어셈블리프로그래밍설계 및 실습 보고서

실험제목: Assembly_Project

제출일자: 2023년 12월 04일 (월)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 화 6,7 목 5

학 번: 2020202037

성 명: 염정호

1. Introduction목적

임의의 부동소수점으로 이루어진 배열을 병합정렬과 삽입정렬 두 가지로 정렬하고 메모리에 저장하는 코드를 구현해 서로 비교한다.

2. Background

삽입정렬 :

정렬 알고리즘 중 하나로 배열 내 데이터를 순회하며 정렬이 필요한 원소를 적당한 위치에 삽입하는 알고리즘이다. 정렬된 부분에 새로운 데이터를 적절한 위치에 삽입하고 이와 같은 과정을 반복하며 정렬한다. 배열의 크기가 큰 경우 많은 이동을 유발하기 때문에 소량의 자료를 처리하는데 적합하다.

1. 처음 값은 해당 위치에 그대로 정렬

3	1	7	4	9	6	10	5	2	8

2. 두 번째 데이터 1과 바로 앞에 위치한 3과 비교를 실행한다. 두 번째 데이터가 더 작음으로 3을 뒤로 옮기고 1을 앞에 위치시킨다.

3	1	7	4	9	6	10	5	2	8

3. 같은 방법으로 n번째 값을 배열의 가장 첫 요소부터 n-1번째 요소까지 비교 하면서 데이터가 저장될 위치를 찾아 삽입하고 이전 데이터를 옮겨준다.

안정적인 정렬 방법이며 레코드 수가 적을 경우 알고리즘 자체가 간단함으로 복잡한 다른 정렬 방법보다 안전해 일부 레코드가 정렬 되어있는 경우 매우 효율적이지만, 많은 레코드들의 이동을 필요로 해 레코드 수가 클 경우 비효율적이라는 단점이 있다.

시간 복잡도 :

최선의 경우(이미 정렬된 채로 삽입되는 경우) 이동 없이 1번만의 비교로 모든 배열이 정렬되기 때문에 O(n)이 된다.

최악의 경우는 데이터가 역순으로 저장되어 각 루프마다 i번의 반복이 수행되기 때문에 O(n^2)이 된다.

병합정렬 :

분할 정복 알고리즘의 하나로 하나의 문제를 작은 두 개의 문제로 구분하고 각각의 문제를 해결 한 뒤 결과값을 합쳐 원래의 문제를 해결하는 방식이다.

정렬 방법 ;

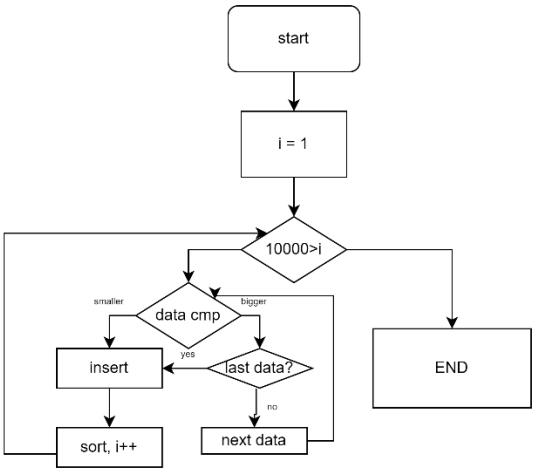
1. 정렬되지 않은 리스트를 하나의 원소만을 가지고 있는 n개의 리스트로 구분한다.
2. 정렬된 리스트가 하나만 남을 때 까지 반복해서 병합하며 정렬된 부분 리스트를 생

성한다. 즉 마지막으로 남은 부분 리스트가 정렬된 리스트가 된다.

병합정렬은 임시 배열이 필요하다는 단점이 존재하지만 데이터의 개수에 크게 영향을 받지 않고 시간 복잡도가(nlog2n)으로 동일하다.

3. Algorithm

삽입정렬 순서도



삽입 정렬의 경우 정렬되지 않은 데이터를 불러와 결과 메모리 정렬 후 저장하게 된다. 삽입될 데이터은 이미 정렬된 데이터의 첫번째 요소부터 마지막 요소 까지 크기를 비교하게 되는데 비교순서는 다음과 같다.

