알고리즘 설계와 분석

(Design and Analysis of Algorithms)

Machine Problem 3

20161603

신민준

Table of Contents

1. Implementation
2. Experiment
   1. Environment Specifications
   2. Experiment Setup
   3. Result Data
   4. Data Analysis
      1. Random text file – Lorem Ipsum
      2. Worst-case text file
3. Additional Comments
4. Implementation

|  |
| --- |
| /\*\*  \* @brief Class for implementing Huffman Code Encryption & Decryption  \*/  class Huffman {  protected:  std::string inFilename, outFilename;  std::ifstream inFile;  std::ofstream outFile;  HuffmanOption option;  node\_ptr huffmanTree;  std::vector<node\_ptr> huffmanCode;  /\* compare class for priority queue \*/  class compare {  public:  bool operator()(const node\_ptr& v, const node\_ptr& u) const {  return v->freq > u->freq;  }  };  priority\_queue<node\_ptr, std::vector<node\_ptr>, compare> pq;  void makeHuffmanTree(void);  void assignBinaryCode(node\_ptr, std::string);  void initialize(void);  void openFile(void);  void writeToFile();  void reconstructTree(void);  char binToByte(string);  string byteToBin(unsigned char\*, int);  void addToTree(node\_ptr);  public:  Huffman(std::string, HuffmanOption);  const string getHuffmanCodeOf(char);  void encode(void);  void decode(void);  }; |

이 Sorting Algorithm을 구현하기 위해, 기본적으로 Quicksort를 구현하되, 몇 가지 최적화 방법을 적용해 더 효율적인 알고리즘이 되도록 했습니다.

먼저, Tail Recursion Optimising을 적용해 각 level에서 딱 한번의 recursive call을 수행하도록 구현했습니다. 이 때, recursive call이 실행되는 배열은 pivot을 기준으로 나뉜 두 배열 중 더 길이가 짧은 배열입니다. 기존의 Quicksort에서는 두번의 recursive call을 반드시 실행했기에, 이 방법을 사용하면 총 function call의 갯수를 최소화하여 실행 시간을 줄일 수 있습니다.

두번째로, Quicksort의 pivot을 정할 때, 배열의 첫 원소, 중간 원소, 그리고 마지막 원소 중 중간값을 가지는 원소를 pivot으로 정해, 특정 worst-case input에 대해 대처할 수 있도록 최적화했습니다.

다음 최적화 방법으로, Quicksort의 recursive call이 특정한 depth를 초과하게 되면, 남은 list를 heapsort를 사용해 정렬하도록 했고, 만일 input의 크기가 16 이하인 경우, insertion sort로 해당 배열을 정렬하게 구현했습니다. 이 방법을 사용하면 recursive call의 기하급수적인 증가를 완화할 수 있고, 특정 size 이하의 경우 insertion sort가 quicksort보다 실증적으로 더 빠르기에 결과적으로 계산 속도를 개선할 것입니다.

제가 소개한 최적화 방법은 각자 Linear한 시간 복잡도, 즉 의 time complexity를 지니고, max depth를 로 설정해놓았기에, worst time complexity와 average time complexity는 둘 다 의 time complexity를 가집니다.

1. Experiment
   1. Environment Specifications

MacBook Pro (13-inch, 2018)

OS - macOS Catalina version 10.15.1

CPU - 2.3 GHz Intel Core i5

RAM - 8GB 2133 MHz LPDDR3

Graphics - Intel Iris Plus Graphics 655 1538MB

Shell - zsh 5.7.1 (x86\_64-apple-darwin18.2.0)

Compiler - Apple clang version 11.0.0 (clang-1100.0.33.12)

* 1. Experiment Setup

Makefile의 경우, -O0 옵션을 사용해 컴파일 시의 컴파일러 최적화를 제외했습니다. Makefile의 내용은 다음과 같습니다.

|  |
| --- |
| # Flags: C11 standard & no compiler optimising  CFLAGS += -std=c11 -O0  # Libmath  LIBS += -lm  # making the executable from object file  mp2\_20161603: mp2\_20161603.o     $(CC) $(CFLAGS) mp2\_20161603.o -o mp2\_20161603 $(LIBS)  # an object file is made from one source code  mp2\_20161603.o: mp2\_20161603.c     $(CC) $(CFLAGS) -c mp2\_20161603.c  clean:     rm mp2\_20161603 mp2\_20161603.o |

Figure 1 Makefile

실험에 사용할 input을 생성하는 makein 프로그램을 구현했습니다. 해당 코드와 input을 생성한 터미널 명령어는 다음과 같습니다.

|  |
| --- |
| int main(int argc, const char\* argv[]) {     FILE\* fp;     if(argc < 6) {        fprintf(stderr, "Usage: %s filename num min max sorted [seed]\n", argv[0]);        exit(1);     }     if(!(fp = fopen(argv[1], "w"))) {        fprintf(stderr, "Error while opening file %s\n", argv[0]);        exit(1);     }     int num = atoi(argv[2]);     int min = atoi(argv[3]);     int max = atoi(argv[4]);     int\* data;     data = malloc(sizeof(int)\*num);     srand(atoi(argv[6]));     for(int i=0 ; i<num ; i++) {        data[i] = rand()%(max-min)+min;     }     if(atoi(argv[5])) {        quick\_sort(data, 0, num-1);     }     fprintf(fp, "%d ", num);     for(int i=0 ; i<num-1 ; i++) {        fprintf(fp, "%d ", data[i]);     }     fprintf(fp, "%d", data[num-1]);     fclose(fp);     return 0;  } |

Code 2

|  |
| --- |
| : 1572507275:0;./makein 10.txt 10 -10000000 1000000 0 214748  : 1572507791:0;./makein 100.txt 100 -10000000 1000000 0 214748  : 1572507795:0;./makein 1000.txt 1000 -10000000 1000000 0 214748  : 1572507799:0;./makein 10000.txt 10000 -10000000 1000000 0 214748  : 1572507802:0;./makein 100000.txt 100000 -10000000 1000000 0 214748  : 1572507805:0;./makein 1000000.txt 1000000 -10000000 1000000 0 214748  : 1572507809:0;./makein 10000000.txt 10000000 -10000000 1000000 0 214748  : 1572507819:0;./makein 100000000.txt 100000000 -10000000 1000000 0 214748  : 1572702065:0;./makein 10.txt 10 -10000000 10000000 1 1337  : 1572702069:0;./makein 100.txt 100 -10000000 10000000 1 1337  : 1572702073:0;./makein 1000.txt 1000 -10000000 10000000 1 1337  : 1572702078:0;./makein 10000.txt 10000 -10000000 10000000 1 1337  : 1572702082:0;./makein 100000.txt 100000 -10000000 10000000 1 1337  : 1572702087:0;./makein 1000000.txt 1000000 -10000000 10000000 1 1337  : 1572702091:0;./makein 10000000.txt 10000000 -10000000 10000000 1 1337  : 1572702100:0;./makein 100000000.txt 100000000 -10000000 10000000 1 1337 |

Figure 2 ~/.zsh\_history

[Figure 2]에서 보인 것과 같은 명령어에서 seed 값만 다르게 한 input 파일을 3번 생성해 각 input에 대해 실험을 진행했습니다.

* 1. Result Data
  2. Data Analysis
     1. Random text file – Lorem Ipsum
     2. Non-increasing Data

1. Additional Comments