Hands-on assignment 1 introduction

2271056 정은채

1-1: Quick Sort

Step1: Implement two types of quick sort algorithm

O(n^2) : pcode1_1_1.txt, ha1_1_1.py

O(nlogn) : pcode1_1_2.txt, ha1_1_2.py

에서 확인 가능합니다.

- 수행 예시

input : [21, 3, 12, 15, 7, 32, 4, 25, 9, 18]

output : [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]

(두 개 코드 결과값 동일)

- 소요 시간

O(n^2) : 0.00007 sec

O(nlogn) : 0.00002 sec

Step2: Analyze the time complexities of your implementations

O(n^2) :
기본 퀵 정렬은 배열의 첫 번째 원소를 피벗으로 사용합니다. 배열을 분할하는 데에 O(n)의 시간이 걸립니다. 이 분할이 재귀적으로 이루어지므로, 최악의 경우(피벗이 항상 최소값이나 최대값일 때) 각 단계에서 하나의 원소만 제외되고, 나머지 모든 원소들이 다른 하위 배열로 이동합니다. 이는 각 분할 단계마다 거의 전체 배열을 다시 처리해야 함을 의미합니다.

최악의 경우 시간 복잡도는 다음과 같이 계산됩니다:

- 첫 번째 분할: O(n)

- 두 번째 분할: O(n-1)

- ...

- 마지막 분할: O(1)

이를 모두 합치면 O(n)+O(n−1)+...+O(1)=O(n(n+1)2)=O(n2)O(n)+O(n−1)+...+O(1)=O(2n(n+1))=O(n^2)이 됩니다.

O(nlogn) :

최적화된 퀵 정렬은 배열의 마지막 원소를 피벗으로 사용하며, Lomuto 분할 방식을 사용합니다. 이 방식은 배열을 효과적으로 분할하여 평균적으로 두 개의 거의 동일한 크기의 하위 배열을 생성합니다. 평균적으로 각 분할이 O(n) 시간이 걸리고, 퀵 정렬이 균형잡힌 방식으로 배열을 분할한다면 재귀의 깊이는 log(n)이 됩니다. 따라서 평균 시간 복잡도는 O(nlogn)입니다.

Step3: Explain why one type performs more efficiently than the other

기본 퀵 정렬과 최적화된 퀵 정렬의 효율성 차이는 주로 피벗 선택과 분할 방법에 있습니다.

<피벗 선택>

기본 퀵 정렬에서는 배열의 첫 번째 요소를 피벗으로 사용합니다. 이 방식은 배열이 이미 정렬되어 있거나 거의 정렬된 상태일 때 최악의 성능을 보이는데, 그 이유는 피벗이 최소값이나 최대값이 되어 매번 가장 불균형한 분할을 하게 되기 때문입니다. 이 경우, 분할로 인해 한 쪽은 비게 되고 다른 쪽에는 거의 모든 원소가 남게 되어 O(n^2)의 시간 복잡도를 갖게 됩니다.

최적화된 퀵 정렬은 일반적으로 배열의 끝에 있는 요소를 피벗으로 사용하거나(로무토 분할), 다른 전략(예: 중간값, 무작위 선택, 혹은 중앙값의 중앙값 등)을 사용하여 피벗을 선택합니다. 이는 분할이 더 균형을 이루게 도와주며, 평균적으로 배열을 반으로 나누어 재귀의 깊이를 log(n)으로 유지합니다. 이에 따라 평균 시간 복잡도는 O(n log n)이 됩니다.

<분할 방법>

기본 퀵 정렬은 재귀적으로 더 작은 서브어레이를 만들어나가는 반면, 최적화된 퀵 정렬은 Lomuto 분할처럼 효율적인 분할 방법을 사용해 배열을 더 균등하게 나눕니다. 이런 분할은 더 적은 재귀 호출을 필요로 하며, 결과적으로 더 적은 처리 시간이 필요합니다.

<최악의 시나리오 회피>

최적화된 퀵 정렬은 더 균형 잡힌 분할을 위해 다양한 전략을 사용하여 최악의 경우를 피합니다. 이는 평균적으로 더 나은 성능을 보장하며, 특히 대규모 데이터셋에서 성능 차이가 두드러집니다.