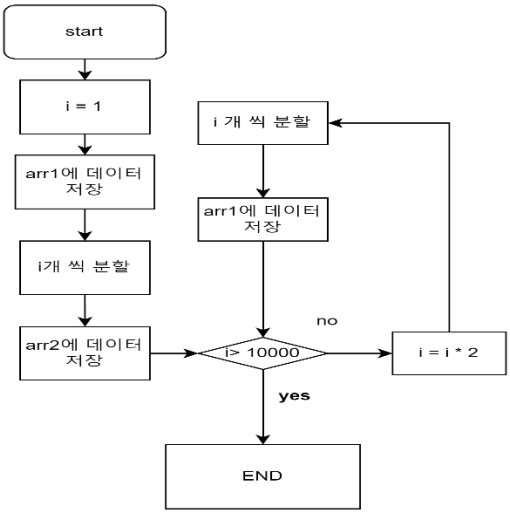
1. Sign bit 부호 비트가 같은 경우 부호비트 이외의 값을 비교해주어야 한다.
   부호비트가 다른 경우 삽입하고자 하는 데이터 비트가 -이고 비교 대상이 + 인 경우 현재 위치에 데이터를 삽입하며 삽입하고자 하는 데이터가 +인 경우 다음 데이터와 비교를 실행한다.
2. 부호를 제외한 나머지 비트의 경우 부호비트가 음수인 경우 나머지 비트가 더 작은 쪽이 승자가 되고 양수인 경우 남은 비트가 더 큰 쪽이 승자가 되며 삽입 대상 데이터가 패자일 경우 해당 위치에 데이터를 삽입하고 더 이상 비교할 데이터가 없는 경우 정렬된 데이터의 마지막 위치에 값을 저장한다.

데이터 비교를 수행하기 위해 데이터를 가져올 때 sign Exponential, matisa를 구분해서 가지고 온다. Exponential이 모두 1인 경우 not a number로 처리하여 저장될 데이터 위치의 맨 마지막 단에 삽입되고 비교는 진행하지 않고 바로 다음 비트로 넘어가도록 설정하였다.

삽입 정렬에선 프로그램의 비교 횟수를 줄이기 위해 swp명령어를 이용해 메모리의 데이터와 레지스터의 값을 교환 할 수 있도록 하였다. 또한 sign비트를 제거한 데이터를 얻기 위해서는 쉬프트를 두 번 진행해야 했지만 7fffffff값을 and연산 시켜 2번을 1번 연산하는

것으로 대체할 수 있었다. 다만 레지스터의 개수에 제한이 있다 보니 원래 r0의 정렬된 배열 시작 주소 값을 저장할 배열이 없어 label을 이용해 데이터를 불러와야 했다.

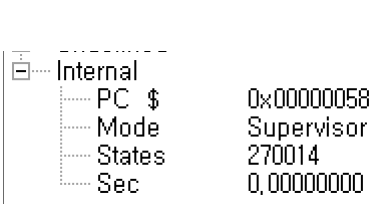
병합 정렬 순서도



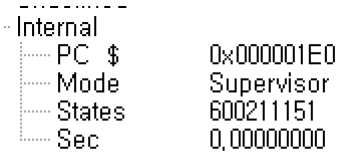
병합 정렬의 경우 arr1과 arr2를 옮겨가며 양 옆의 데이터들과 병합을 진행한다.
데이터 병합의 경우 1개 2개 4개 처럼 2^n승 단위로 진행된다. 해당 단위로 10000의 배열을 정렬하게 되면 병합하기 전에 남는 배열이 생기게 되는데 남는 데이터양에 따라서 해당 데이터들 끼리 병합을 진행하거나 데이터를 그대로 넣어줄 수 있도록 설계하였다.
남는 데이터가 i개 이상인 경우 둘이 병합을 진행하고
남는 데이터가 i개 미만인 경우 i개를 그대로 삽입 하도록 하였다.
이미 정렬 된 부분 또한 나중에 i개 이상의 데이터에서 처리하여 정렬 후 삽입된다.
삽입 정렬의 경우 nan과 inf를 미리 파악해 데이터 비교 횟수를 줄일 수 있었지만. 병합 정렬의 경우 해당 부분이 포함되면 로직이 더 복잡해지고 삽입 정렬에 비해 성능 업그레이드가 적어 따로 처리해주지 않고 일반 로직 안에서 정렬 되도록 하였다.
각 데이터의 정렬에서 비교는 삽입 정렬에서 사용한 로직과 마찬가지로 sign비트를 비교 후 그 외 비트끼리 비교하여 정렬 할 수 있도록 했다.
병합 정렬의 경우 데이터의 저장 위치가 arr1 , arr2로 달라질 수 있기 때문에 임시 배열에 데이터를 저장한 뒤 데이터가 끝나는 베열에서 결과값에 로드 할 수 있도록 진행하였다.

