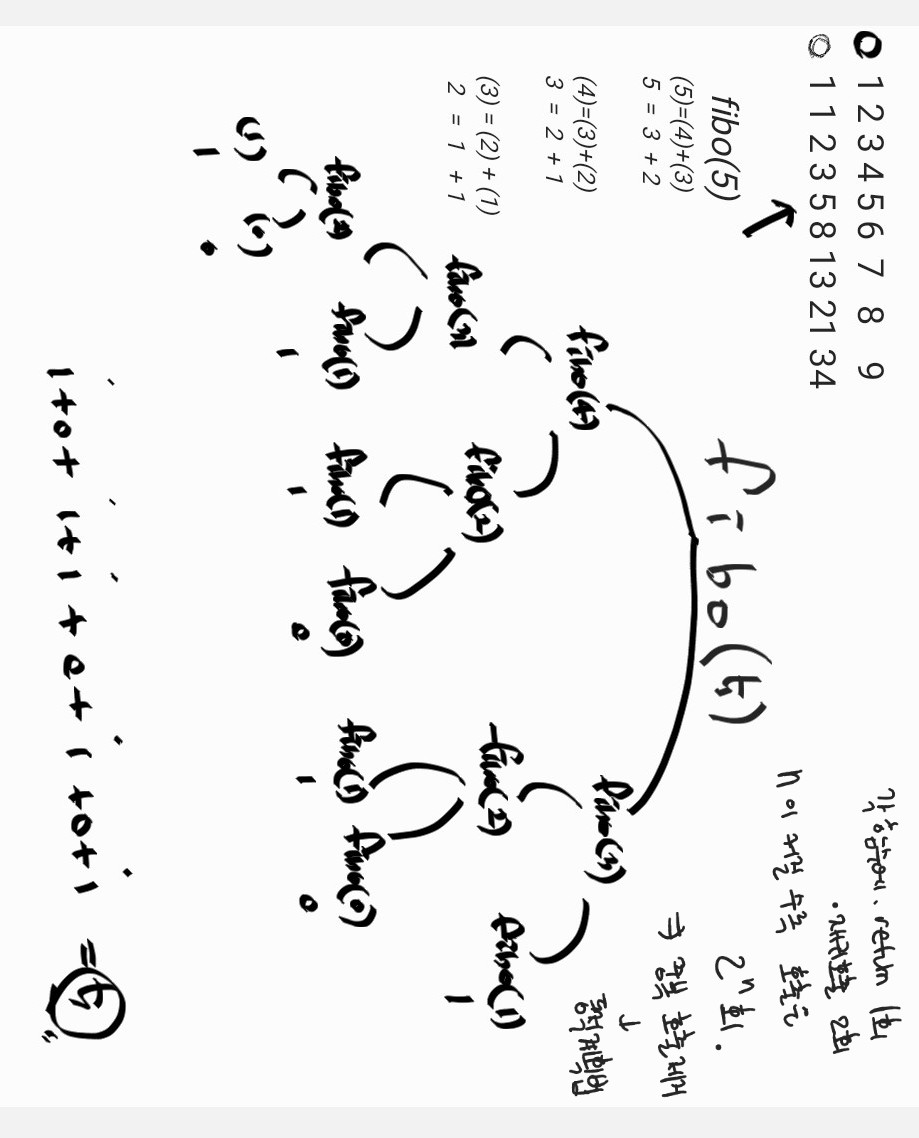
1. **피보나치 수열**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

다음은 피보나치 수열을 분할 정복 알고리즘으로 나타낸 것이다. 수열의 위치를 입력하면, 해당 위치의 수를 반환하는 알고리즘이다. 메인 함수는 다음과 같이, 위치를 입력받고 함수를 호출하여, 값을 출력한다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

수열을 분할 정복 알고리즘으로 나타낸 것이다. 수열의 위치를 입력하면, 해당 위치의 수를 반환하는 알고리즘이다. 메인 함수는 다음과 피보나치 수열의 기본적인 원래는 an = an-1 + an-2 이다. 따라서 a5 = a4 + a3 라는 것을 쉽게 알 수 있다. 이것은 또 a4 = a3 + a2  , a3 = a2 + a1 로 볼 수 있다. 이러한 방식으로 계속 수열의 위치에 대한 값을 분할하며, a1 =1 이라는 것을 이용해 그 합을 전체 해로 반환한다. 문제를 분할하는 과정에서 알 수 있지만, 재귀함수가 한번 호출될 때마다 두개의 또 다른 재귀함수를 호출하기 때문에 시간 복잡도는 O(2n)이 된다. 여기서 비효율적인 부분은 같은 값을 가지는 재귀 함수를 중복으로 호출한다는 것이다. a4 = a3 + a2  , a3 = a2 + a1 부분만 보더라도, 겹치는 부분이 많다는 것을 알 수 있다. 이러한 분할 정복은 오히려 문제해결의 효율을 떨어트릴 수 있다. 이 문제는 동적 계획법과 반복문을 이용하면, 성능을 계선할 수 있다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

