2018년도 가을학기 SCSC 알고리즘 실습 6주차

문현수, munhyunsu@cs-cnu.org

Thursday October 11, 2018

1 개요

정렬은 순서가 있는 '모음'에서 기준에 맞게 순서를 변경해주는 것을 말한다. 예컨대 가나다 순으로 '정렬'되어있는 사전이나 숫자로 순서가 표시된 설명서 등이 있다. 만일 이러한 정보가 기준없이 나열되어 있다면 원하는 정보를 찾기 굉장히 어려울 것이다. 이처럼 정렬이 되어있지 않으면 많은 불편을 초래하기 때문에 정렬은 실생활에서 자연스레 사용되고 있다.

일상 생활에 자연스레 녹아있기 때문에 '정렬'은 프로그래밍에서도 전통적인 알고리즘 문제로 꼽힌다. 굉장히 많은 정렬 '알고리즘'이 있으며, 각각의 정렬들은 그 특징이 있기 때문에 적재적소에 활용된다. 즉, 시간복잡도와 공간복잡도가 낮은 정렬 뿐 아니라 성능은 조금 나쁘더라도 특정한 상황에서 더 유용하게 활용되는 정렬 방법이 있다. 기준에 맞게 순서를 변경한다는 결과물은 같지만 연산하는 방법에 따라서 성능/활용 가능 영역이 달라지기 때문에 알고리즘을 공부하는 우리는 꼭 배워야한다.

Python 3에는 이미 '정렬'이 구현되어 있다. 따라서 '순서'와 '중복'이 있는 자료형인 `List`에 `sort()` 혹은 `sorted()`를 호출하여 기준대로 나열할 수 있다. 하지만 우리는 '정렬 알고리즘'을 공부하는데에 목적이 있으므로 내부에 구현되어 있는 도구를 사용하지 않고, `List`에 대한 정렬을 학습한다. 우리는 **Bubble sort**, **Selection sort**, **Insertion sort**, **Merge sort**, **Quick sort** 총 5개의 정렬 알고리즘을 살펴본다. 그리고 마지막으로 각 정렬의 성능을 측정하여 그래프로 출력하고 비교한다.

2 Sorting Algorithms

2.1 Bubble Sort

거품 정렬은 마치 거품 안에서 연산이 이루어지는 것 같다는 의미로 명명되었다. 거품 안에 앞뒤 총 2개의 원소를 넣어두고 기준에 맞게 정렬이 되었는지 확인한다.

만약 기준에 맞게 정렬되어있지 않다면 그 순서를 바꾼다. 그 후 그 다음 원소 2개를 거품안에 넣고 같은 연산을 반복한다.

Listing 1: Buuble sort 정렬 순서(최적화 버전)

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 정렬 , () 로 거품
    표시
----- 첫번째 반복 -----
[(4, 19), 1, 9, 10]: 위치 유지
[4, (19, 1), 9, 10]: 위치 변경
-> [4, (1, 19), 9, 10]
[4, 1, (19, 9), 10]: 위치 변경
-> [4, 1, (9, 19), 10]
[4, 1, 9, (19, 10)]: 위치 변경
-> [4, 1, 9, (10, 19)]
----- 두번째 반복 -----
[(4, 1), 9, 10, 19]: 위치 변경
-> [(1, 4), 9, 10, 19]
[1, (4, 9), 10, 19]: 위치 유지
[1, 4, (9, 10), 19]: 위치 유지
[1, 4, 9, (10, 19)]: 위치 유지
----- 세번째 반복 -----
[(1, 4), 9, 10, 19]: 위치 유지
[1, (4, 9), 10, 19]: 위치 유지
[1, 4, (9, 10), 19]: 위치 유지
[1, 4, 9, (10, 19)]: 위치 유지
----- 반복동안 위치 변경이 없었으므로 종료 -----
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

위의 예제처럼 2개의 원소를 계속 비교하며 N-1번 반복하여 정렬을 완료할 수 있다. 하지만 약간의 트릭(True / False)을 사용하여 매 반복마다 위치가 변경되었는지 확인하는 것으로 N-1번 반복하기 전에 정렬을 끝낼 수 있다. Bubble sort의 수도 코드는 아래와 같다.