<메모리 사용>

최적화된 퀵 정렬은 인플레이스(in-place)로 작동하며 추가 메모리를 거의 사용하지 않습니다. 기본 퀵 정렬의 경우 더 작은 값, 같은 값, 더 큰 값 리스트를 새로 생성하기 때문에 더 많은 메모리를 사용하게 됩니다. 메모리 사용이 적은 알고리즘은 일반적으로 캐시 활용이 더 효율적이며 성능이 더 좋습니다.

이러한 이유들로 인해, 최적화된 퀵 정렬이 기본 퀵 정렬보다 더 효율적으로 수행됩니다. 평균적인 경우와 대부분의 실제 시나리오에서 최적화된 알고리즘이 더 나은 성능을 나타내며, 특히 데이터가 무작위로 분포되어 있을 때 더욱 그렇습니다.

1-2: Find minimum number of US postage stamps

Step1: Implement greedy algorithm which gives us the minimum count of postage stamps needed to send the post

pcode1_2.txt, ha1_2.py

에서 확인 가능합니다.

Step2: Analyze the time complexity of your implementation

그리디 알고리즘의 구현은 주로 두 가지 주요 단계로 구성됩니다. 정렬과 그리디 선택입니다.

〈정렬〉

우표 가격을 내림차순으로 정렬하는 작업이 필요합니다. 이 작업은 일반적인 정렬 알고리즘을 사용할 때 O(nlogn)의 시간 복잡도를 가집니다. 여기서 n은 우표의 종류 수, 즉 우표 가격 목록의 길이입니다.

〈그리디 선택〉

정렬된 우표 목록을 순회하면서 각 우표를 가능한 많이 사용합니다. 각 우표에 대해, 우표 가격이 남은 금액보다 작거나 같은 동안 우표를 선택하고 남은 금액에서 해당 우표 가격을 빼줍니다. 이 단계의 최악의 경우 시간 복잡도는 O(m)이 될 수 있는데, 여기서 m은 목표 금액입니다. 최악의 경우는 모든 우표가 1센트짜리 우표만으로 구성되어 있을 때입니다.

결론적으로, 그리디 알고리즘의 시간 복잡도는 정렬의 O(nlogn)과 그리디 선택의 O(m)을 합한 것이므로, 총 시간 복잡도는 O(nlogn+m)가 됩니다. 이는 n과 m에 따라 달라지지만, 일반적으로 nlogn항이 지배적인 영향을 미칩니다.

이러한 분석을 통해 그리디 알고리즘이 우표 문제를 해결하기 위한 효율적인 접근 방식이라는 것을 알 수 있지만, 이 문제에 대해 최적의 해를 항상 보장하지는 않는다는 것도 명심해야 합니다. 예를 들어, 큰 단위의 우표를 사용하는 것이 즉각적으로 최적처럼 보일 수 있지만, 때로는 더 많은 수의 우표를 사용하게 될 수도 있기 때문입니다.

Step3: Show that your implementation does NOT provide an optimal solution by comparing the solution returned by the implementation with the optimal one that can be manually found

그리디 알고리즘 구현이 최적의 해를 제공하지 않음을 보이기 위해서는 구현에 의해 반환된 해답과 수동으로 찾을 수 있는 최적의 해답을 비교해야 합니다.

그리디 알고리즘은 가장 큰 우표부터 사용하여 남은 금액을 채워 나가는 방식입니다. 이 경우 140센트를 맞추기 위해 [100, 34, 1, 1, 1, 1, 1, 1]의 우표를 선택했습니다. 총 8개의 우표를 사용했습니다.

하지만 수동으로 최적의 해를 찾아보면, [70, 70] 두 개의 우표만으로도 140센트를 정확히 맞출 수 있음을 알 수 있습니다. 이 경우에는 단 2개의 우표만 사용하면 됩니다.

이는 더 작은 동전의 값의 배수가 더 큰 동전의 값이 되지 않아서 발생되는 문제입니다.