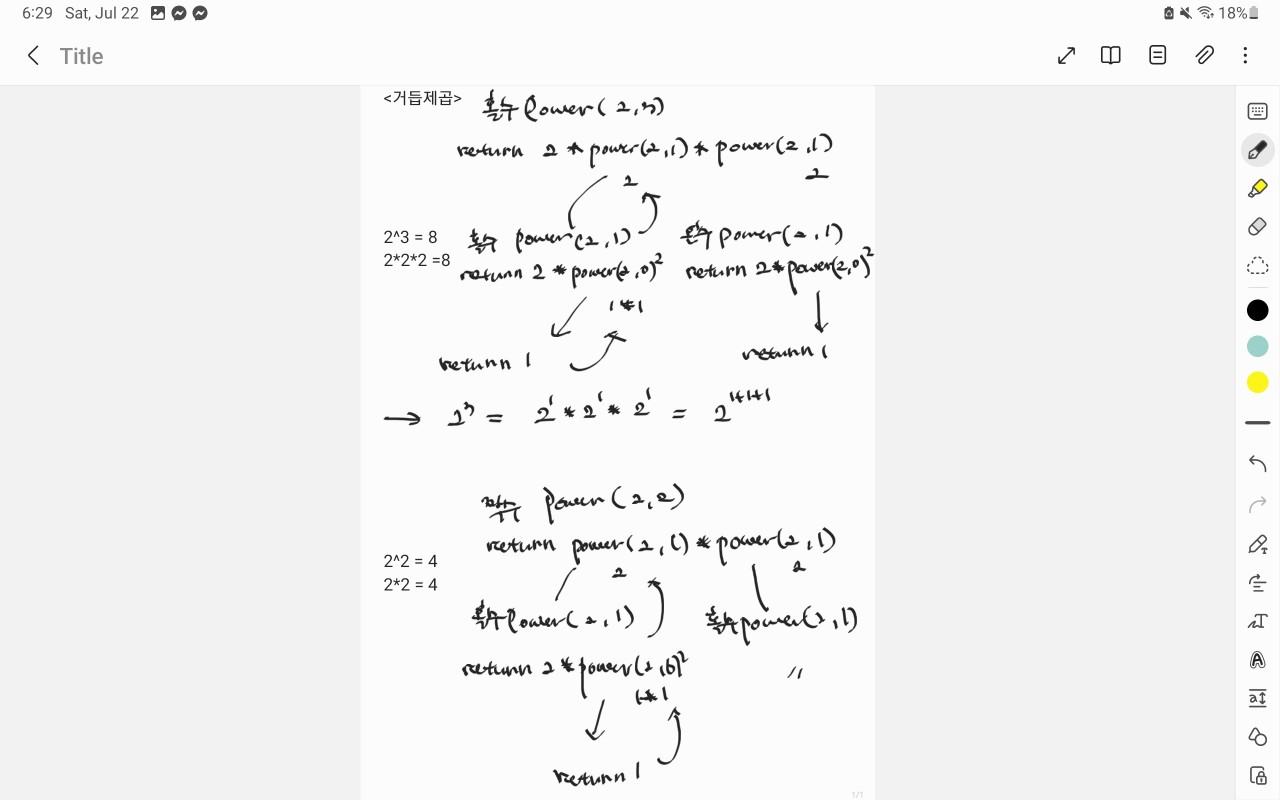
위의 프로그램은 동적 계획법을 이용해 중복된 호출에 대한 값을 저장하여 이용한다. 좌측 fValues[i]에 저장된 값을 또 다시 활용하기 때문에 재귀형태의 알고리즘보다 개선된 비용을 가진다. 피보나치 수열의 각 항을 한 번씩만 계산하면 되므로 전체 시간 복잡도가 O(n)가 된다.