Listing 2: Bubble sort 수도 코드

```
1 procedure bubbleSort( A : list of sortable items )
2     n = length(A)
3     repeat
4         swapped = false
5         for i = 1 to n-1 inclusive do
6             if A[i-1] > A[i] then
```

```
7                   swap(A[i-1], A[i])
8                   swapped = true
9               end if
10          end for
11          n = n - 1
12      until not swapped
13  end procedure
```

처음 배우는 정렬을 수도 코드로만 이해하면 어려우므로 Python 3 코드도 함께 살펴보자. 우리는 아래의 코드에서 `reverse`라는 인자를 활용하여 오름차순과 내림차순을 선택할 수 있음을 배운다. 이번 실습 및 과제에서 배우는 모든 정렬에는 `reverse`라는 인자를 활용하여 구현하는 것이 목표이므로 코드를 잘 읽어보도록 하자. 또한 Python 3에서 _(언더바)로 시작하는 함수는 '비공개' 함수로 간주하므로 `_swap()`과 같이 내부에서만 활용하는 함수에 활용하도록 하자.

Listing 3: bubble.py

```
1  def _swap(input_list, index_a, index_b):
2      temp = input_list[index_a]
3      input_list[index_a] = input_list[index_b]
4      input_list[index_b] = temp
5
6
7  def bubble_sort(target, reverse=False):
8      swapped = True
9      while swapped:
10         swapped = False
11         for index in range(0, len(target)-1, 1):
12             if reverse:
13                 if target[index] < target[index+1]:
14                     _swap(target, index, index + 1)
15                     swapped = True
16             else:
17                 if target[index] > target[index+1]:
18                     _swap(target, index, index + 1)
19                     swapped = True
20
21
22 def main():
23     list01 = ['z', 'y', 'x', 'w', 'v',
24               'u', 't', 's', 'r', 'q',
25               'p', 'o', 'n', 'm', 'l',
```

```
26                  'k', 'j', 'i', 'h', 'g',
27                  'f', 'e', 'd', 'c', 'b',
28                  'a']
29      bubble_sort(list01)
30      print(list01)
31      bubble_sort(list01, reverse=True)
32      print(list01)
33
34
35  if __name__ == '__main__':
36      main()
```

2.2 Selection Sort

선택 정렬은 원소를 하나씩 선택하여 '정렬된' 순서로 놓아 문제를 해결한다. 가장 작은 원소를 선택하여 정렬할 수도 있고 가장 큰 원소를 선택하여 정렬할 수도 있으니 원하는 방법으로 구현 가능하다. '정렬된 리스트'에 원소를 하나씩 추가한다는 개념으로 접근하면 이해하기 쉽다. 예제를 통해 살펴보도록 하자.

Listing 4: Selection sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 , 앞에서부터
    정렬
**: 채울 위치 , (): 선택된 원소
[**4, 19, (1), 9, 10]: 위치 변경
-> [**(1), 19, 4, 9, 10]
[1, **19, (4), 9, 10]: 위치 변경
-> [1, **(4), 19, 9, 10]
[1, 4, **19, (9), 10]: 위치 변경
-> [1, 4, **(9), 19, 10]
[1, 4, 9, **19, (10)]: 위치 변경
-> [1, 4, 9, **(10), 19]
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

위의 예제처럼 한칸씩 정렬시켜가며 순서를 맞추는 방법이 **선택 정렬**이다. Python 3로 구현해야하므로 수도 코드도 살펴보자.

Listing 5: Selection sort 수도 코드

```
1  int i,j;
2  int n;
```

```
3  for (j = 0; j < n-1; j++)
4  {
5      int iMin = j;
6      for (i = j+1; i < n; i++)
7      {
8          if (a[i] < a[iMin])
9          {
10             iMin = i;
11         }
12     }
13
14     if (iMin != j)
15     {
16         swap(a[j], a[iMin]);
17     }
18 }
```

2.3 Insertion Sort

삽입 정렬은 선택 정렬과 비슷해보이지만 '위치 변경'을 정렬 될 때까지 계속하며 맞는 위치를 찾아가는 부분이 다르다. 예제를 통하여 살펴보도록 하자. 선택 정렬과 '정렬된 범위'를 늘려간다는 점에서 비슷하지만 위치를 찾는 방법이 다르다. 사소한 차이지만 복잡도 및 코드에서는 큰 차이가 나므로 기억해두자.

