CSE3081-1: Design and Analysis of Algorithms (Fall 2018)

Machine Problem 2: Master of Sorting

Report

전공 : 컴퓨터공학과

학번 : 20171667

이름 : 이용욱

보고서 목차

1. 과제 개요
2. 작성한 코드에 대한 설명
3. 알고리즘의 종류와 입력의 크기에 따른 소요시간
4. 프로젝트 개요

정수로 이루어진 수열이 주어졌다. 이 수열을 오름차순으로 배열하는 것이 과제이다.

이 과제에서는

알고리즘1 : Insertion Sort

알고리즘2 : Quick Sort

알고리즘3 : Heap Sort

알고리즘4 : Introspective Sort 와 Counting Sort중 선택

을 통해 과제를 완수하였다.

또한 10부터 10^6 까지 크기를 변화시키긴, 임의의 수열과 내림차순 수열을 입력으로 해서

알고리즘의 종류에 따른 소요시간을 비교해보았다.

1. 작성한 코드

C++11로 작성하였으며 Visual Studio 2017에서 개발하였다.

G++ 컴파일러에서도 정상적으로 작동함을 확인하였다.

다만 C++11에서 개발하였으므로

컴파일시에 “g++ -std=c++11 mp2\_20171667”과 같이 c++버전을 지정해주어야 한다.

* 1. main (int argc, char \*argv[]) 함수

매개변수로 입력파일의 파일명과 알고리즘의 인덱스를 받는다.

입력/출력 파일을 열고 arr벡터에 입력을 담는다.

clock (); 을 통해 시작시간을 재다.

인덱스에 해당하는 알고리즘을 구현하는 함수에 배열을 넘겨준다

끝 시간을 재고 소팅된 배열과 소요시간을 출력한다.

**InsertionSort**

삽입정렬을 수행한다.

arr[i] = for\_insert는 지금 삽입할 원소를 저장한다.

j-1에 있는 원소를 j로 당겨가며 for\_insert보다 작거나 같은 첫 원소를 찾는다

arr[j-1] <= for\_insert < arr[j+1]

이 성립하므로 for\_insert는 j에 대입한다.

**QuickSort**

최적화 없는 Quick Sort

left > right 이거나 left==right이면 divid 할 필요가 없으므로 종료

arr[right]을 기준으로 Partition 수행

[left, mid-1]에는 arr[right] 보다 작은 원소만이 [mid+1, right]에는 arr[right] 보다 크거나 같은 원소만이 있다.

QuickSort (arr, left, mid - 1);

QuickSort (arr, mid + 1, right);

을 호출한다.

**Partition**

주어진 배열의 [left, right]구간을, pviot보다 작은 원소와 작지 않은 원소로 분할한다.

변수 first\_not\_small는 pviot보다 작지 않을 수 있는 첫 원소의 인덱스를 저장한다.

pivot보다 작은 원소를 발견하면, first\_not\_small에 넣고

first\_not\_small은 1증가시킨다.

주어진 배열의 [left, right]구간에서, pviot보다 작지 않은 첫 원소의 인덱스를 반환한다.

**Heap Sort**

**class MinHeap**

heap의 원소 개수 size와

min heap을 구현한 트리 vector<int>tree를 갖는다.

**MinHeap::build**

트리를 받아 생성되었을 경우에 이를 min heap으로 변경하는 함수

이 프로젝트에서는 사용하지 않았다

**MinHeap::push (const int new\_element)**

트리에 원소를 추가하는 함수

벡터의 크기가 필요미만이면 resize한다

마지막 원소에 e를 저장한다

현제 새 원소의 위치를 curr로 두고

더 큰 부모가 없을 때까지 부모와 자식을 swap한다

**MinHeap::pop ()**

min heap에서 원소를 pop하는 함수

빈 트리인경우, 0을 출력한다

root의 원소를 저장하고, 이를 마지막원소로 대체한다

curr : 현재 마지막 원소였던 원소가 있는 위치

next : 마지막 원소였던 원소가 다음으로 향할 위치

next는 curr의 두 자식 중 최소를 가리킨다.

next가 curr 보다 작으면 둘을 swap한다

그렇지 않으면 반복문을 종료한다.

**HeapSort**

MinHeap 객체를 선언한다

arr의 모든 원소를 push한다.

arr의 각 원소에 pop의 반환값을 저장한다.

**나의 최적화 소트**

Counting Sort( O(max - min) ) 과 IntroSort ( O(n log(n)) ) 의 시간복잡도를 구하여 효율적인 것을 선택한다.

다만 Counting Sort 는 저장공간을 많이 차지하므로

512MB 이상을 사용할 것으로 예상되는 경우 작동시키지 않았다.

max - min 의 크기가 2^27 이면 count 배열이 512MB를 차지한다.

**CountingSort**

count 배열에 각 원소가 얼마나 등장하는 지 저장하고 이를 순회한다.

O(element\_max - element\_min)만에 정렬할 수 있으나 공간을 많이 사용한다.

각 원소의 등장 횟수를 count 배열에 저장하고 count 배열을 작은 인덱스부터 순회하며 둥장한 만큼 arr 배열에에 다시 저장한다.

**IntroSort**

Introspective Sort 를 하는 함수이다.

여기에는 4가지 최적화 기법이 사용되었다.

1. Tail Call Optimization

2. IntroSpective Sort

3. 3 - median pivot

4. threshold 까지만 sort 하고 추후에 Insertion Sort 로 sort 하기

1. Tail Call Optimization

Tail Call Optimization 이 적용되었다.

