자료구조 과제 4 보고서

2021-16621 컴퓨터공학부 김민수

Skeleton 수정 사항

제공된 Skeleton에서는 Do___Sort 함수의 인자로 배열을 받고, 정렬한 배열을 return하는 방식을 활용했으나, Java가 배열을 인자로 받을 때, 배열의 reference를 받는다는 점을 활용하여, 따로 배열을 return하는 구문을 제거했습니다.

각 정렬 알고리즘의 동작 방식

a. Bubble Sort
   배열 내의 i와 i+1쌍의 비교와 swap을 통해, 가장 큰 값을 배열의 오른쪽 끝에 고정시킵니다. 고정된 값을 제외한 구간에서 이를 반복하여 정렬합니다.

b. Insertion Sort
   배열의 1번 요소부터 n-1번 요소까지 올바른 위치에 삽입하여 정렬합니다.
   기존 배열의 요소 값이 삽입하고자 하는 값보다 큰 경우 이를 우쉬프트하고, 이를 반복하여 올바른 위치를 찾고 삽입합니다.

c. Heap Sort
   배열에 BuildHeap을 수행하여 maxHeap을 구성합니다.
   maxHeap의 0번 원소와 마지막 원소를 swap, 마지막 원소를 제외한 구간에서 0번 원소에 대해 percolateDown 수행, 반복하여 정렬합니다.

d. Merge Sort
   중간 값(mid)를 설정하여, 이를 기준으로 나눈 배열에 대해 재귀적으로 merge sort를 수행합니다. 이후 나눈 배열을 merge하여 정렬합니다. 이때 추가 저장 공간을 사용합니다.

e. Quick Sort
   Randomized Pivot Quick Sort 방식으로 구현했습니다. Partition을 수행하기 전, 가장 오른쪽 끝 원소를 random하게 배치하여, pivot을 randomize 했습니다. 이후 Partition을 진행하고, pivot을 기준으로 우측, 좌측 배열에 재귀적으로 Quick Sort를 수행합니다. 배열의 중복 비율이 높은 경우에도 효율적으로 작동하는 Three-Way Partitioning Quick Sort algorithm도 존재하지만, 중복 비율과 Quick Sort의 성능이 반비례한다는 사실을 보이는 과제의 목적과는 맞지 않다고 판단하여, Randomized Pivot 방식으로 구현했습니다.

f. Radix Sort
   입력된 배열을 음수 배열과 양수(0 포함) 배열로 나눕니다. 음수 값들에 '-'를 취해, 양수로 바꿔서 배열에 저장합니다. 이후 양수 배열은 오름차순으로, 음수 배열은 내림차순으로 정렬합니다. 음수 배열 값들에 다시 '-'를 취해서 원래 값으로 변환하고, 정렬된 음수, 양수 배열을 합쳐서 전체 정렬을 완성합니다.
   정렬 과정에서 FindMaxDigit 함수를 통해 가장 큰 자릿수를 찾고, 각 자릿수 별로 counting sort를 반복하여 정렬을 진행합니다. 누적합의 방향을 통해 오름차순, 내림차순 정렬을 결정합니다.

동작 시간 분석 (Data 개수에 따른 분석, 이외의 요인은 Randomize하여 영향을 최소화)

Data의 개수, 즉 배열의 길이에 따른 정렬 시간을 분석했습니다. 과제 Spec에서 주어진 r command를 활용했으며, 배열 길이가 10~1000000인 case에 대해서 각각 분석을 진행했습니다. R command를 통해 입력 받은 numsize, rmin, rmax 인자를 SortingTimeTest 함수에 넘겨주어 각 case에 대한 sorting time을 측정했는데, 진행 절차는 다음과 같습니다.

