자료구조및프로그래밍 HW12

C211171 최후락

2025 12 14

1 개요

삽입 정렬(Insertion Sort), 퀵정렬(Quick Sort), 자연 합병 정렬(Natural Merge Sort), 힙 정렬(Heap Sort)을 직접 구현하고, 다양한 입력 데이터에 대해 실행 시간을 측정한 뒤 그래프를 통해서 분석한다. 각 알고리즘 구현과정에서 강의록을 참고해서 작성했다.

- 과제를 진행한 환경
- 삽입 정렬(Insertion Sort)
- 퀵 정렬(Quick Sort)
- 자연 합병 정렬(Natural Merge Sort)
- 힙 정렬(Heap Sort)
- 테스트 결과
- 어려웠던 점

2 과제를 진행한 환경

Windows 환경에서 WSL(Ubuntu)를 사용하여 수행하였다. 과제에 사용된 시스템 사양은 다음과 같다.

- CPU: AMD Ryzen 7 7700X (8-Core, 4.50 GHz)
- RAM: 32 GB
- 운영체제: Windows 64-bit
- 컴파일러: g++ (C++17)

또한 아래와 같은 makfile을 만들어서 사용했다.

Listing 1: makefile

```
CC = g++
CFLAGS = -O2 -std=c++17
OBJS = hw12.o
TARGET = hw12

run: all
  ./$(TARGET)

all: $(TARGET)

$(TARGET): $(OBJS)
  $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

hw12.o: hw12.cpp sort.h
  $(CC) $(CFLAGS) -c hw12.cpp

clean:
  rm -f *.o
  rm -f $(TARGET)
```

3 삽입 정렬(Insertion Sort)

삽입 정렬은 이미 정렬된 부분 배열에 새로운 원소를 하나씩 삽입하는 방식의 정렬 알고리즘이다. 값을 전부 오른쪽으로 옮긴 뒤 왼쪽으로 하나씩 정렬하며 이미 정렬된 부분에서 적절한 위치에 끼워 넣는다.
구현이 단순하지만 입력 크기가 커질수록 실행 시간이 급격히 증가하며, 평균 및 최악의 경우 시간 복잡도는 $O(N^2)$이다.

Listing 2: Insertion Sort implementation

```
    void insertionSort(int arr[], int n) {
      for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
          arr[j + 1] = arr[j];
          j--;
        }
        arr[j + 1] = key;
      }
    }
```

4 퀵 정렬(Quick Sort)

퀵 정렬은 피벗을 기준으로 배열을 분할하여 재귀적으로 정렬하는 알고리즘이다. 평균적으로 $O(N \log N)$의 시간 복잡도를 가지며, 테스트 결과에서 삽입 정렬에 비해 훨씬 우수한 성능을 보였다.

Listing 3: Quick Sort using Hoare partition

```
    template <typename T>
    int partition(T* a, int left, int right) {
      T pivot = a[left];
      int i = left - 1;
      int j = right + 1;

      while (true) {
        do { i++; } while (a[i] < pivot);
        do { j--; } while (a[j] > pivot);
        if (i >= j) return j;
        swap(a[i], a[j]);
      }
    }

    template <typename T>
    void quickSort(T* a, int left, int right) {
      if (left < right) {
        int p = partition(a, left, right);
        quickSort(a, left, p);
        quickSort(a, p + 1, right);
      }
    }
```

5 자연 합병 정렬(Natural Merge Sort)

자연 합병 정렬은 배열 내에 이미 정렬되어 있는 구간(run)을 찾아 이를 반복적으로 병합하는 방식의 정렬 알고리즘이다. 입력 데이터가 부분적으로 정렬되어 있는 경우 일반 합병 정렬보다 효율적으로 동작할 수 있으며, 전체 시간 복잡도는 $O(N \log N)$이다.

run을 생성하기 위해 배열을 왼쪽부터 순차적으로 스캔하면서 인접한 두 원소를 비교하여 오름차순이 유지되는지를 확인한다. 오름차순 조건이 처음으로 깨지는 지점까지를 하나의 run으로 간주하며, 이 과정을 배열 끝까지 반복하면 여러 개의 run이 자연스럽게 생성된다. 처음에는 run을 별도의 배열에 저장하는 방식을 고려하였으나, 불필요한 메모리 사용이 크다고 판단하였다. 따라서, run을 실제로 배열에 저장하지 않고 left, mid, right 인덱스를 이용하여 run의 경계를 관리함으로써 추가적인 메모리 사용을 최소화하였다.