1. **거듭제곱**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

거듭제곱도 밑을 지수만큼 반복적으로 곱하기 때문에 곱하는 연산을 나눠서 처리할 수 있다. 여기서 지수를 나눌때 홀수에 대한 처리가 필요하다. 24의 경우 22 \* 22 로 나눌 수 있지만, 25 = 2\*22\*22 로 처리해야 한다. 지수의 분할은 지수가 0이 되어 값이 1이 될 때까지 반복한다. 마찬가지로, 하나의 함수에서 2개의 재귀함수를 호출한다. 2n 의 과정에서 n은 log2n로 감소한다. 따라서 2 log2n=n2 가 된다. 이전의 피보나치 수열보다는 성능이 좋지만, 이 알고리즘에서도 반복되는 호출을 확인할 수 있다. 따라서 아래의 그림에서 자세히 설명한다.

****

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

Power(2,3)의 지수가 홀수인 경우 2\*power(2,1)\*power(2,1)가 반환되는데, 이때 power(2,1)이 같은 값이지만, 반복 호출되는 것을 확인할 수 있다. 이는 짝수 일때도 마찬가지이다.

**텍스트, 스크린샷, 친필, 폰트이(가) 표시된 사진

자동 생성된 설명**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

Power(2,2)의 지수가 짝수인 경우도 power(2,1)\*power(2,1)가 반환되는데, 이때 power(2,1)이 같은 값이지만, 반복 호출되는 것을 확인할 수 있다. 여기서도, 이 값을 기억할 수 있도록 새로운 변수를 생성하고, 중복 호출을 없애는 동적계획법을 이용한다.

**텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명**

여기서도, 이 값을 기억할 수 있도록 새로운 변수를 생성하고, 중복 호출을 없애는 동적계획법을 이용한다. 재귀호출될 때 half는 각 함수마다 생성되는 값으로, line 74~76과 같이 계산된 값을 저장해두고, 다시 사용하는 방식으로 변경한다. 각 재귀함수당 하나의 재귀함수를 호출하지만, 지수가 매 호출마다 expo/2로 감소하므로, 시간복잡도는 log2 expo => O(log n)의 개선된 시간복잡도를 가진다.

텍스트, 스크린샷, 멀티미디어 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

밑과 지수를 입력받아, 분할정복 알고리즘을 계산한 결과는 다음과 같다.