SortingTimeTest function 설명
rmax, rmin 범위에서 무작위로 추출한 numsize 크기의 int 배열을 생성합니다. 모든 Sort를 실행하고, 각 Sort의 수행 시간을 SortAndTime Map에 저장합니다. 위 과정을 10번 반복하여, 각 Sort의 평균 수행시간을 구합니다.
Rmax와 rmin은 (-1073741823, 1073741823)로 설정하여, r command가 제공하는 가능한 가장 넓은 범위를 사용했습니다. 이는 가능한 일반적인 상황에서의 정렬 성능을 분석하기 위해 사용한 방법입니다. (특정 자릿수, 중복 비율, 사전 정렬 비율 고려 X).
추가적으로, JIT complier의 특성으로 인한 sort time 변화를 방지하기 위해서, build 후 한 번만 실행하고 다시 compile하는 방식으로 측정을 진행했습니다. 길이 10, 100의 배열은 ms가 아닌 ns 단위로 측정했고(ms 단위로는 차이가 보이지 않음), 길이 1000이상의 배열은 Bubble Sort를, 50000이상의 배열은 Insertion Sort를 제외한 나머지 sort의 수행 시간만 측정했습니다. (너무 긴 측정 시간으로 인해서 배제).

각 case에 대해 함수를 3번 실행했으며, 전체의 평균 값으로 분석을 도출했습니다. 아래는 정리된 결과입니다. <세부 자료는 첨부된 excel file을 참고하시길 바랍니다>

	Bubble	Insertion	Heap	Merge	Quick	Radix	
10	3072	1461	3860	5096	7872	107508	(ns)
100	181630	56043	27353	33416	35056	1002373	
1,000		0.24	0.187	0.233	0.065	0.601	(ms)
10,000		11.46	1.5	1.46	0.66	2.57	
50,000			6.43	7.57	4.53	7.47	
100,000			11.73	13.93	7.87	11	
500,000			72.1	64.5	46.37	45.87	
1,000,000			162	145.8	97.06	98.33	

길이 10인 배열에서는 Insertion sort가, 길이 100인 배열은 Heap sort가 가장 우세한 성능을 보였습니다. 이후, 길이 1,000부터 100,000까지의 배열에 대해서는 Quick Sort가 가장 우세했으며, 길이 500,000과 1,000,000인 배열에 대해서는 Quick과 Radix가 동일한 수준의 성능을 보였습니다.

예상대로 Insertion Sort는 길이가 짧은 case에서 우세한 성능을 보였고, Quick Sort는 대부분의 case에서 가장 좋은 성능을 보였습니다. 한 가지 놀라운 점은, Radix Sort가 짧은 배열에서는 저조한 성능을 보였으나, 긴 배열에서는 우세한 성능을 보였다는 점입니다. Radix Sort가 양수와 음수를 별도로 처리하고, 정렬 과정에서 추가적인 배열(메모리)을 사용하는 동작의 오버헤드가 커서 이러한 결과가 나왔다고 추론했습니다.

하이퍼 파라미터를 찾기 위해 별도로 구현한 함수, 기능

Search 함수의 올바른 하이퍼 파라미터를 찾기 위해, 별도로 SortingPerformanceTest.java와 Python의 sklearn tree module을 사용했습니다. <코드는 제출물에 첨부되어 있습니다, Chat GPT 참고하여 구현> 해당 Class와 데이터 분석의 작동 방식은 다음과 같습니다.

배열의 길이, 사전 정렬 비율, 중복 비율, 최대 자릿수, 4가지 변수에 관해 다양한 분포를 가진 int array를 만들어, 해당 배열을 정렬하는 가장 빠른 Sort Algorithm을 csv file에 기록합니다. 이때 해당 배열의 특성도 함께 기록합니다. Csv file에 기록한 data Set을 바탕으로, decision tree model을 생성하여 데이터 분석을 통해 적절한 하이퍼 파라미터를 탐색했습니다.