이 예시를 통해 그리디 알고리즘 구현이 항상 최적의 해를 제공하지 않음을 알 수 있습니다. 구현된 알고리즘은 각 단계에서 지역적으로 최선의 선택을 하지만, 그 선택이 전역적인 최소 우표 사용 수로 이어지지 않을 수 있기 때문입니다. 이 문제는 우표 가격의 구성에 따라 그리디 알고리즘이 최적의 해를 찾지 못하는 경우가 있을 수 있으며, 그 결과로 더 많은 우표를 사용할 수도 있음을 시사합니다.

1-3: improved in-place merge sort algorithm

Step1: Implement an improved version of in-place merge sort algorithm

pcode1_3.txt, ha1_3.py

에서 확인 가능합니다.

Step2: Analyze the time and space complexity of your implementation

〈시간 복잡도〉 - O(nlogn)

이 알고리즘의 시간 복잡도는 기본적인 병합 정렬과 유사하게 분할 단계에서 O(logn)입니다. 각 분할에서는 배열을 반으로 나누고, 이 과정은 총 logn번 일어납니다. 병합 단계에서는 각 레벨에서 최대 O(n)의 시간이 소요될 수 있습니다.

따라서 재귀의 각 레벨에서 대략 O(n)의 시간이 걸리고, logn 레벨이 있으므로, 전체 시간 복잡도는 O(nlogn)이 됩니다.

하지만 이 구현은 두 정렬된 배열을 재배치하는 과정에서 각 요소가 한 위치씩 이동하므로, 최악의 경우 각 병합 단계에서 O(n^2)의 시간이 소요될 수 있습니다. (O(n^2logn) 가능)

〈공간 복잡도〉 - O(1)

이 알고리즘의 공간 복잡도는 O(1)입니다. 병합 정렬은 일반적으로 분할 단계에서 임시 배열을 사용하므로 공간 복잡도가 O(n)이 되지만, 제자리 병합 정렬 알고리즘은 추가 공간을 사용하지 않습니다. 병합 단계에서는 원본 배열 내에서만 원소들을 이동시키기 때문에, 추가적인 공간을 필요로 하지 않습니다.

Step3: Briefly explain your strategies about how to improve the time complexity

시간 복잡도를 개선하기 위해 고려할 수 있는 몇 가지 전략이 있습니다. 이러한 전략들은 in-place 병합 정렬의 병합 단계에서 요소들을 이동하는 방식을 최적화하는 것에 초점을 맞춥니다

1. 블록 교환 (Block Swap):

병합 과정에서 요소들을 한 칸씩 이동시키는 대신에, 더 큰 블록 단위로 원소들을 교환할 수 있습니다. 원소들을 이동시키는 총 횟수를 줄여서 병합 과정의 시간 복잡도를 개선할 수 있습니다.

2. 불필요한 복사 방지:

병합할 때 모든 원소를 복사하지 않고, 실제로 위치를 변경해야 하는 원소들만을 이동시키는 것으로 불필요한 작업을 줄일 수 있습니다.

3. 탐색 기반 위치 이동 (Binary Search & Rotation):

두 부분 배열의 병합 시, 첫 번째 부분 배열의 원소가 두 번째 부분 배열 내에서 들어갈 위치를 이진 탐색을 사용해 빠르게 찾아내고, 그 위치로의 블록 회전을 통해 원소를 이동시킬 수 있습니다.

4. 비교 및 이동 최적화:

병합 과정 중에 각 부분 배열에서 가장 큰 원소와 가장 작은 원소를 비교하여, 병합이 필요한 부분만을 타겟으로 하여 불필요한 비교와 이동을 피합니다.

5. 하이브리드 접근법:

작은 배열에 대해서는 병합 정렬 대신 더 빠른 정렬 알고리즘(예: 삽입 정렬)을 사용하고, 재귀의 깊이가 일정 수준에 도달하면 더 이상 분할하지 않고 이러한 대체 알고리즘을 적용할 수 있습니다.

실제로 이러한 최적화를 적용하려면 알고리즘의 구현이 더 복잡해질 수 있으므로, 성능 향상이 반드시 필요한 상황에서만 고려해야 합니다.