**<출력>**

**텍스트, 폰트, 스크린샷, 블랙이(가) 표시된 사진

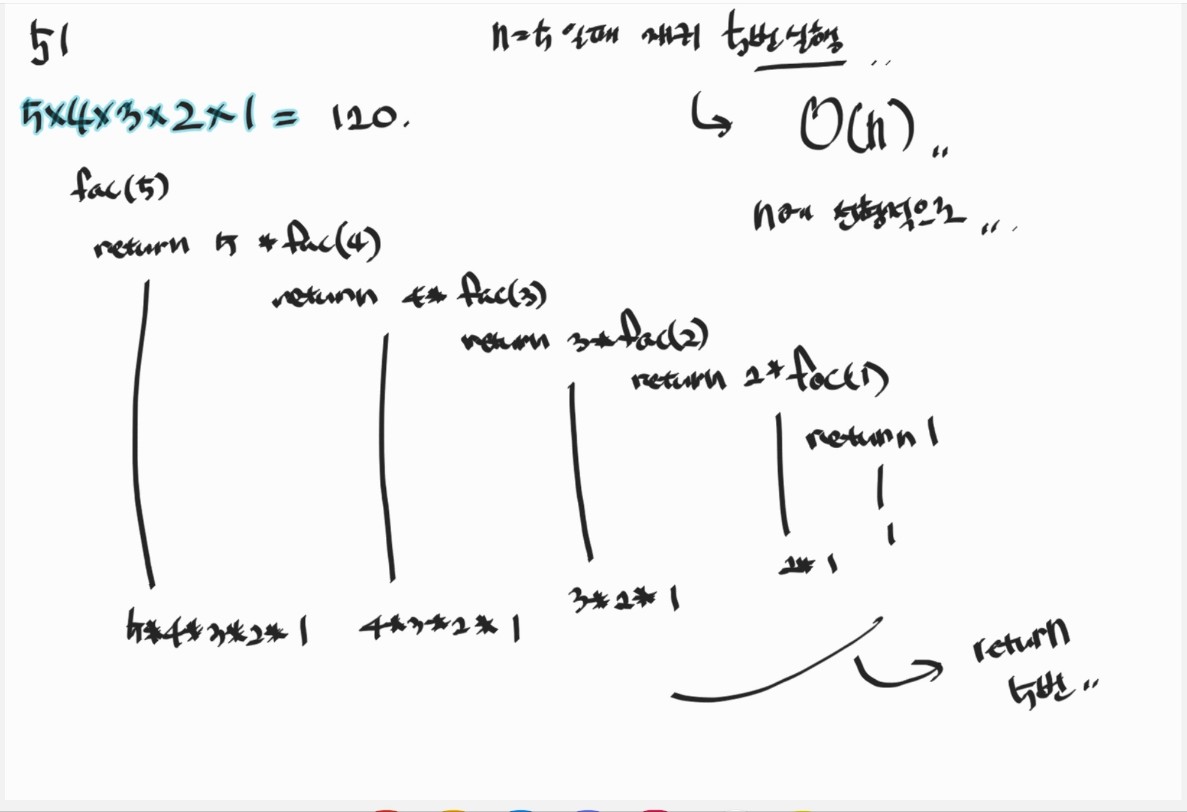
자동 생성된 설명**

1. **팩토리얼**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

팩토리얼은 비교적 간단하게 분할할 수 있다. n! 은 n\*(n-1)\*(n-2)\* ’ ’ ’ \*2\*1로 분할 계산할 수 있다. 크기 n에서 1씩 감소하며, 재귀함수를 호출하는 방식으로 n만큼 재귀함수가 호출된다. 따라서 O(n)의 시간복잡도를 가진다.



**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

메인함수에서는 n!을 계산할 n을 입력받아 그 결과를 출력한다.

**<결과>**

**텍스트, 폰트, 블랙, 화이트이(가) 표시된 사진

자동 생성된 설명**

**3. Divide & conqeur ( 분할 정복 ) 예제 프로그램 2 : 카라추바 곱셈**

텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

자동 생성된 설명

(4\*10000)+(20\*1000)+(23\*100)+(31\*10)+(12) =

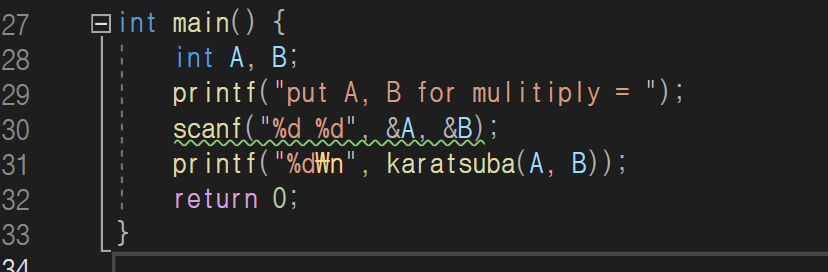
(6\*10000)+(2\*1000)+(6\*100)+(2\*10)+(2) =

6 2 6 2 2

기존에 곱셈을 하는 방식을 떠올려보면 위의 표와 같다. 213\*294의 곱셈을 할 때 294의 자릿수마다 2,1,3에 곱해서 그 값을 더하는 과정이 필요하다. 서로 같은 길이의 정수라고 가정했을 때 곱셈을 위해서 3번 반복하는 이중 for문이 요구될 것이다. (n자릿수) \*( n 자릿수) = n2 => O(n2)의 시간 복잡도

카라츠바 알고리즘은 큰 수의 곱셈을 빠르게 계산하는 알고리즘으로, 일반적인 곱셈 알고리즘에 비해 더 효율적이다. 큰 수를 작은 단위로 나누어 계산하는 분할 정복(Divide and Conquer) 기법을 활용한다. 함수 호출 횟수와 반복문이 없기 때문에, 카라츠바 알고리즘의 시간 복잡도는 재귀 호출의 깊이에 종속적이다.

메인 함수에서는 두수를 입력받고, karatsuba함수로 그 값을 전달한 뒤 계산된 값을 받아 출력한다.