Listing 4: Natural Merge Sort implementation

```
    template <typename T>
    bool mergeRuns(T* a, T* temp, int n) {
      int i = 0;
      bool merged = false;

      while (i < n) {
        int left = i;
        while (i + 1 < n && a[i] <= a[i + 1]) i++;
        int mid = i;
        if (mid == n - 1) break;

        i++;
        while (i + 1 < n && a[i] <= a[i + 1]) i++;
        int right = i;

        int p = left, q = mid + 1, k = left;
        while (p <= mid && q <= right)
        temp[k++] = (a[p] <= a[q]) ? a[p++] : a[q++];
        while (p <= mid) temp[k++] = a[p++];
        while (q <= right) temp[k++] = a[q++];

        for (int t = left; t <= right; t++)
        a[t] = temp[t];

        merged = true;
        i++;
      }
      return merged;
    }

    template <typename T>
    void naturalMergeSort(T* a, int n) {
      T* temp = new T[n];
```

```
34        while (mergeRuns(a, temp, n));
35        delete[] temp;
36    }
```

6 힙 정렬(Heap Sort)

힙 정렬은 힙 자료구조를 이용하여 최댓값을 반복적으로 추출하는 방식의 정렬 알고리즘이다. 힙을 만들고 root값을 뽑아서 맨 오른쪽으로 옮기는 과정을 반복한다. 이 과정을 반복하면 힙의 특성상 제일 큰 값이 순서대로 선택되어 정렬된다.

입력 데이터의 정렬 상태와 관계없이 항상 $O(N \log N)$의 시간 복잡도를 보장하는 특징이 있다.

Listing 5: Heap Sort using max heap

```
template <typename T>
void adjust(T* h, int root, int n) {
  T temp = h[root];
  int child = root * 2;

  while (child <= n) {
    if (child < n && h[child] < h[child + 1])
    child++;
    if (temp >= h[child]) break;
    h[root] = h[child];
    root = child;
    child *= 2;
  }
  h[root] = temp;
}

template <typename T>
void heapSort(T* a, int n) {
  T* h = new T[n + 1];
  for (int i = 0; i < n; i++)
  h[i + 1] = a[i];

  for (int i = n / 2; i >= 1; i--)
  adjust(h, i, n);

  for (int i = n; i >= 2; i--) {
    swap(h[1], h[i]);
    adjust(h, 1, i - 1);
  }

  for (int i = 0; i < n; i++)
  a[i] = h[i + 1];

  delete[] h;
}
```

7 테스트 결과

과제에서 제공된 서로 다른 입력 데이터에 대해 정렬 알고리즘들의 실행 시간을 비교한 결과를 정리하였다.
입력 데이터는 감소(decreasing), 부분 정렬(partial), 랜덤(random), 완전 정렬(sorted) 상태가 있으며, 각 경우에 대해 삽입 정렬, 퀵 정렬, 자연 합병 정렬, 힙 정렬의 성능 차이를 확인하였다. 코드 실행 결과를 엑셀에 정리한 뒤 그래프로 시각화하였다. 그래프의 X축은 입력 크기 N이며, 실행 시간 차이를 명확히 보기 위해 로그 스케일을 적용하였다.

7.1 decreasing

감소 순서로 정렬된 입력은 삽입 정렬에 가장 불리한 경우에 해당한다. 그래프에서 확인할 수 있듯이, 삽입 정렬은 입력 크기가 증가할수록 실행 시간이 급격히 증가하였다. 이는 대부분의 원소가 이미 정렬된 부분 배열의 앞쪽으로 이동해야 하므로 비교와 이동 연산이 반복적으로 발생하기 때문이다.

