

알고리즘 과제 #3

융합전자공학부 201500329

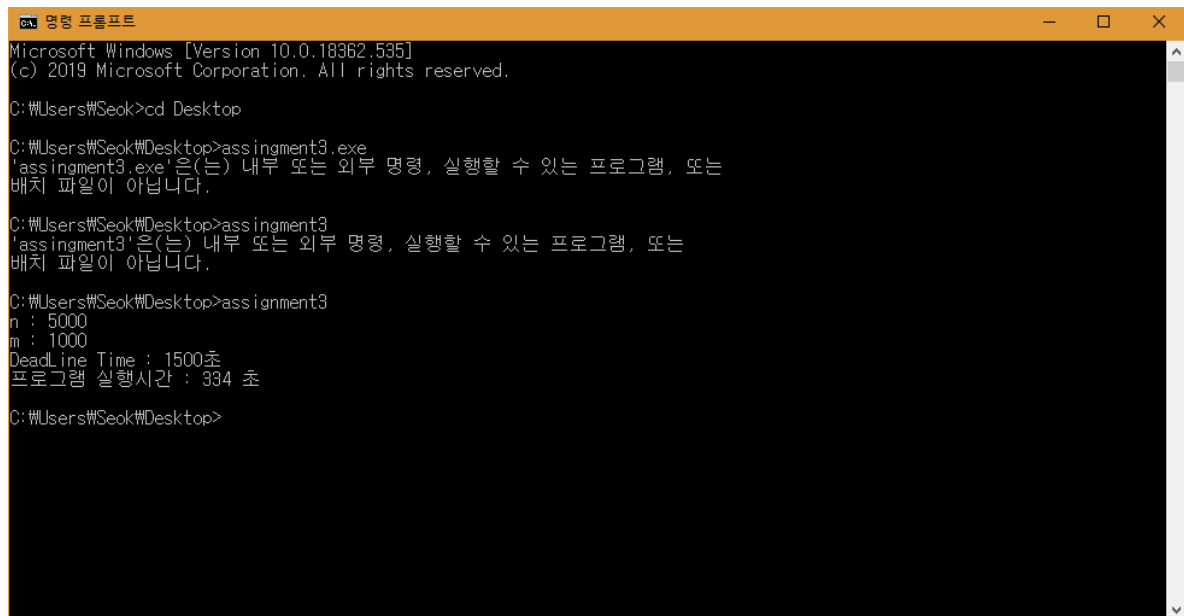
석정우

1. 구현환경

- 사용 언어 : C++ 14
- 사용 프로그램 : Dev C++ (도구 -> 컴파일러 설정 -> 컴파일러 추가명령 -> `-std=c++14` 입력)

2. 인터페이스

- 프로그램 초기화면



```
Microsoft Windows [Version 10.0.18362.535]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Seok>cd Desktop

C:\Users\Seok\Desktop>assingment3.exe
'assingment3.exe'은(는) 내부 또는 외부 명령, 실행할 수 있는 프로그램, 또는
배치 파일이 아닙니다.

C:\Users\Seok\Desktop>assingment3
'assingment3'은(는) 내부 또는 외부 명령, 실행할 수 있는 프로그램, 또는
배치 파일이 아닙니다.

C:\Users\Seok\Desktop>assignment3
n : 5000
m : 1000
DeadLine Time : 1500초
프로그램 실행시간 : 334 초

C:\Users\Seok\Desktop>
```

프로그램을 실행시키면, 같은 디렉토리에 있는 `input.txt`를 읽어온다. 내용을 읽어와서 총 $m \text{Bit} * n \text{Line}$ 인 지 출력하고나서, 해밍 거리를 계산하는 수행 과정을 거친 후, `output.txt`에 결과를 저장한다. 프로그램이 종료되면, 프로그램의 실행 시간이 출력된다.