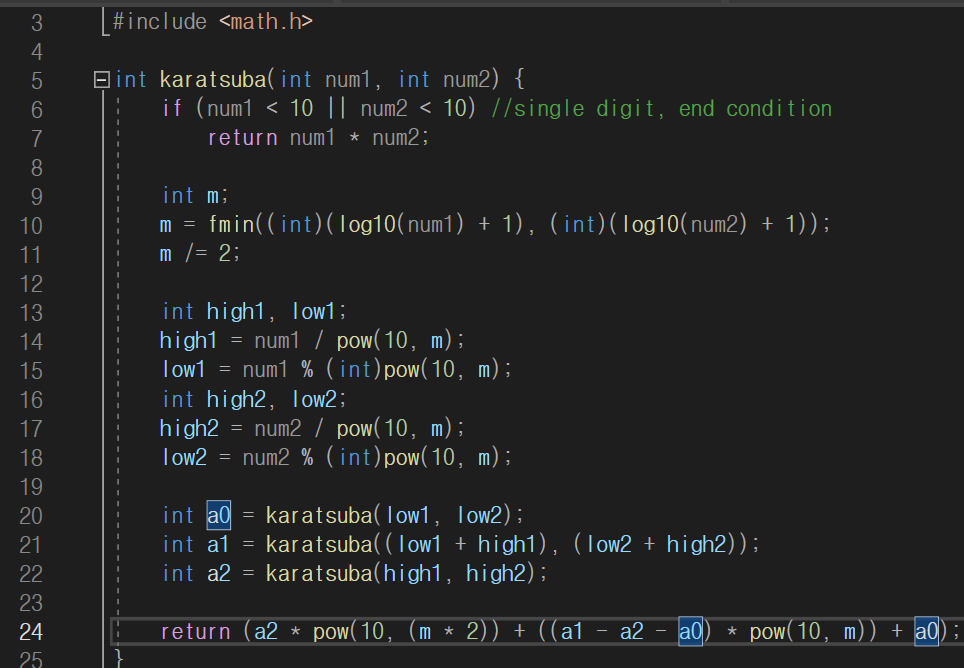
이에 대한 출력은 다음과 같다. 앞에서 수기로 계산한 결과와 동일한 것을 확인할 수 있다.

<출력>

텍스트, 폰트, 블랙, 스크린샷이(가) 표시된 사진

자동 생성된 설명

카사추바 곱셈 알고리즘은 크게 3가지 파트로 나뉜다. 첫번째는 재귀함수의 종료 조건, 두번째는 자릿수에 따른 분할, 세번째는 재귀함수 호출 및 반환이다.



텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

num1과 num2가 한 자리 수일 경우에는, 그냥 두 수를 곱하여 결과를 반환한다. 더 이상 분할할 수 없는 위치이기 때문이다. 따라서, 재귀 호출을 멈추는 조건으로 사용한다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

m을 정의하고, fmin 함수를 사용하여 두 수 num1과 num2 중 작은 자릿수를 구한다. 여기서 fmin 함수는 두 값을 비교하여 작은 값을 반환한다. log10 함수를 사용하여 num1과 num2의 자릿수를 계산하고, m을 두 수의 자릿수 중 작은 값을 반환하는 것이다. 이것의 반을 m에 저장함으로써 큰 수를 더 작은 단위로 분할하여 카라츠바 알고리즘을 적용한다. high1, low1, high2, low2를 정의하여 num1과 num2를 두 부분으로 분할한다. high1은 num1의 앞 부분 (높은 자릿수), low1은 num1의 뒷 부분 (낮은 자릿수)을 나타낸다. 마찬가지로, high2는 num2의 앞 부분 (높은 자릿수), low2는 num2의 뒷 부분 (낮은 자릿수)이다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

마지막 a0, a1, a2를 정의하고, 재귀 호출을 사용하여 부분 곱셈을 계산하여 결과값을 저장한다. a0은 낮은 자릿수 부분을 곱한 결과를 저장한다. a1은 낮은 자릿수와 높은 자릿수를 합친 후 곱셈한 결과를 저장하고, a2는 높은 자릿수 부분을 곱한 값을 저장한다. 반환할 때는 a2 \* pow(10, (m \* 2))를 m의 자릿수만큼 왼쪽으로 이동시킨다. 마찬가지로 ((z1 - z2 - z0) \* pow(10, m))를 계산하여 앞서, 설명한 자리수에 대한 보충을 해준다. 이렇게 계산한 세 부분 결과를 더한 곱셈 결과가 초기 문제의 해가 된다.