Listing 6: Insertion sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순
(): 선택된 원소 , **: 정렬된 리스트의 끝
[(4)**, 19, 1, 9, 10]: 첫 원소이자 정렬된 리스트의 끝 ,
    정렬됨
[4, (19)**, 1, 9, 10]: 위치 변경 시작
-> [4, (19)**, 1, 9, 10]
[4, 19, (1)**, 9, 10]: 위치 변경 시작
-> [4, (1), 19**, 9, 10]
-> [(1), 4, 19**, 9, 10]
[1, 4, 19, (9)**, 10]: 위치 변경 시작
-> [1, 4, (9), 19**, 10]
[1, 4, 9, 19, (10)**]: 위치 변경 시작
-> [1, 4, 9, (10), 19**]
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

삽입 정렬의 수도코드는 다음과 같다.

Listing 7: Insertion sort 수도 코드

```
1  i ← 1
2  while i < length(A)
3      j ← i
4      while j > 0 and A[j−1] > A[j]
5          swap A[j] and A[j−1]
6          j ← j − 1
7      end while
8      i ← i + 1
9  end while
```

2.4 Merge Sort

지금까지의 정렬은 시간 복잡도가 높아서 실제로 활용하기 어려웠다. 하지만 **병합 정렬**은 시간 복잡도가 낮아 활용하기 좋으며, 특히 빅데이터 처리에서 큰 강점을 보인다. 예제로 살펴보도록 하자.

Listing 8: Merge sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순
(): 원소 블락 ,
[(4, 19), (1, 9, 10)]: 재귀 호출
−>[(4), (19)]: 최소 원소 , 합병 시작
−−>[(4, 19)]: 부분 정렬

−>[(1), (9, 10)]: 재귀 호출
−−>[(1)]: 최소 원소 , 합병 시작
−−>[(9), (10)]: 최소 원소 , 합병 시작
−−−>[(9, 10)]: 부분 정렬
−−−−>[(1, 9, 10)]: 부분 정렬

−−−−−>[(1, 4, 9, 10, 19)]: 부분 정렬
[1, 4, 9, 10, 19]: 정렬 상태 ( 출력 )
```

합병 정렬은 재귀적 기법을 사용하기 때문에 예제로 따라가기 굉장히 난해한 알고리즘이다. 수도 코드로 살펴보도록 하자. 2개의 함수로 구성되어 하나는 '분할', 다른 하나는 '합병'을 수행함에 집중하자.

Listing 9: Merge sort 수도 코드

```
1  function merge_sort(list m)
2      if length of m ≤ 1 then
3          return m
4
5      var left := empty list
6      var right := empty list
7      for each x with index i in m do
8          if i < (length of m)/2 then
9              add x to left
10         else
11             add x to right
12
13     left := merge_sort(left)
14     right := merge_sort(right)
15
16     return merge(left, right)
17
18 function merge(left, right)
19     var result := empty list
20
21     while left is not empty and right is not empty do
22         if first(left) ≤ first(right) then
23             append first(left) to result
24             left := rest(left)
25         else
26             append first(right) to result
27             right := rest(right)
28
29     while left is not empty do
30         append first(left) to result
31         left := rest(left)
32     while right is not empty do
33         append first(right) to result
34         right := rest(right)
35     return result
```

2.5 Quick Sort

퀵 정렬은 다른 원소와 비교만으로 정렬을 하는 알고리즘으로 알고리즘 테스트에서 가장 많이 활용되는 알고리즘 중 하나다. 이론적 성능이 가장 좋은 것은 아니지만,

특수한 상황이 아닌이상 무난하게 사용할 수 있기에 실제로도 많이 구현되어 사용된다. 예제를 통해 살펴보도록 하자.

Listing 10: Quick sort 정렬 순서

```
[4, 19, 1, 9, 10]: 초기 상태 ( 입력 ), 오름차순 , 마지막 원소
    Pivot
**: Pivot, M: C: L: Left index, R: Right index, (): 재귀 호출시 그룹
-> [4MC, 19, 1, 9, 10**]: Marker, Checker 교환
-> [4, 19M, 1C, 9, 10**]: Marker, Checker 교환
-> [4, 1, 19M, 9C, 10**]: Marker, Checker 교환
-> [4, 1, 9, 19MC, 10**]:
-> [4, 1, 9, 19M, 10**]: Pivot, Marker 교환
[(4, 1, 9), 10, (19)]: index 4 반환

-> [4MC, 1, 9**]: Marker, Checker 교환
-> [4, 1MC, 9**]: Marker, Checker 교환
-> [4, 1, 9**M]: Pivot, Marker 교환
[(4, 1), 9]: index 2 반환