반면 퀵 정렬, 자연 합병 정렬, 힙 정렬은 입력 데이터의 정렬 상태에 크게 영향을 받지 않고 비교적 완만한 증가 추세를 보였다.

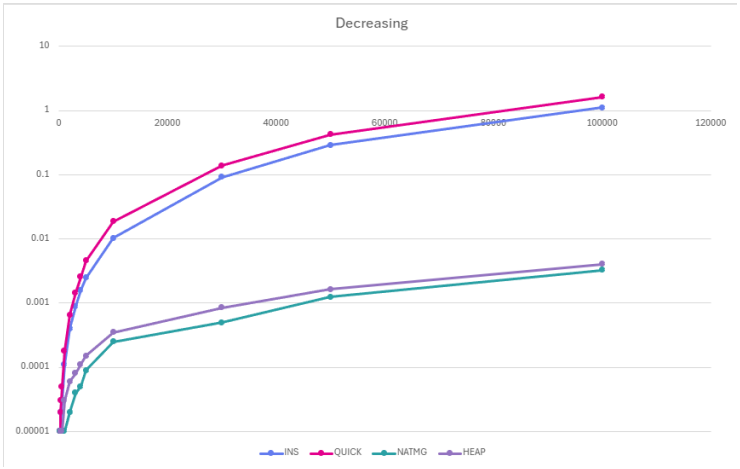

Figure 1: decreasing

7.2 partial

부분적으로 정렬된 입력의 경우, 자연 합병 정렬이 비교적 안정적인 성능을 보였다. 이는 배열 내에 이미 정렬된 구간(run)이 존재할 때, 해당 구간을 그대로 활용하여 병합 횟수를 줄일 수 있기 때문이다.

삽입 정렬 역시 완전히 랜덤한 입력에 비해 상대적으로 빠르게 동작하는 모습을 보였으나, 입력 크기가 커질수록 실행 시간은 점차 증가하였다. 퀵 정렬과 힙 정렬은 입력 데이터의 부분 정렬 여부와 관계없이 전반적으로 일정한 성능을 유지하였다.

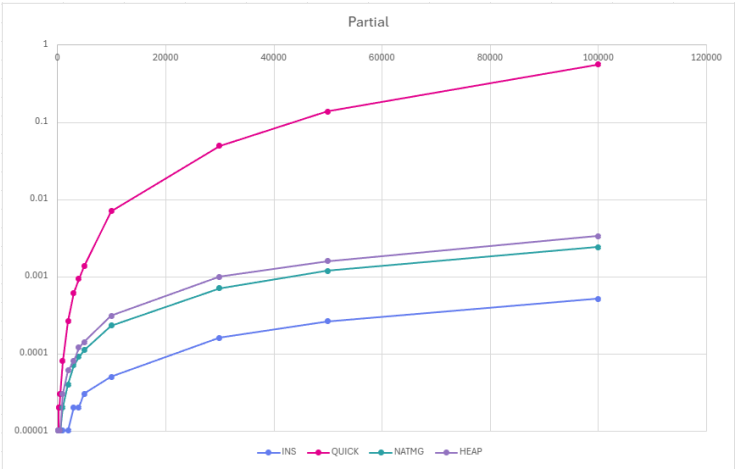

Figure 2: partial

7.3 random

랜덤 데이터에 대한 실험 결과, 삽입 정렬은 입력 크기가 커질수록 실행 시간이 빠르게 증가하였다. 반면 퀵 정렬, 자연 합병 정렬, 힙 정렬은 입력 크기에 따라 비교적 완만한 증가 경향을 보였다.

특히 퀵 정렬은 대부분의 입력 크기에서 가장 짧은 실행 시간을 기록하였다. 이는 랜덤 데이터에서 분할이 비교적 균형 있게 이루어질 가능성이 높기 때문으로 볼 수 있다.

