## 정렬 알고리즘 보고서

* 유지원 / 2208
* 제출일

### 1. 정렬 알고리즘 소개

#### 알고리즘명: 스무스 정렬 (Smooth Sort)

* 정의 및 동작 원리 설명

스무스 정렬(Smoothsort)은 에츠허르 데이크스트라(Edsger Dijkstra)가 1981년에 발표한 비교 정렬 알고리즘으로, 힙 정렬의 변형에 속합니다. 일반적인 힙 정렬과 마찬가지로 제자리(in-place) 에서 수행되며 입력 순서가 보장되지 않는 불안정 정렬입니다. 스무스 정렬의 최선 시간 복잡도는 O(n) (입력이 거의 정렬되어 있을 경우)이고, 평균 및 최악 복잡도는 O(n log n)으로 힙 정렬과 동일합니다. 특히 이미 정렬된 배열에 대해 선형 시간에 가까운 속도로 동작하므로 어댑티브 정렬(adaptive sort)의 성격을 띱니다. 아래에서는 스무스 정렬의 핵심 개념인 레오나르도 수열과 레오나르도 힙을 살펴보고, 알고리즘의 동작 원리와 연산 과정, 시간 복잡도 분석, 의사코드, 파이썬 구현, 그리고 다른 정렬 알고리즘과의 성능 비교 결과를 상세히 설명합니다.

1. 레오나르도 수열과 레오나르도 힙

레오나르도 수열(Leonardo numbers)은 스무스 정렬의 이론적 기반이 되는 수열로, 피보나치 수열과 유사하지만 다음과 같은 점화 관계로 정의됩니다.

* L(0) = 1, L(1) = 1
* L(k+2) = L(k+1) + L(k) + 1   *(k ≥ 0)*

이 점화식에 따라 생성되는 레오나르도 수열의 처음 몇 항은 다음과 같습니다:

1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, …

레오나르도 수 L(k)는 흔히 레오나르도 트리(Leonardo tree)의 노드 개수를 나타내는데, 레오나르도 트리란 크기가 L(k)인 형태의 이진 트리입니다. 이 트리는 순서 k의 레오나르도 트리라고도 하며, 재귀적으로 정의됩니다:

* k = 0 또는 k = 1일 때 레오나르도 트리는 노드 1개로 구성됩니다 (즉, 크기 L(0)=L(1)=1).
* k ≥ 2일 때 순서 k 레오나르도 트리는 왼쪽 자식 서브트리로 순서 k-1 레오나르도 트리, 오른쪽 자식 서브트리로 순서 k-2 레오나르도 트리를 가지며, 이 두 서브트리의 루트 위에 새로운 루트 노드를 연결한 구조입니다. 이렇게 하면 전체 노드 수는 L(k-1) + L(k-2) + 1 = L(k)가 되어 점화식과 일치합니다.

이러한 정의에 의해, 레오나르도 트리는 항상 노드 1개이거나 2개의 자식을 가진다는 특징이 있습니다 (한쪽 자식만 있는 경우가 발생하지 않습니다). 예를 들어, 순서 4의 레오나르도 트리는 크기 L(4)=9인 트리이며, 그 왼쪽 서브트리는 순서 3 (L(3)=5 노드), 오른쪽 서브트리는 순서 2 (L(2)=3 노드)의 레오나르도 트리로 구성됩니다.

스무스 정렬에서는 데이터를 레오나르도 힙(Leonardo heap) 이라는 구조로 관리하는데, 이는 하나 이상의 레오나르도 트리들로 이루어진 포레스트(forest) 형태의 힙입니다. 즉, 정렬할 n개의 원소를 여러 개의 레오나르도 트리에 분산시켜 저장하며, 각 트리의 노드 수는 레오나르도 수열값 중 하나가 됩니다. 레오나르도 힙에 포함된 트리들의 크기는 모두 다르며, 보통 Greedy 알고리즘으로 가장 큰 레오나르도 수부터 차례로 n을 표현하도록 트리 크기가 결정됩니다. 예를 들어 n=13이라면, 13에 가장 가까운 레오나르도 수 L(4)=9를 택하고 나면 남은 4를 L(2)=3과 L(0)=1로 분해하여, 크기 9, 3, 1인 레오나르도 트리 3개로 포레스트를 구성할 수 있습니다. 이러한 포레스트가 곧 레오나르도 힙 구조입니다.