둘 중 큰 구간은 이 함수에서 계속 작동하고,

작은 구간은 새 함수를 호출하여 작동한다.

이로서 함수 호출의 횟수를 줄일 수 있다.

2. IntroSpective Sort

만약 이 제귀함수의 깊이가 log(len)\*2를 넘은 경우,

이 구간은 heap sort 한다

pivot이 worst하게 설정되어 시간복잡도가 O(n^2)이 되는 것을 막기 위함이다.

3. 3 - median pivot

arr[left], arr[(left+right)/2], arr[right]을 소팅한다.

셋의 중간값인 arr[(left + right) / 2]을 pivot으로 두고 Partition한다.

입력이 감소하거나, arr[right] 값이 지나치게 크거나 작은 경우에 효과적이다.

4.threshold 까지만 sort 하고 추후에 Insertion Sort 로 sort 하기

구간의 길이가 32를 넘어야 위의 코드를 실행하였다.

아닌 경우 나중에 insertion sort한다.

이 알고리즘의 작동 결과로는 32보다 작은 크기의 partition으로 전체 입력이 분할되는데

서로 다른 partition의 원소끼리는 대소관계가 일정하다. 따라서 이 상태에서 insertion sort를 동작시키면 O(32N)안에 소팅을 할 수 있다.

이러한 소팅이 제귀함수를 더 많이 부르는 것 보다 효율적임이 알려져 있다.

1. 알고리즘의 종류와 입력의 크기에 따른 소요시간 실험
   1. 실험환경

구동환경

OS : Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-130-generic x86\_64)

컴파일 및 실행 : 학교 Cspro 서버에서 g++ -std=c++11 옵션으로

CPU : Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz

cache size : 256000KB

RAM size : 65852732 kB

* 1. 실험 설정

위에서 구현한 세 알고리즘이 실제로 다양한 크기의 입력을 처리하는 시간을 비교하였다.

입력은 수열의 길이를 10부터 10의 배수로 늘려가며 10^6 까지 입력하였다.

입력은 1부터 size까지를 중복 없이 갖는다. 이를 <algorithm> 해더파일의 random\_shiffle() 을 이용하여 섞었다.

다른 입력은 size 부터 1까지의 값을 내림차순으로 배열하였다.

Introspective Sort 와 Counting Sort 는 따로 측정하였다

* 1. 실험 결과

임의의 수열

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Quick Sort | Heap Sort | Introspective Sort | Counting Sort |
| 10^1 | 0.000003 | 0.000005 | 0.000011 | 0.000022 | 0.000026 |
| 10^2 | 0.000054 | 0.000033 | 0.000058 | 0.000051 | 0.000050 |
| 10^3 | 0.004924 | 0.000510 | 0.000825 | 0.000442 | 0.000437 |
| 10^4 | 0.305218 | 0.005984 | 0.010871 | 0.005666 | 0.005530 |
| 10^5 | 19.725719 | 0.056381 | 0.093319 | 0.053600 | 0.048901 |
| 10^6 | 2000 (추정) | 0.494750 | 0.801001 | 0.439332 | 0.459863 |

여기서는 Quick Sort와 Introspective Sort가 가장 빨랐다.

또한 각 알고리즘의 시간복잡도에 비례하게 소요시간이 증가하였다.

큰 크기의 입력의 경우 Introspective Sort 와 Counting Sort가 가장 빨랐다.

내림차순 수열

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Insertion Sort | Quick Sort | Heap Sort | Introspective Sort | Counting Sort |
| 10^1 | 0.000004 | 0.000005 | 0.000010 | 0.000021 | 0.000022 |
| 10^2 | 0.000107 | 0.000138 | 0.000073 | 0.000062 | 0.000057 |
| 10^3 | 0.010137 | 0.012555 | 0.001040 | 0.000534 | 0.000528 |
| 10^4 | 0.582174 | 0.691339 | 0.013818 | 0.005673 | 0.005697 |
| 10^5 | 39.552215 | 48.778537 | 0.117927 | 0.055767 | 0.054894 |
| 10^6 | 4000 (추정) | 5000(추정) | 0.990818 | 0.502171 | 0.497804 |

우리의 Quick Sort는 pivot을 right으로 설정하기 때문에

n^2을 시간복잡도로 가져서 Insertion Sort와 유사한 결과가 나왔다.

여기서도 큰 크기의 입력의 경우 Introspective Sort 와 Counting Sort가 가장 빨랐다.

Random List의 실행 시간에 log\_10 을 취한 결과이다.

실험에서 예상한 바와 같이, 입력의 크기가 커질수록 Insertion Sort가 다른 배열보다 매우 큰 소요시간을 보이는데, 이는 이 알고리즘의 시간복잡도만이 O(n^2)이기 때문이다.

따라서 다른 알고리즘보다 증가 속도가 큰 그래프를 보인다.

Decreasing List의 실행 시간에 log\_10 을 취한 결과이다.

실험에서 예상한 바와 같이, arr[r]을 pivot으로 하는 Quick Sort또한 O(n^2)의 시간복잡도를 가지게 되었다. 그리하여, Insertion Sort와 매우 유사한 소요시간을 보였다.

반면 이 상황에서 Introspective는 pivot이 3-중간값 을 통해 정했고, 여러가지 최적화 기법을 사용하였기 때문에 O(n log n)의 시간복잡도를 유지하였다.