- 출력 `output.txt` 화면


```

ifstream infile;

//정답 출력하기 위한 변수
int minanswer;

//mBit Stringdmf 최대 5000개 저장할 수 있는 stringSet 선언
string stringSet[5001];

//input 파일을 stringSet에 저장한다
infile.open("input.txt", ios::in);
int number = 0;
while (getline(infile, stringSet[number])){
    number++;
}

//nBit, mBit값 저장
nBit = number;
mBit = stringSet[0].length();

//input파일 close
infile.close();

cout << "n : " << nBit << endl << "m : " << mBit << endl;
cout << "DeadLine Time : " << DEADLINE << "초" << endl;

//정답 스트링과 그에 해당하는 d-value 저장하는 pair
pair<string, int> anspair;

//stringSet 출력. 초기 개발당시 디버그용
//printAllStringSet(stringSet);

//Greedy 알고리즘으로 구한 정답저장.
anspair = Greedy(stringSet);
returnAnswer[0] = anspair.first;
returnDvalue[0] = anspair.second;

cout << '.';
timeoverIdx++;

//Branch and Bound 알고리즘으로 구한 정답저장.
anspair = Branch(stringSet);
returnAnswer[1] = anspair.first;
returnDvalue[1] = anspair.second;

cout << '.';
timeoverIdx++;

//BruteForce 알고리즘으로 구한 정답저장.
ofstream file("output.txt", ios::out);
anspair = BruteForce(stringSet);
returnAnswer[2] = anspair.first;
returnDvalue[2] = anspair.second;

//d-value가 제일 작은 정답 출력
minanswer = *min_element(returnDvalue, returnDvalue+3);
for(int i = 0; i <= timeoverIdx; i++){
    if(returnDvalue[i] == minanswer){
        file << "string t: " << returnAnswer[i] << endl;
    }
}

```

```

        file << "d-value : "<< returnDvalue[i] << endl;
        break;
    }
    /*
    file << "string t: "<< returnAnswer[i] << endl;
    file << "d-value : "<< returnDvalue[i] << endl;
    */
}
cout << '.';

//프로그램 종료시간 측정.
time_t end = time(NULL);
cout << "프로그램 실행시간 : " << end - start << " 초" << endl;

return 0;
}

```

- Greedy Algorithm

```

pair<string, int> Greedy(string * stringSet){
    //모든 n개의 string중에서 특정 m번째 자리수 중에서 0, 1 두가지중에 뭐가 더 많은지 저장
    int check[2];

    //m Bit 각 자리수마다 탐색한다.
    for(int i = 0; i < mBit; i++){
        //실행도중 시간오버되면 지금까지 구한 답 출력하고 강제 종료.
        timeover(stringSet, DEADLINE);

        check[0] = 0; check[1] = 0;

        for(int j = 0; j < nBit; j++){
            if(stringSet[j][i] == '0'){
                check[0] ++;
            }
            else{
                check[1] ++;
            }
        }

        //더 많은 것을 선택해야 그 때의 해밍 디스턴스를 최소화 할 수 있다.
        if(check[0] > check[1]){
            ansString.replace(i,0,"0");
        }
        else{
            ansString.replace(i,0,"1");
        }
    }

    //정답 return.
    return make_pair(ansString, HammingDistance(stringSet, ansString));
}

```

- Brute Force Algorithm

```

pair<string, int> BruteForce(string * stringSet){

```

```

// mBit가 20자리 이상되면 2^20 * nBit이상의 연산과정 -> 프로그램 오류
if(mBit > 20){
    return make_pair("NULL", mBit*2);
}

int minVal = mBit;

long idx = pow(2,mBit);
string bruteString;

for(long j = 0; j < idx; j++){
    //실행중 타임오버나면
    timeover(stringSet, DEADLINE);

    bruteString.clear();
    if(idx == (j+1) * 8)
        cout << "12.5%" << endl;
    else if(idx == (j+1) * 4)
        cout << "25%" << endl;
    else if(idx == (j+1) * 2)
        cout << "50%" << endl;

    //bruteString에 j값에 해당하는 이진수를 넣는다.
    for (int i = mBit-1; i >= 0; i--){
        string s = std::to_string((j >> i & 1 ? 1 : 0));
        bruteString.append(s);
    }
    //bruteString의 해밍거리를 구해서, 현재 최소값보다 작으면 바꾼다.
    int temp = HammingDistance(stringSet, bruteString);
    if(minVal > temp){
        minVal = temp;
        ansString = bruteString;
    }
}
//정답 반환
return make_pair(ansString, HammingDistance(stringSet, ansString));
}