레오나르도 힙에서는 각 레오나르도 트리가 부분 힙의 역할을 하며, 일반적인 최대 힙(max-heap)과 유사하게 각 트리 내에서는 부모 노드의 값이 자식 노드의 값보다 크거나 같다는 힙 속성을 만족시킵니다. 더 나아가, 레오나르도 힙을 구성하는 여러 트리들의 루트 노드들 사이에도 정렬된 순서 관계가 유지됩니다. 구체적으로, 레오나르도 힙에서는 오른쪽으로 갈수록 루트 값이 작아지도록 (또는 왼쪽으로 갈수록 커지도록) 루트들을 배치하여, 가장 오른쪽 끝 트리의 루트가 전체 힙에서 가장 큰 값이 되도록 유지합니다. 이러한 추가 제약을 데이크스트라는 각 루트에 가상의 “의붓아들(stepson)” 링크를 두어 좌측 이웃 루트와 연결하는 방식으로 구현하였고, 이를 통해 모든 트리를 하나의 전체 힙처럼 취급하여 전역 최댓값이 항상 배열의 끝에 위치하도록 만들었습니다. 아래 그림은 크기가 9, 3, 1인 세 개의 레오나르도 트리로 구성된 레오나르도 힙의 예시를 보여줍니다. 각 트리마다 루트 노드에 그 트리의 최대값이 저장되어 있으며, 가장 오른쪽 트리의 루트가 전체 원소 중 최대값(예시에서는 97)입니다.

예시: 크기 9, 3, 1의 세 레오나르도 트리로 이루어진 레오나르도 힙. 왼쪽부터 순서 4 (노드 9개), 순서 2 (노드 3개), 순서 1 (노드 1개)의 레오나르도 트리들이 나열되어 있다. 각 트리는 최대 힙 속성을 만족하여 루트에 해당 트리의 최댓값이 위치하며, 루트들 사이도 왼쪽이 더 크도록 유지되므로 전체 최대값 97은 가장 오른쪽 루트에 위치해 있다.

2. 스무스 정렬의 동작 원리

스무스 정렬은 크게 두 단계로 이루어집니다. 첫 번째 단계에서는 입력 배열의 앞부분부터 점진적으로 레오나르도 힙 구조를 구축하고(힙 증가 단계), 두 번째 단계에서는 힙에서 가장 큰 원소를 차례로 제거하여 배열 뒷부분부터 정렬된 상태로 확정합니다(힙 감소 단계). 이러한 과정은 전통적인 힙 정렬의 “구축 + 반복적 최대 추출” 절차와 유사하지만, 단일 이진 힙 대신 여러 개의 레오나르도 힙을 사용하고, 추가적인 규칙으로 인해 이미 정렬된 데이터에 가까울수록 연산 비용이 줄어드는 형태로 최적화되어 있습니다. 이제 각 단계별 연산을 상세히 살펴보겠습니다.

2.1 힙 구축 단계 (삽입 연산)

우선 빈 레오나르도 힙 포레스트에서 시작하여, 배열의 원소들을 순차적으로 하나씩 레오나르도 힙 구조에 삽입합니다. A[0] (첫 번째 원소)은 포레스트의 첫 번째 트리 (노드 1개짜리 레오나르도 트리)로 사용됩니다. 이후 일반적인 인덱스 i 단계에서 A[i]를 힙에 추가할 때에는 다음 합병 규칙을 따릅니다.