거듭제곱 연산인 pow(10, m)과 log10 함수의 시간 복잡도는 O(log m)이라고 가정할 때, 카라츠바 알고리즘의 시간 는 T(n) = 3 \* T(n/2) + O(log m)이다. 여기서 3은 재귀 호출이 3번 발생한다는 것이다. 자릿수만큼 반복해야 했던 O(n2)의 시간 복잡도를 가진 곱셈을 각 함수에서 3번의 재귀 호출로 해결할 수 있다는 것이다. 한번의 재귀에서 분할 시 T(n)의 시간 복잡도는 O(n log2(3) ) ≈ O(n 1.585 )이다. 일반적인 곱셈 알고리즘보다 훨씬 효율적인 시간 복잡도를 가진 알고리즘이며, 특히 큰 수에 대한 곱셈일 때 효율은 더 커진다.

1. **Divide & conqeur ( 분할 정복 ) 시간 복잡도**
2. **현대 기법**

요즘에는 분할 정복 알고리즘의 시간복잡도를 개선하기 위해 여러 기법이 논의되고 있다. 몇 가지 중요한 기법은 다음과 같다.

메모이제이션 (Memoization): 메모이제이션은 하위 문제들을 해결한 결과를 저장하여 중복 계산을 피하는 방법이다. 이미 계산된 하위 문제들의 결과를 캐시하고, 이후 동일한 하위 문제들을 만났을 때 저장된 결과를 활용하여 중복 계산을 제외한다. 이를 통해 시간 복잡도를 크게 줄일 수 있다.

동적 계획법 (Dynamic Programming): 동적 계획법은 최적화 문제를 해결하는데 사용되는 기법으로, 하위 문제들의 결과를 저장하고 이를 활용하여 상위 문제를 해결하는 방법이다. 동적 계획법은 메모이제이션과 매우 밀접한 관련이 있으며, 중복 계산을 피하는데 효과적이다. 이는 앞서 예제에서 설명한 바가 있다.

최적화 기법 (Optimization Techniques): 일부 분할 정복 알고리즘은 최적화 기법을 사용하여 불필요한 계산을 줄이고 최적의 해결책을 찾을 수 있도록 합니다. 예를 들어, 최소 비용으로 경로를 찾는 문제에서 백트래킹, 가지치기 등의 기법을 사용하여 가능한 경우를 모두 탐색하지 않고 최적의 경로를 찾을 수 있다.

병렬 처리기법 : 현 시대에 컴퓨터 아키텍처로 병렬처리를 한다면, 재귀함수가 이점이 되는 부분이 많다. 분할정복의 효율을 극대화할 수 있는 방법이다. 이것은 하드웨어적인 방안이라고 할 수 있다. 분할될때 리턴 값을 기다리지 않는 독립적인 프로세스가 되면, 멀티 스레드 환경에서 문제를 빠른 속도로 처리할 수 있다.

보통 재귀를 사용한 분할정복은 중복된 호출이 생길 가능성이 있는데, 대부분의 논의되는 사항은 중복된 호출을 줄이기 위해 반복문을 사용하거나, 비용을 공간에 작게 투자하여 재귀 함수에 대한 값을 저장하고, 중복 호출을 그 값으로 대신하려는 방식을 택하고 있다. 이러한 기법들은 분할 정복 알고리즘의 성능을 개선하는데 사용되며, 문제의 특성과 크기에 따라 적절한 기법을 선택하여 최적의 해결책을 찾을 수 있다.

1. **퀵 정렬 시간 복잡도 분석**

최적의 해결책을 찾을 수 있다. 두가지 경우에 따라 퀵 정렬은 큰 성능차이를 보인다. 먼저 최악의 경우는 대부분의 숫자들이 이미 정렬된 경우이다. 현재 알고리즘을 기준으로 생각해봤을 때 가장 왼쪽을 의 숫자를 피벗을 선택하기 때문에 피벗은 비교할 숫자열 중에서 작은 위치에 속하게 된다. 한번의 정렬이 끝나고, 피벗의 위치가 정해지면, 피벗을 기준으로 나눈 두개의 새로운 퀵 정렬함수를 재귀호출한다. 이때 아래 그림과 같이 피벗의 위치가 정해질 경우 좌 우가 똑같이 분할되지 않기 때문에 모든 숫자를 탐색하기 위해 n의 시간과 n-1의 재귀함수 호출이 발생한다. 따라서, n2-1 로, O(n2)의 시간 복잡도를 가지게 된다.

**텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진

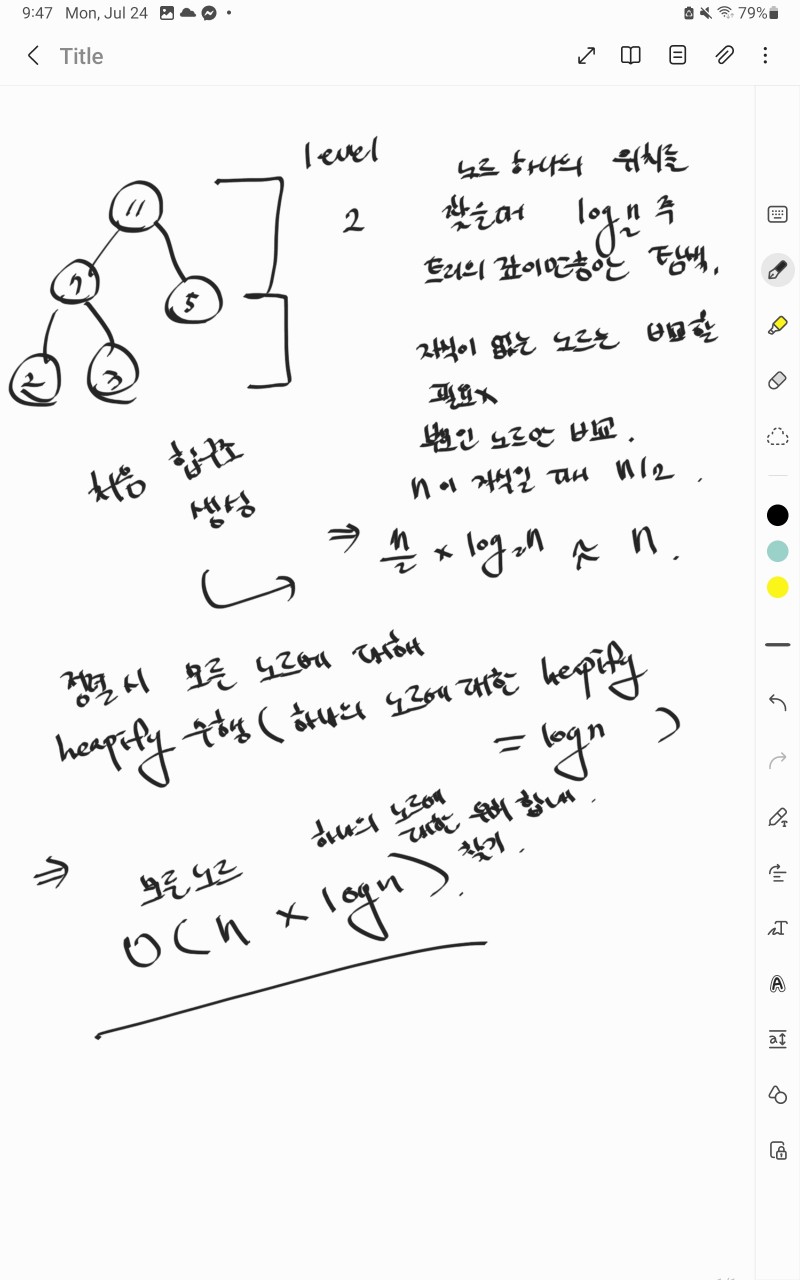
자동 생성된 설명**

**텍스트, 친필, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

반대로, 가장 왼쪽에서 선택한 피벗이 숫자열에 중앙에 위치할 수 있게 된 경우 피벗을 기준으로 정확히 반으로 나뉘게 되며, 이때 재귀 함수의 호출이 최악의 경우에 비해 훨씬 감소한다. 반을 분할하며 정렬하기 때문에 log2n 번 재귀 함수가 호출된다. 한번의 함수에서 피벗의 위치를 결정하기 위해 모든 숫자열을 탐색해야 하는 것은 동일하므로, n의 시간이 소요된다. 따라서, 평균적인 경우 O(n\*logn)의 시간복잡도를 가진다. 로그의 성질 상 데이터의 양의 많아질수록 그 차이가 커진다는 것을 쉽게 알 수 있다.