배열의 특성을 파악하기 위한 함수들

FindMaxDigit: 배열 속의 가장 큰 자릿수를 return하는 함수

FindDuplicateFraction: 배열의 중복 비율을 return하는 함수

배열 속에 들어있는 element와 그 빈도를 HashMap에 기록하고 element의 빈도가 2 이상인 경우, 빈도에서 1을 뺀 값을 모두 더하여 중복 비율 측정

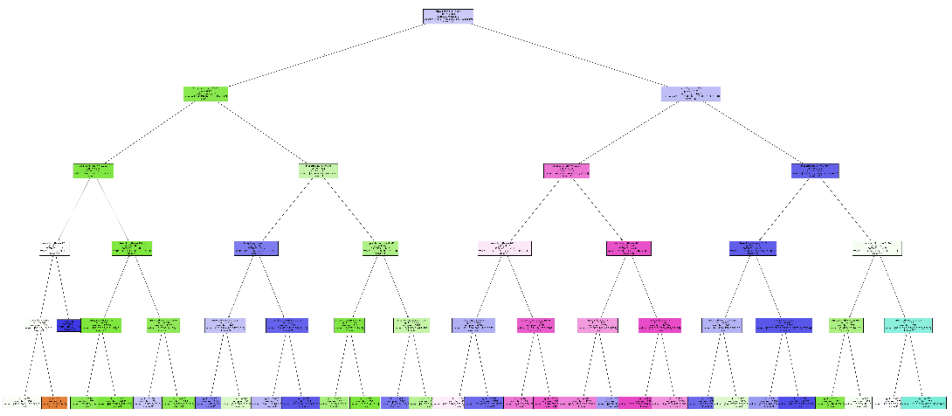
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]과 같은 배열에 대해 중복 비율이 5/10 (50%)가 되도록 기준 설정

각 정렬 성능에 미치는 영향을 나타내기 위한 가장 적절한 중복 비율 계산 방식이라고 생각해서, 해당 방법으로 중복 비율을 계산했습니다.

FindPreSortedFraction: 배열의 i와 i+1 쌍을 비교하여 정렬된 정도를 return 하는 함수

100%면 오름차순 정렬, 0%면 내림차순 정렬을 뜻함. 비율이 50%일 때, 사전 정렬 비율이 가장 낮으며, 중복 비율과 사전 정렬 비율은 종속적임. (동일한 원소를 일렬로 세운 것은 사전 정렬된 것으로 판단되기 때문)

Decision Tree의 분석 결과



생성한 약 10만 개의 Data Set을 바탕으로 분석을 진행한 결과 위 그림의 Decision Tree가 도출되었습니다. Max Depth를 5로 설정하여 꽤나 복잡한 그림이 도출되었는데, 이로 인해 실제로 조건 분기를 분석할 때는 텍스트 파일 형태로 변환하여 분석을 진행했습니다. 복잡함에도 Depth를 5로 설정한 이유는 5일 때 가장 정확도가 높게 나왔기 때문입니다. <원본 Tree 이미지와 Text file, Data Set을 정리한 Csv file은 첨부 파일을 참고하시길 바랍니다>

Search 함수의 하이퍼 파라미터 설정 기준 요약

실제 Search 함수의 하이퍼 파라미터는 도출된 Decision Tree의 값을 따라 설정했는데, 조건 분기 내에서도 너무 작거나 무시해도 되는 수준의 분기는 제거했으며, 소수점을 제거한 간소화된 parameter를 설정했습니다. 각각의 조건을 간단하게 요약하자면 다음과 같습니다.

배열 내 element의 최대 자릿수

중복 비율 89% 이하에 대해
배열의 길이가 410~14600, 최대 자릿수가 3 이하면 Radix Sort
배열의 길이가 14600 이상, 최대 자릿수가 4 이하면 Radix Sort

중복 비율 89% 초과에 대해
배열의 길이가 410~3046, 최대 자릿수가 2 이하면 Radix Sort
배열의 길이가 3046 이상, 최대 자릿수가 4 이하면 Radix Sort

세부 조건이 많이 포함되어 있긴 하지만, 대략 최대 자릿수가 4 이하인 경우에 대해 Radix Sort가 가장 좋은 성능을 보인다는 점을 확인할 수 있습니다.