* 합병 규칙: 현재 포레스트의 마지막 두 트리 크기가 연속된 레오나르도 수 L(x+1)와 L(x) (예: 5와 3 또는 3과 1 등)라면, 새로 추가한 원소를 이 둘을 합친 더 큰 레오나르도 트리 L(x+2)의 루트로 만들고 세 트리를 하나로 합칩니다.
* 위 조건을 만족하지 않으면, 새 원소는 크기 1의 새로운 레오나르도 트리 (순서 1 또는 0)로 포레스트에 추가됩니다. 단, 특별한 예외로 이미 마지막 트리가 크기 1 (L(1))인 상황에서 또 하나의 크기 1 트리를 추가하게 될 경우, 두 트리를 연속된 크기로 간주하기 위해 두 번째 트리를 L(0)으로 취급합니다. (L(0)과 L(1)은 값은 같지만 순서가 다르므로 “연속된” 크기라고 볼 수 있습니다.)

위 과정을 통해 새로운 트리가 추가되거나 기존 두 트리가 합병되어 더 큰 트리가 생성되면, 해당 트리에 대하여 힙 속성과 루트 정렬(문맥상 “스트링 속성(string property)”라고도 함) 조건을 복구해야 합니다. 이를 위한 스무스 정렬만의 특수한 연산이 “트링클(trinkle)”이며, 이는 아래와 같이 구현됩니다.

1. 루트 정렬 속성 복구: 새로 형성된 (또는 합병된) 트리를 현재 “현재 힙(current heap)”으로 지정합니다. 이 트리의 왼쪽에 이웃한 트리가 존재하고, 그 트리의 루트 값이 현재 힙의 루트 값보다 크다면 두 루트 노드를 교환합니다 (더 큰 값이 오른쪽으로 이동하도록). 이 교환은 현재 힙의 힙 속성을 깨뜨리지 않으면서, 이웃한 트리들의 루트들이 올바른 크기 순서를 갖도록 합니다. 교환이 발생하면 이제 더 큰 값이 이동한 왼쪽 트리가 새로운 “현재 힙”이 되고, 이 과정(왼쪽 이웃과 루트 비교 및 교환)을 반복합니다. 즉, 포레스트에서 루트들의 좌우 순서가 올바르게 될 때까지 루트 값을 거품올리기(bubble up) 합니다.
2. 힙 속성 복구 (내부 정렬): 이제 포레스트 내 루트들의 상대 순서는 만족되었지만, 현재 힙의 내부에서는 방금 삽입된 (또는 이동된) 값이 자식들보다 작아졌을 수 있습니다. 따라서 이 값을 자식들과 비교하며 아래로 내려보내는 “필터(filter)” 연산, 즉 하향-힙화(sift-down)를 실행합니다. 현재 힙의 루트 노드가 두 자식 중 더 큰 자식보다 작다면, 더 큰 자식과 루트 값을 교환합니다. 그런 다음 교환된 자식 노드를 새로운 루트(현재 힙) 위치로 삼아 이 과정을 해당 서브트리가 리프에 도달하거나 더 이상 자식보다 클 때까지 반복합니다. 레오나르도 트리는 각 노드가 반드시 두 개 혹은 0개의 자식만 가지므로, 이 필터 연산은 비교적 단순하게 구현할 수 있습니다.

위 삽입 과정을 배열의 마지막 원소까지(A[n-1]) 반복하면, 입력 배열의 전체 원소가 레오나르도 힙 포레스트에 모두 포함됩니다. 이 단계가 완료된 시점에서 배열 내의 각 원소들은 레오나르도 힙 구조의 힙 속성을 만족하고 있으며(즉, 각 레오나르도 트리는 자체적으로 최대 힙 정렬), 추가로 루트들 사이의 크기 순서도 유지되어 배열의 마지막 원소가 전체 최대값이 됩니다. 중요한 것은, 입력 데이터가 이미 정렬되어 있었다면 애초에 루트들의 순서가 항상 조건을 만족하므로 교환 연산이 전혀 일어나지 않고, 필터 연산도 최소한으로 일어나 삽입 단계 전체가 선형 시간에 완료된다는 점입니다. 반면 무작위 데이터의 경우 여러 번의 합병과 루트 교환, 필터 연산이 발생하여 힙 정렬과 동일한 O(n log n) 단계가 됩니다.