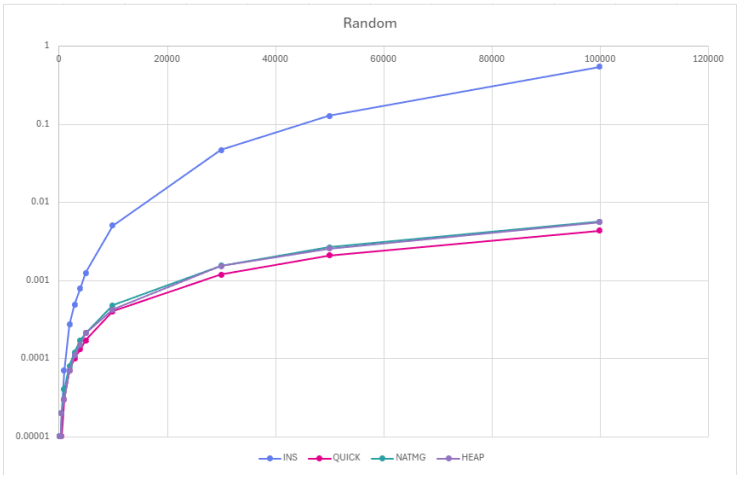

Figure 3: random

7.4 sorted

완전히 정렬된 입력의 경우, 삽입 정렬의 실행 시간이 매우 짧게 나타났다. 이는 내부 반복문에서 추가적인 이동 연산이 거의 발생하지 않아, 삽입 정렬이 O(n)에 가깝게 동작하기 때문이다.

자연 합병 정렬 또한 전체 배열이 하나의 run으로 인식되어 병합 과정이 거의 발생하지 않았다. 반면 퀵 정렬과 힙 정렬은 입력 데이터가 이미 정렬되어 있더라도 기본적인 분할 및 힙 구성 과정을 수행하므로 상대적으로 일정한 실행 시간을 보였다.

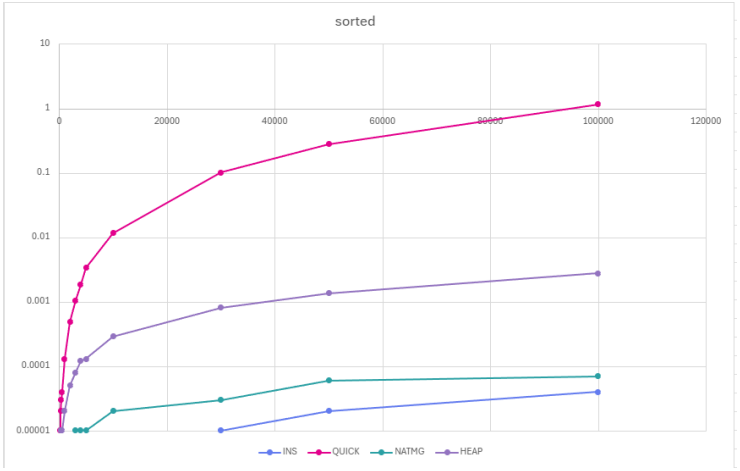

Figure 4: sorted

8 어려웠던 점

이번 과제에서는 정렬 알고리즘 구현뿐만 아니라 실행 환경을 구축하는 과정에서도 어려움이 있었다. 과제에서 제공한 코드가 리눅스 환경을 전제로 작성되어 있어, Windows 환경에서 그대로 실행하기 어려웠다. 이를 해결하기 위해 WSL(Ubuntu)을 설치하고, 파일 시스템을 리눅스 환경에 맞게 연결한 뒤 컴파일 및 실행이 정상적으로 이루어지도록 설정하였다. 특히 include 경로와 실행 방식의 차이로 인해 초기에는 컴파일 오류가 발생하였으나, 빌드 과정과 파일 구조를 정리하면서 문제를 해결할 수 있었다.

또한 자연 합병 정렬을 구현하는 과정에서도 고민이 필요하였다. 처음에는 run을 별도의 배열에 저장하여 처리하는 방식을 떠올렸으나, 이 경우 불필요한 메모리 사용과 복사가 많이 발생한다고 판단하였다. 이에 따라 run을 실제로 저장하지 않고, left, mid, right와 같은 인덱스를 이용하여 run의 경계를 관리하는 방식으로 구현을 변경하였다. 이 과정에서 인덱스 처리와 병합 순서를 정확히 맞추는 것이 쉽지 않았으나, 여러 번의 테스트를 통해 안정적으로 동작하도록 수정하였다.