중복 비율에 대한 구분
높은 중복 비율을 보이는 배열에 대해서는, 이론적 직관을 따르면 Heap Sort 혹은 Insertion Sort가 유리해야 하며, Quick Sort가 불리해야 합니다. 하지만, 이번 분석에서 중복 비율에 따라 좋은 성능을 보이는 Sort에 대한 해답은 찾을 수 없었습니다. 아마 다음의 2가지 요인으로 인해 해당 결과가 나타났다고 생각합니다.

1. 중복 비율 계산 방식이 정확한 중복 비율을 표현하지 못한 점
2. 사전 정렬 비율과 종속적인 관계의 중복 비율

이번 과제에 사용한 중복 비율의 계산 방식은, 실제로 중복된 원소가 많은 배열의 중복 비율을 높게 책정하지만, 그렇지 않은 경우에 대해서도 비율을 높게 책정하는 문제점이 있습니다.
예시로 [1, 1, 2, 2, 3, 3]과 [1, 1, 1, 2, 3, 4]의 중복도를 모두 50%로 부여하지만, 실제 Quick/Heap/Insertion의 성능에 영향을 더 많이 미치는 것은 후자입니다. 이와 같은 case로 인해, 정확한 중복도 계산이 이루어지지 않아, 중복 비율에 대한 구분이 나타나지 않았다고 추론했습니다. 또, 중복도가 높으면, 대체적으로 사전 정렬 비율이 높아지는데, 이로 인한 영향도 있을 것이라 추론했습니다.

사전 정렬 비율에 대한 구분
길이 244 이하, 사전 정렬 비율 39% 이상이면 Insertion Sort
길이 244~410, 사전 정렬 비율 60% 이상이면 Insertion Sort
길이 410~1100, 최대 자릿수 5 이상, 사전 정렬 비율 75% 이상이면 Insertion Sort
길이 410 이상, 최대 자릿수 5 이상, 사전 정렬 비율 99% 이상이면 Insertion Sort

배열의 길이가 짧을 때는 사전 정렬 비율이 조금만 높아도 Insertion Sort가 좋은 성능을 보이지

만, 배열의 길이가 길 때는 더 높은 사전 정렬 비율을 보여야 Insertion sort의 성능이 좋아지는 것을 확인할 수 있습니다. 결국, 사전 정렬 비율이 높으면 Insertion Sort가 유리하다는 이론적 직관을 실제로 관찰했습니다.

나머지 조건

길이가 178 이하, Insertion Sort (짧은 길이의 배열에는 Insertion 이 유리함)
위에서 언급하지 않은 나머지 모든 경우에는 Quick Sort 가 가장 빠르다는 결론을 얻었습니다.
이는 중복 비율에 대한 Sorting time 을 정확히 반영하지 못한 결과이기 때문에, 정확하지 않을 수 있습니다.

Search 와 모든 정렬을 사용한 시간의 비교

CompareTime, SortTime, generateArray 함수를 활용해서, Search 함수의 parameter 를 기준으로 가장 빠른 Sort 를 찾는 시간과, 실제로 모든 Sort 를 수행하여 가장 빠른 Sort 를 찾는 시간을 비교했습니다. 아래는 다양한 특성의 배열에 대해 함수를 실행한 결과들입니다.

Search 의 수행 시간이 모든 Sorting 을 진행하는 것보다 훨씬 짧은 것을 확인할 수 있습니다.
아쉽게도 정답 비율이 아주 높게 나타나지는 않았는데, 이는 위에서 분석한 요인들로 인한 것이라 생각됩니다.

```
c
DoSearch performance time 38914100ns, Fastest sort found by Search is Q
Doing all sort performance time 1319800100ns, Fastest sort found by actual sorting is Q
```

```
c
DoSearch performance time 32001700ns, Fastest sort found by Search is Q
Doing all sort performance time 1236635100ns, Fastest sort found by actual sorting is Q
```

```
c
DoSearch performance time 27022300ns, Fastest sort found by Search is I
Doing all sort performance time 117256300ns, Fastest sort found by actual sorting is I
```