2.2 힙 감소 단계 (삭제 연산)

두 번째 단계에서는, 힙 구조에 남아있는 모든 원소를 하나씩 제거하면서 배열을 정렬된 상태로 만들어갑니다. 이 과정은 힙 정렬의 반복적 최대 추출 단계와 유사하나, 레오나르도 힙 특유의 동작을 갖습니다. 구체적으로는 다음과 같습니다:

1. 최대값 추출: 현재 레오나르도 힙 포레스트에서 가장 큰 값(전역 최댓값)은 항상 가장 오른쪽 레오나르도 트리의 루트에 위치합니다. 이 값을 제거하여 배열의 가장 뒤쪽 정렬 위치로 보냅니다. (구현상으로는 이 값과 배열의 마지막 위치 값을 교환한 뒤, 배열의 크기를 1 줄이는 식으로 처리할 수 있습니다.)
2. 힙 축소 및 복구: 최댓값을 제거하면 해당 레오나르도 트리는 루트가 없어지면서 두 개의 서브트리로 분리됩니다. 예를 들어 크기 9 레오나르도 트리에서 루트를 제거하면, 남은 노드들은 크기 5와 3의 두 레오나르도 트리가 됩니다. 이러한 노출된 두 서브 힙을 기존 포레스트에 다시 편입시켜야 합니다. 우선 제거된 트리의 위치에 그 왼쪽에 있던 이웃 트리들이 오도록 포레스트를 재구성하고, 방금 분리된 두 서브트리를 포레스트의 가장 왼쪽 두 트리로 추가합니다. (왜 왼쪽에 추가하느냐 하면, 새로 생긴 두 트리는 원래 큰 트리의 부분이었으므로 비교적 큰 크기를 가지며, 포레스트에서 크기 순으로 왼쪽편에 위치하게 됩니다.)
3. 힙 및 루트 속성 재정립: 새로운 포레스트에서, 방금 추가된 두 트리의 루트들과 주변 트리 루트들 간의 크기 순서(스트링 속성)를 다시 확인하고 필요하면 교환합니다. 특히, 왼쪽에서 편입된 두 트리 중 첫 번째 트리의 루트와 그 왼쪽 이웃 트리의 루트를 비교하여 필요하면 교환(더 큰 값을 오른쪽으로)하고, 이어서 두 번째 트리에 대해서도 같은 작업을 수행합니다 (데이크스트라의 설명에 따르면 이때 왼쪽 이웃의 루트와 새로 노출된 두 힙의 자식 루트들과는 비교할 필요가 없으며, 루트들 간 비교만으로 충분합니다). 루트 정렬이 끝나면, 각 새로운 트리에 대해 필터(하향 힙화) 연산을 적용하여 각 트리 내부의 힙 속성을 복구합니다. 이때도 레오나르도 트리의 구조상 자식 누락의 염려가 없으므로 효율적으로 수행됩니다.

위의 추출 단계를 남은 원소가 없을 때까지 반복하면, 최대값부터 차례로 배열의 뒤쪽으로 확정되어 결국 전체 배열이 오름차순으로 정렬됩니다. (또는 값을 별도 리스트에 누적하면 오름차순 결과를 얻을 수 있습니다.) 스무스 정렬은 이러한 감소 단계에서도 입력이 일부 정렬되어 있었다면 작업이 빨라지는 성질이 있습니다. 그러나 최악의 경우에는 힙 정렬과 동일하게 매 단계 O(log n)씩 걸려 총 O(n log n)의 시간이 소요됩니다.

A screenshot of a computer

AI-generated content may be incorrect.