1. **힙 정렬 시간 복잡도 분석**

****

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

힙 정렬은 힙 구조를 먼저 생성한 이후에 루트 노드를 배열의 맨 뒤부터 배치하는 오름차순 정렬이다. 먼저 처음 힙 구조를 생성할 때 하나의 노드에 대한 위치를 찾는 과정을 n/2번 반복한다. 하나의 완전 이진트리는 여러 이진트리의 집합으로 볼 수 있다. 여기서 i가 그 부분트리들의 루트라고 할 때, 각 트리를 최대 힙으로 만들기 위한 시간 복잡도는 그 트리의 깊이 만큼이다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

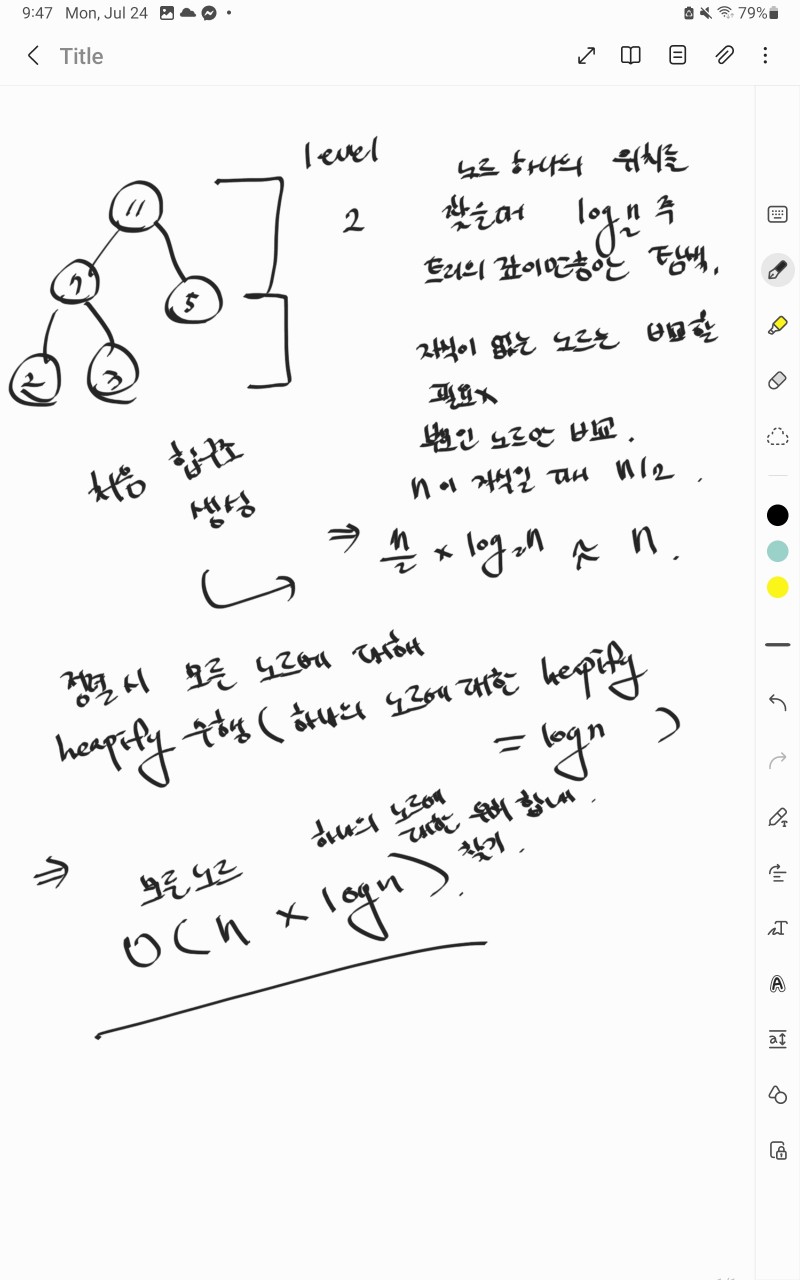
자동 생성된 설명**

다시 비교할 때는 형제 트리에 대한 비교는 할 필요가 없기 때문에 사실 상 트리의 깊이만큼만 교환이 이루어진다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

이러한 반복을 n번, 즉 숫자의 개수만큼 반복해야 최대 힙에서의 루트에 위치한 노드를 배열의 뒤에서부터 정렬할 수 있다.



모든 노드에 대한 최대 힙 생성은 앞서 설명한 것과 같이, logn이기 때문에 힙 정렬의 시간복잡도는 O(n\*logn)이 된다.

**<참고 자료>**

1. [알고리즘] 카라추바의 빠른 곱셈(Karatsuba . ( 2020. 7. 28. 17:17). https://hhlab.tistory.com/11.
2. Karatsuba algorithm for fast multiplication using Divide and Conquer algorithm . (18 Apr, 2023). Retrieved from https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/.