```

- Branch and Bound Algorithm

```

pair<string, int> Branch(string * stringSet){
    //재귀로 구현했으나, 재귀함수 호출 오버플로우가 나서 스택으로 구현함
    stack<pair<string, int>> st;

    int temp;
    int minVal = mBit;
    string est, sel0, sel1, sel2, sel3;;
    int idx, revidx;

    //초기 예측값은 모두 "0"인 mBit string
    for(int i=0;i<mBit ;i++)
        est.append("0");
    //BackTracking 방법이므로, 다시 돌아와야 하기에 스택에 넣어줌
    st.push(make_pair(est,0));
}

```

```

while(!st.empty()){
    //하다가 시간초과 되면 종료
    timeover(stringSet, DEADLINE);

    //stack의 맨위 꺼내기
    est = st.top().first;
    idx = st.top().second;
    st.pop();
    //cout << "pop" << endl;

    //일반적인 재귀함수 호출시 return하는것과 동일
    if(idx == est.length())
        continue;

    int check[2] = { 0 };

    sel0 = est;
    sel1 = est;
    //하나는 0, 하나는 1로 예측
    sel0[idx] = '0';
    sel1[idx] = '1';

    //각각 예측한것의 해밍거리를 계산함
    check[0] = HammingDistance(stringSet, sel0);
    check[1] = HammingDistance(stringSet, sel1);

    //0을 넣은게 1보다 작으면, 0을 넣은것으로 계속 branch함
    if (check[0] < check[1]){
        if(minVal > check[0]){
            minVal = check[0];
            ansString = sel0;
        }
        st.push(make_pair(sel0, idx+1));
    }

    //같으면 둘다 branch
    else if(check[0] == check[1]){
        if(minVal > check[0]){
            minVal = check[0];
            ansString = sel0;
        }
        st.push(make_pair(sel0, idx+1));
        st.push(make_pair(sel1, idx+1));
    }

    //1이 더 작으면 1을 넣은것으로 branch
    else{
        if(minVal > check[1]){
            minVal = check[1];
            ansString = sel1;
        }
        st.push(make_pair(sel1, idx+1));
    }

}

return make_pair(ansString, HammingDistance(stringSet, ansString));
}

```

4. 전체 구현 알고리즘 설명

메인문에서 그리디, 브랜치, 브루트 포스 순으로 알고리즘을 실행한다.

그리디 방법은 매우 빠르지만 많이 정확하지 않으므로, 처음 실행해서 최악의 경우 다른 방법이 시간내에 실행되지 않을 경우, 답으로 출력한다.

그 다음 Branch and Bound를 사용해서 더 정확한 답을 찾는다.

Branch and Bound는, 처음 예상 정답을 넣는다. 위 코드에서는 모든 비트가 0인 mBit string이 초기 예측값이다. 그 후 처음부터 하나씩 0 또는 1을 넣어가면서 재귀적으로 다음으로 넘어간다. 이때 0, 1 중 해밍 디스턴스가 작은것만 재귀적으로 다음으로 넘어가게 된다. (해밍 거리가 큰 것은 가지친다.) 또한 0, 1 중 해밍 디스턴스가 같다면 둘 다 스택에 넣어서 가지를 치지않고 둘다 Branch 하게 된다.

그래서 마지막 자리 까지 도달하게 되면 return(위에서는 continue로 구현)하게 되고, 스택이 빌때까지 계속해서 연산을 하면서 최소 d-value를 갱신하고, 그때의 정답 string을 저장한다.

마지막으로 브루트 포스방법은, 모든 가지수를 다 넣어보는건데, 2^{20} 개 이상의 경우에는 현실적인 구현이 어렵고 오류가 발생할 확률이 높아서 실행하지 않으며, 그 이하의 경우에는 모든 가지수를 검증하면서, 데드라인 시간이 지나면, 종료해서 지금까지 구한 답 중 최소 d-value를 출력한다.