* 의사코드 or 흐름도

PROCEDURE NUMBERS-LEONARDO(n)

# n 개의 요소를 덮을 수 있는 레오나르도 수열(L) 생성

# 1-indexed 배열을 가정

L ← [1, 1]

next ← L[LENGTH(L)] + L[LENGTH(L) − 1] + 1

WHILE LENGTH(L) ≥ 2 AND n > next DO

APPEND next TO L

next ← L[LENGTH(L)] + L[LENGTH(L) − 1] + 1

END WHILE

REVERSE(L) # 클러스터화할 때 큰 수부터 사용하기 위해 뒤집기

RETURN L

END PROCEDURE

PROCEDURE ARR-TO-HEAP(A[1..n])

# 입력 배열 A를 레오나르도 수 크기에 따라 분할한 뒤,

# 각 부분을 힙으로 변환하고, 클러스터 리스트(히프 리스트)를 반환한다.

#

# 반환값 listHeaps는 각 부분(리스트)이 MIN-HEAP 성질을 갖도록

# heapify된 리스트들의 1-indexed 배열이다.

L ← NUMBERS-LEONARDO(n)

listHeaps ← ⟨empty list⟩

m ← 1

FOR EACH size IN L DO

IF n − m + 1 ≥ size THEN

# A[m .. m+size−1] 구간을 새로운 힙으로 추가

subHeap ← A[m .. m + size − 1]

CALL BUILD-MIN-HEAP(subHeap) # CLRS의 힙구축

APPEND subHeap TO listHeaps

m ← m + size

END IF

END FOR

REVERSE(listHeaps) # Python 코드에서는 뒤집어서 큰 덩어리부터 처리

RETURN listHeaps

END PROCEDURE

PROCEDURE COUNT-INDEXES(i, idxList)

# 1-indexed 힙 H에서, idxList[i] 노드의 자식 인덱스를 idxList에 추가

# idxList는 1-indexed 배열로 가정 (즉, 첫 번째 원소의 인덱스는 1)

APPEND 2 \* idxList[i] + 1 TO idxList

APPEND 2 \* idxList[i] + 2 TO idxList

RETURN idxList

END PROCEDURE

PROCEDURE GET-LIST(idxList[1..k], H[1..m])

# 힙 H에서 idxList에 포함된 인덱스 원소들을 새로운 리스트에 모아 반환

newList ← ⟨empty list⟩

FOR j FROM 1 TO LENGTH(idxList) DO

IF idxList[j] ≤ m THEN

APPEND H[idxList[j]] TO newList

END IF

END FOR

RETURN newList

END PROCEDURE

PROCEDURE HEAP-DIVISION(H[1..m])

# 힙 H를 왼쪽 자식과 오른쪽 자식들로 분할하여 두 개의 리스트(heap) 반환

# Python 코드의 heapDivision과 동일

# 초기 인덱스 리스트

idxLeft ← [1] # 왼쪽 자식 인덱스로 시작

idxRight ← [2] # 오른쪽 자식 인덱스로 시작

i ← 1

WHILE idxLeft[LENGTH(idxLeft)] < m DO

idxLeft ← COUNT-INDEXES(i, idxLeft)

idxRight ← COUNT-INDEXES(i, idxRight)

i ← i + 1

END WHILE

heapLeft ← GET-LIST(idxLeft, H)

heapRight ← GET-LIST(idxRight, H)

RETURN (heapLeft, heapRight)

END PROCEDURE

PROCEDURE SMOOTHSORT(A[1..n])

# 주어진 배열 A를 Smooth Sort 방식으로 정렬하여,

# 오름차순 정렬된 결과 리스트를 반환

listHeaps ← ARR-TO-HEAP(A) # 배열을 레오나르도 힙들로 분할하여 리스트 생성

result ← ⟨empty list⟩

WHILE listHeaps ≠ ⟨empty list⟩ DO

# 1) 각 힙의 루트(최소값)를 찾아서, 전체 힙 리스트 중 '최소 루트' 위치를 구함

minRootValue ← +∞

minIndex ← −1

FOR hIdx FROM 1 TO LENGTH(listHeaps) DO

IF listHeaps[hIdx][1] < minRootValue THEN

minRootValue ← listHeaps[hIdx][1]

minIndex ← hIdx

END IF

END FOR

# 2) 루트끼리 맞교환: listHeaps[1].root ↔ listHeaps[minIndex].root

root0 ← listHeaps[1][1]

rootMin ← listHeaps[minIndex][1]