4. Performance & Result

Insert



비교연산 시작 전



비교연산 시작 후

연산 횟수는 데이터에 따라 약간에 변화가 있음

총 599,941,137state / code size : 512

```
0x00031190: 4A 9F 6B FF A8 0F 60 FF 94 F4 59 FF 36 ED 52 FF 96 6C 44 FF D4 D9 28 FF C2 88 0C FF 2E E7 01 FF
0x000311B0: FE E0 F8 FE FC 73 DE FE E8 AC D5 FE 50 28 CA FE 0E 45 C9 FE 82 FE BD FE 3C 63 AD FE 2C 98 9A FE
0x000311D0: 52 CB 61 FE 76 C4 5F FE 08 C1 50 FE 82 F2 4E FE 20 A7 25 FE CE FB 12 FE 9E 74 0F FE DA C4 0E FE
0x000311F0: D0 1F 05 FE FC 25 F5 FD 3C 90 DF FD 2C 02 D7 FD 66 22 D3 FD 80 40 A4 FD 46 D4 84 FD 62 96 72 FD
0x00031210: 3E F9 6F FD C6 C0 6C FD 88 03 64 FD 48 21 5A FD 3A 8A 43 FD 4A 24 37 FD 70 F0 34 FD 02 37 30 FD
0x00031230: FE EE 2A FD 3A 59 27 FD A8 A0 25 FD F2 2F 21 FD 7E 32 0C FD F8 6B E5 FC D8 4B E2 FC FE D2 D1 FC
0x00031250: E6 6D CC FC 08 D2 CB FC 20 AB C5 FC A8 6E B6 FC 2E 5D B5 FC 64 E2 B0 FC 48 06 9A FC 5A 72 6C FC
0x00031270: 00 A1 62 FC 9C FD 61 FC AE 91 57 FC A0 2E 41 FC 82 1F 40 FC DC A5 38 FC FE 04 32 FC 38 E8 2C FC
```

시작 데이터

```
0x00033904: F8 9B 1B 80
0x00033908: 14 C7 19 80
0x0003390C: 00 00 00 00
0x00033910: BE E7 12 00
```

중간데이터 양수에서 음수로 전환 되는 부분.

```
0x0003ACCC: 8F 1B 4D 7F 06 5B 50 7F CA 63 51 7F D2 E8 55 7F 28 48 59 7F 42 C7 59 7F C8 3F 5A 7F 7A DA 62 7F
0x0003ACEC: 9D D8 69 7F 3C 7B 6A 7F C0 61 6C 7F 6D D1 6E 7F 2A EF 6F 7F 5C A9 70 7F 2E F5 71 7F 08 44 72 7F
0x0003AD0C: 89 F0 76 7F 8C 5C F7 7F C8 9E B0 7F 5C 82 DB 7F 80 C6 D6 FF 96 9D D6 FF DA E1 B9 7F B8 55 F0 7F
0x0003AD2C: C5 10 E3 7F B5 32 98 7F 0C DF AB 7F 51 26 D1 7F FC BF DB FF 7A 3A E1 7F 5E 8A CC FF 70 14 8F 7F
0x0003AD4C: 5D 21 A0 7F E6 B7 86 7F 64 D8 CE 7F 84 C6 91 7F 9B 25 E8 7F D2 C7 CC 7F AA BF DC 7F 60 4C E4 7F
0x0003AD6C: 12 EE BA 7F 60 CF B9 7F 50 EB 9A 7F 0C 9F D4 7F DD 6A B9 7F 60 5B BB 7F 0E 2A A5 FF FE FF C1 7F
0x0003AD8C: 26 1B EE 7F 56 65 F6 FF EC C5 AC 7F F0 8C AD FF 16 66 A2 FF 1E 2A E6 7F 76 02 AD 7F 54 30 F1 7F
0x0003ADAC: 9C BB 9A 7F A6 2F E8 7F 66 EB BD FF 9E 1D BE 7F 33 E0 EE 7F CE 78 84 7F 66 12 ED FF 28 78 9F 7F
```

데이터 마지막 not a number처리

```
0x000311AC: 5E 8A CC FF
0x000311B0: FC BF DB FF
0x000311B4: 96 9D D6 FF
0x000311B8: 80 C6 D6 FF
0x000311BC: 00 00 80 FF
0x000311C0: A8 0F 60 FF
0x000311C4: 94 F4 59 FF
0x000311C8: 36 ED 52 FF
0x000311CC: 96 6C 44 FF
```

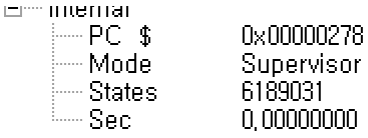
NAN – (-)무한대 – 음수, 양수 (+)무한대 순으로 정렬

```
0x0003AD20: 5C A9 70 7F
0x0003AD24: 2E F5 71 7F
0x0003AD28: 08 44 72 7F
0x0003AD2C: 89 F0 76 7F
0x0003AD30: 00 00 80 7F
0x0003AD34: C3 55 84 7F
0x0003AD38: 8C 5C F7 7F
```