-> [4MC, 1**]
-> [4M, 1**]: Pivot, Marker 교환
[1, 4]
[1, 4, 9]
[1, 4, 9, 10, 19]
```

Quick sort는 이해하기 굉장히 어려우므로 수도 코드로 잘 살펴보아야 한다. 예제와 코드를 따라가며 살펴보는 것이 필수다.

Listing 11: Quick sort 수도 코드(Lomuto 버전)

```
1  algorithm quicksort(A, lo, hi) is
2      if lo < hi then
3          p := partition(A, lo, hi)
4          quicksort(A, lo, p - 1 )
5          quicksort(A, p + 1, hi)
6
7  algorithm partition(A, lo, hi) is
8      pivot := A[hi]
9      i := lo
10     for j := lo to hi - 1 do
11         if A[j] < pivot then
```

```
12              swap A[i] with A[j]
13              i := i + 1
14      swap A[i] with A[hi]
15      return i
```

위의 수도 코드는 이해하기 쉬운 버전의 **Quick sort** 수도 코드다. 조금 더 효율적인 수도 코드는 아래와 같다.

Listing 12: Quick sort 수도 코드(Hoare 버전)

```
1   algorithm quicksort(A, lo, hi) is
2       if lo < hi then
3           p := partition(A, lo, hi)
4           quicksort(A, lo, p)
5           quicksort(A, p + 1, hi)
6
7   algorithm partition(A, lo, hi) is
8       pivot := A[lo]
9       i := lo - 1
10      j := hi + 1
11      loop forever
12          do
13              i := i + 1
14          while A[i] < pivot
15
16          do
17              j := j - 1
18          while A[j] > pivot
19
20          if i >= j then
21              return j
22
23          swap A[i] with A[j]
```

2.6 성능 비교

각 정렬 알고리즘은 성능에서 큰 차이가 난다. 각 정렬 알고리즘을 구현한 후 주어진 예제 코드를 사용하면 Figure 1와 같은 성능 비교 그래프를 얻을 수 있다.

그림에서 Heap sort는 추후 배울 예정이며 Library는 Python 3에 구현되어 있는 도구의 속도다.

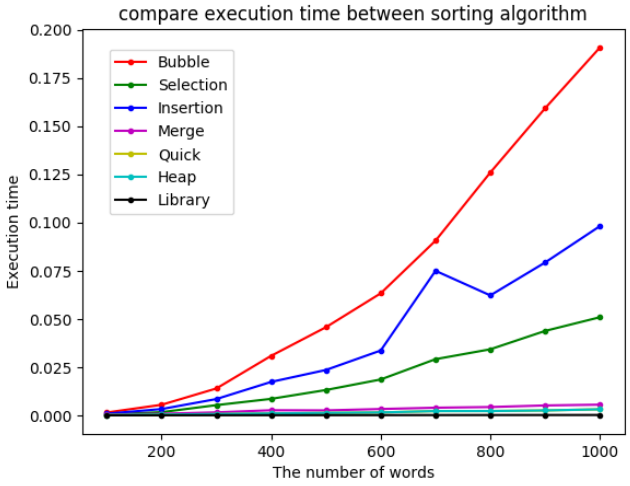

Figure 1: comparison.png

2.7 과제 목표

- 원소 10개 이상 예제 손으로 그려보기(Merge, Quick 필수!, 나머지 선택)
- Bubble, Selection, Insertion, Merge, Quick 구현
- 성능 비교
- (선택) 시간 복잡도, 공간 복잡도 관점으로 성능 비교 그래프 분석

2.8 제출 관련

- 마감 날짜: 2018. 10. 17. 23:59:59
- 딜레이: 1일당 10% 감점(처음 2일까지는 -2)
- 제출 방법: 과목 사이버캠퍼스
- 제출 형식: 과제 리포트 PDF(HWP, DOC 받지 않음!), 소스코드(구현한 .py 만 추가할 것)를 압축한 .zip 파일
- 리포트에 포함해야하는 내용: 목표, 목표를 위해 알아야하는 것, 해결 방법, 결과, (선택)느낀점 or전달할 말
- 제출 파일 제목: AL_201550320_문현수_06.zip(파일명 준수!)

2.9 조교 연락처

- 문현수

- munhyunsu@cs-cnu.org
- 공학5호관 633호 데이터네트워크연구실
- 이메일, 연구실 방문