# CLRS의 HEAP-REPLACE 개념으로 생각할 수 있다.

listHeaps[1][1] ← rootMin

listHeaps[minIndex][1] ← root0

# 3) 만약 첫 번째 힙의 크기가 1보다 크면, 분할 필요

IF LENGTH(listHeaps[1]) > 1 THEN

flag ← TRUE

(heapLeft, heapRight) ← HEAP-DIVISION(listHeaps[1])

ELSE

flag ← FALSE

END IF

# 4) 첫 번째 힙에서 최소값(루트)을 추출

# CLRS의 HEAP-POP (MIN-HEAP 기준으로 pop)

minimum ← HEAP-EXTRACT-MIN(listHeaps[1])

APPEND minimum TO result

# 5) 첫 번째 힙(이제 루트가 제거되어 비어 있거나 크기가 작아짐)을 리스트에서 제거

DELETE listHeaps[1]

# 6) 만약 HEAP-DIVISION이 발생했다면, 분할된 두 힙을 맨 앞에 삽입

IF flag = TRUE THEN

INSERT heapRight AT listHeaps[1]

INSERT heapLeft AT listHeaps[1]

END IF

END WHILE

RETURN result

END PROCEDURE

* 시간 복잡도: 최선 / 평균 / 최악

 최선: O(n) (거의 정렬된 입력에 대해 선형 시간 달성)

 평균: O(n log n)

 최악: O(n log n)

* 안정성 여부 (stable)

스무스 정렬은 **불안정 정렬 (unstable)** 이다. 같은 값끼리의 상대적 순서를 보장하지 않는다. 힙 기반 연산에서 부모와 자식을 교환하다 보면, 값이 같은 원소라도 순서가 뒤바뀔 수 있다.

### 2. 파이썬 코드 구현

import heapq

def numbersLeonardo(size):

    numbers = [1, 1]

    nextNumber = numbers[-1] + numbers[-2] + 1

    while len(numbers) >= 2 and size > nextNumber:

        numbers.append(nextNumber)

        nextNumber = numbers[-1] + numbers[-2] + 1

    numbers.reverse()

    return numbers

def arrToHeap(data):  *# 배열을 받으면 heap 구조로 바꿈*

    leonardoNumbers = numbersLeonardo(len(data))

    listHeaps = []

    m = 0

    for i in leonardoNumbers:

*# 아직 할당 안된 배열이 다음 레오나르도 수보다 크거나 같으면*

        if len(data) - m >= i:

            listHeaps.append(data[m: m+i])

*# 할당 안된 부분으로 이동*

            m += i

*# 힙 성질 맞추기*

    for i in listHeaps:

        heapq.heapify(i)

*# heap은 non-decreasing이기 때문에 뒤집기*

    listHeaps.reverse()

    return listHeaps

def countIndexes(i, indexes):

    indexes.append(2\*indexes[i]+1)

    indexes.append(2\*indexes[i]+2)

    return indexes

def getList(indexPart, heap):

    heapPart = []

    for i in indexPart:

        if i < len(heap):

            heapPart.append(heap[i])

    return heapPart

def heapDivision(heap):

    heapleft = []

    heapright = []

    index = 0

    indexesLeft = [1]

    indexesRight = [2]

    while indexesLeft[-1] < len(heap):

        indexesLeft = countIndexes(index, indexesLeft)

        indexesRight = countIndexes(index, indexesRight)

        index += 1

    heapleft = getList(indexesLeft, heap)

    heapright = getList(indexesRight, heap)

    return heapleft, heapright

*# Time Complexity O(n) | O(nlogn) | O(nlogn)*

*# Space Complexity AUX : O(1), Total : O(n)*

def smoothSort(array):

    listHeaps = arrToHeap(array)

    result = []

    heapLeft, heapRight = 0, 0

    while (listHeaps):

        flag = 0

        minIndex = listHeaps.index(min(listHeaps))

        currentRoot = listHeaps[0][0]

        currentMin = listHeaps[minIndex][0]

        heapq.heapreplace(listHeaps[0], currentMin)

        heapq.heapreplace(listHeaps[minIndex], currentRoot)

        if len(listHeaps[0]) > 1:

            heapLeft, heapRight = heapDivision(listHeaps[0])

            flag = 1

        minimum = heapq.heappop(listHeaps[0])

        result.append(minimum)

        listHeaps.pop(0)

        if flag == 1:

            listHeaps.insert(0, heapLeft)

            listHeaps.insert(0, heapRight)

    return result

data = [3,5,1,6,2,3,4,6,7]

print(smoothSort(data))

data = [42, 33, 17, 5, 29, 10]

print(smoothSort(data))