NAN끼리는 따로 정렬을 진행하지 않음

Merge

비교연산 시작 전은 이전과 동일



비교연산 시작 후 6,189,031

총 5,919,017 소요

시작

```
0x00031190: 56 65 F6 FF 66 12 ED FF FC BF DB FF 80 C6 D6 FF 96 9D D6 FF 5E 8A CC FF 66 EB BD FF F0 8C AD FF
0x000311B0: 0E 2A A5 FF 16 66 A2 FF 4A 9F 6B FF A8 0F 60 FF 94 F4 59 FF 36 ED 52 FF 96 6C 44 FF D4 D9 28 FF
0x000311D0: C2 88 0C FF 2E E7 01 FF FE E0 F8 FE FC 73 DE FE E8 AC D5 FE 50 28 CA FE 0E 45 C9 FE 82 FE BD FE
0x000311F0: 3C 63 AD FE 2C 98 9A FE 52 CB 61 FE 76 C4 5F FE 08 C1 50 FE 82 F2 4E FE 20 A7 25 FE CE FB 12 FE
0x00031210: 9E 74 0F FE DA C4 0E FE D0 1F 05 FE FC 25 F5 FD 3C 90 DF FD 2C 02 D7 FD 66 22 D3 FD 80 40 A4 FD
0x00031230: 46 D4 84 FD 62 96 72 FD 3E F9 6F FD C6 C0 6C FD 88 03 64 FD 48 21 5A FD 3A 8A 43 FD 4A 24 37 FD
0x00031250: 70 F0 34 FD 02 37 30 FD FE EE 2A FD 3A 59 27 FD A8 A0 25 FD F2 2F 21 FD 7E 32 0C FD F8 6B E5 FC
```

NAN – (-)무한대 – 음수, 양수 (+)무한대 순으로 정렬

시작

```
0x00033900: 14 39 4A 80
0x00033904: F8 9B 1B 80
0x00033908: 14 C7 19 80
0x0003390C: 00 00 00 00
0x00033910: BE E7 12 00
0x00033914: 9C B7 15 00
```

중간 부분

```
0x000338F4: AA 84 6A 80
0x000338F8: A2 E8 5C 80
0x000338FC: 96 E5 50 80
0x00033900: 14 39 4A 80
0x00033904: F8 9B 1B 80
0x00033908: 14 C7 19 80
0x0003390C: 00 00 00 00
0x00033910: BE E7 12 00
0x00033914: 9C B7 15 00
0x00033918: E6 2E 3F 00
0x0003391C: B2 4E 40 00
0x00033920: 5E 17 4A 00
0x00033924: BE 2A 59 00
0x00033928: CE 4E 6C 00
0x0003392C: 52 2C 73 00
0x00033930: 2E D3 82 00
0x00033934: B0 6B 87 00
```

끝

```
0x0003AD20: 5C A9 70 7F
0x0003AD24: 2E F5 71 7F
0x0003AD28: 08 44 72 7F
0x0003AD2C: 89 F0 76 7F
0x0003AD30: 00 00 80 7F
0x0003AD34: 0B 9E 82 7F
0x0003AD38: C3 55 84 7F
0x0003AD3C: CE 78 84 7F
0x0003AD40: E6 B7 86 7F
0x0003AD44: 70 14 8F 7F
```

5. Consideration

처음 데이터를 넣고 쓰는 과정에서 r4, r5에 데이터를 받고 ,r6을 이용해 서로 교환하고 입력하는 로직을 사용하였는데 해당 로직에서는 15억회의 state가 발생하게 되었다. 이를 swap함수를 이용해 고치고 나니 state수를 8억회로 줄일 수 있었다.

처음 플로팅 포인트 비교 연산을 진행할 때 state수를 줄이기 위해 데이터를 부호, 지수부, 가수부 세 개로 분류해서 데이터 비교를 진행하였는데 모든 비교 연산은 32bit전부 이용해서 비교되기 대문에 세 가지가 아닌 부호와 나머지로 구분한 뒤 부호 값에 따라 대수 비교 로직만 바꾸어 주었다.

어셈블리 프로그램의 경우 코드를 무조건적으로 위에서 아래로 읽어오기 때문에 해당 코드 뒤에 다른 코드들을 어떻게 배치하냐에 따라 state수를 줄일 수 있었고 해당 가장 많은 비교가 필요한 연산의 경우 해당 코드의 맨 마지막에 배치에 마지막 예외로 처리함으로 서 비교연산 회수를 줄이는데 도움이 되었다.

6. Reference

합병정렬/https://gmlwjd9405.github.io/2018/05/08/algorithm-merge-sort.html

삽입정렬/https://gmlwjd9405.github.io/2018/05/06/algorithm-insertion-sort.html

이준환/어셈블리프로그램설계및실습/광운대학교/2023