### 3. 성능 비교 실험

랜덤한 숫자 5000개

A graph with blue bars

AI-generated content may be incorrect.

array = [randint(-5000, 5000) for i in range(5000)]

def: sorted(), 최소 실행 시간: 0.00100초

def: countSort(), 최소 실행 시간: 0.00300초

def: introSort(), 최소 실행 시간: 0.01400초

def: quickSort(), 최소 실행 시간: 0.01800초

def: mergeSort(), 최소 실행 시간: 0.02400초

def: smoothSort(), 최소 실행 시간: 0.03200초

def: timSort(), 최소 실행 시간: 0.03500초

def: heapSort(), 최소 실행 시간: 0.03500초

def: insertSort(), 최소 실행 시간: 1.59100초

def: bubbleSort(), 최소 실행 시간: 2.55200초

거꾸로 정렬된 크기 5000의 배열 (Descending)

A graph with blue bars

AI-generated content may be incorrect.

array = [i for i in reversed(range(5000))]

def: sorted(), 최소 실행 시간: 0.00000초

def: countSort(), 최소 실행 시간: 0.00200초

def: introSort(), 최소 실행 시간: 0.00600초

def: quickSort(), 최소 실행 시간: 0.01400초

def: mergeSort(), 최소 실행 시간: 0.01500초

def: timSort(), 최소 실행 시간: 0.02600초

def: smoothSort(), 최소 실행 시간: 0.02800초

def: heapSort(), 최소 실행 시간: 0.02900초

def: insertSort(), 최소 실행 시간: 2.73700초

def: bubbleSort(), 최소 실행 시간: 3.69000초

정렬된 크기 5000 배열 (Ascending)

A graph with blue squares

AI-generated content may be incorrect.

array = [i for i in range(5000)]

def: sorted(), 최소 실행 시간: 0.00000초

def: insertSort(), 최소 실행 시간: 0.00100초

def: countSort(), 최소 실행 시간: 0.00200초

def: introSort(), 최소 실행 시간: 0.00600초

def: timSort(), 최소 실행 시간: 0.01300초

def: mergeSort(), 최소 실행 시간: 0.01300초

def: quickSort(), 최소 실행 시간: 0.01400초

def: smoothSort(), 최소 실행 시간: 0.03100초

def: heapSort(), 최소 실행 시간: 0.03600초

def: bubbleSort(), 최소 실행 시간: 1.08400초

크기 5000의 부분 정렬된 랜덤 배열

A graph with blue bars

AI-generated content may be incorrect.

array4 = [randint(-5000, 5000)

for i in range(5000//3)] + \

[i for i in range(5000//3)] + \

[1] + \

[i for i in reversed(range(5000//3))]

def: sorted(), 최소 실행 시간: 0.00000초

def: countSort(), 최소 실행 시간: 0.00300초

def: introSort(), 최소 실행 시간: 0.01200초

def: quickSort(), 최소 실행 시간: 0.01300초

def: mergeSort(), 최소 실행 시간: 0.01800초

def: timSort(), 최소 실행 시간: 0.02100초

def: heapSort(), 최소 실행 시간: 0.03100초

def: smoothSort(), 최소 실행 시간: 0.03100초

def: insertSort(), 최소 실행 시간: 1.31700초

def: bubbleSort(), 최소 실행 시간: 2.35200초

4. 성찰하기

* 배운 점 정리  
  레오나르도 수열을 이용해 힙을 여러 개의 레오나르도 트리 포레스트로 구성함으로써, 입력이 거의 정렬된 경우에 선형 시간 성능을 얻을 수 있다는 점을 배웠다. 스무스 정렬은 전통적인 힙 정렬과 마찬가지로 제자리 정렬(in-place)을 유지하면서도, 레오나르도 트리 합병과 분할 과정에서 “루트 간 크기 비교(bubble-up) + 내부 힙화(filter)”를 반복하여 힙 속성과 전체 루트 순서를 동시에 유지한다는 사실을 이해했다. 합병(merge) 조건은 “orders 스택에 저장된 마지막 두 트리의 순서가 연속(k-1, k-2)일 때”에만 발생하며, 이 규칙이 레오나르도 수열 구조를 기반으로 힙을 확장·축소하는 핵심 메커니즘임을 깨달았다. 트리 내부 필터링 작업(하향 힙화)에서는 레오나르도 트리의 자식 인덱스가 “왼쪽: 루트 바로 아래, 오른쪽: 루트 바로 아래에서 L[k−2]만큼 떨어진 위치”라는 특징을 활용해 효율적으로 구현할 수 있음을 배웠다. 시간 복잡도 측면에서, 이미 정렬된 입력에 대해서는 힙 구축 단계에서 루트 교환이나 내부 힙화가 거의 발생하지 않아 최선 케이스 O(n)을 달성하지만, 평균·최악의 경우에는 O(n log n)으로 힙 정렬과 동일하다는 점이 핵심 교훈이다. 스무스 정렬은 안정 정렬이 아니므로, 같은 값을 가진 원소들의 상대적 순서가 보장되지 않는다. 이는 힙 기반 정렬의 일반적인 한계이기도 하다. 실제 구현 시에는 “레오나르도 수열 생성 → 서브힙 분할 → 트리 합병/분할 → 루트 간 교환 후 내부 힙화”라는 일련의 단계마다 정확한 인덱스 계산과 조건 검사가 필수적이며, 그렇지 않으면 배열 경계를 벗어나거나 힙 속성이 깨질 수 있다는 점을 배웠다. 파이썬 heapq를 활용한 간단 구현은 개념 이해에 도움이 되지만, 언어 수준의 표준 라이브러리를 통하지 않고 C/C++ 등으로 직접 구현할 경우 상수 계수를 크게 줄여 실제 성능 이점을 체감할 수 있다. 퀵 정렬·병합 정렬·힙 정렬과 비교했을 때, 스무스 정렬은 이미 정렬된 데이터나 거의 정렬된 데이터에서는 이점이 있지만, 랜덤 데이터에서의 상수 오버헤드 때문에 실용적 효율은 다른 정렬에 비해 상대적으로 낮을 수 있다는 사실을 알게 되었다. 종합적으로, 스무스 정렬은 “어댑티브 정렬”의 한 예시로서 이론적으로 흥미로운 특성을 갖고 있으나, 구현 복잡성과 상수 계수로 인해 실무에서는 드물게 사용된다는 점을 이해하게 되었다.

### 5. 참고자료

* Cho, Seokwon. “Explanation of the Smooth Sort Algorithm.” Tistory Blog, May 17, 2024. <https://choiseokwon.tistory.com/217> (accessed June 1, 2025).
* Dijkstra, Edsger W. *Smoothsort, an alternative for sorting in situ (EWD 796a)*. 1981.
* “Smoothsort.” *Wikipedia, The Free Encyclopedia*. Last modified May 29, 2025. <https://en.wikipedia.org/wiki/Smoothsort> (accessed June 1, 2025).
* Infogalactic. “Smoothsort.” Last modified April 10, 2025. https://infogalactic.com/info/Smoothsort (accessed June 1, 2025).
* Buitinck, Lars. “200 lines of Smoothsort.” GitHub, 2023. <https://github.com/larsbs/200-lines-of-smoothsort> (accessed June 1, 2025).
* “ Leonardo Numbers.” *MathWorld – A Wolfram Web Resource*. Last modified August 3, 2024. http://mathworld.wolfram.com/LeonardoNumbers.html (accessed June 1, 2025).