

컴퓨터 알고리즘과 실습 8주차 과제

2016112158 김희수

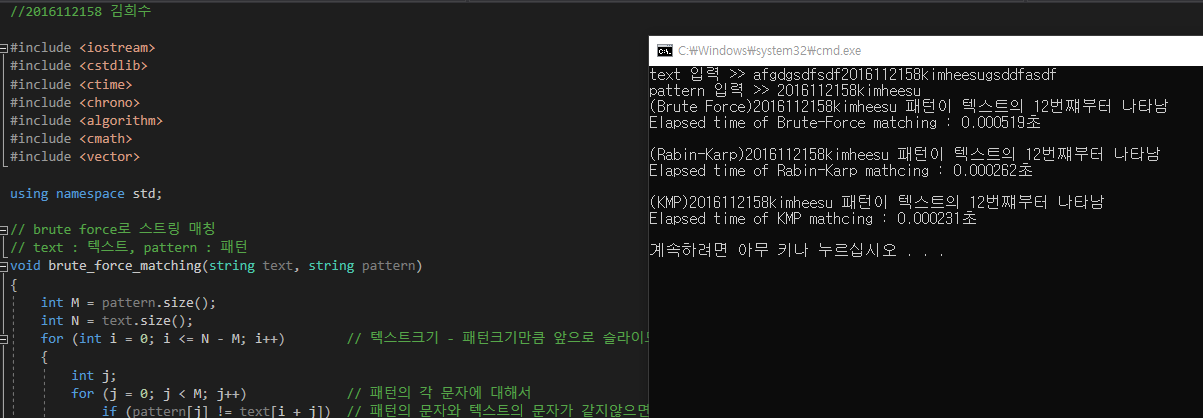


[날짜]

[회사 이름]

[회사 주소]

(실행화면)



스트링 매칭의 세가지 알고리즘을 구현하였다. 첫번째는 직선적으로 문자하나하나씩 모두 비교하는 브루트 포스 알고리즘이다. pattern의 크기와 text의 크기를 변수에 저장하고 비교는 (text크기-pattern크기)번 동안 pattern번씩 이루어진다. pattern번의 반복이 이루어질 때는 곧 패턴의 문자하나하나와 텍스트의 문자하나하나를 비교하는 과정이다. 이 비교과정에서 패턴과 텍스트가 한번이라도 다르다면 그 비교를 중지하고 슬라이드를 한 칸이동하여 다음비교를 수행한다.

다음은 Rabin-Karp알고리즘이다. 개인적으로 이해도 쉽고, 수학적인 발상도 맘에들고, 실행속도도 나쁘지 않은 알고리즘이라 생각한다. 스트링을 숫자 값으로 바꾼뒤 해시값을 계산해 매칭한다는게 주요 골자인데, 이번 과제에선 학번(10진수)와 영문소문자(26진수)를 입력할 것이니 36진법의 알파벳을 사용한다고 보면 되겠다. 처음엔 패턴의 길이와 텍스트의 길이를 포함한 변수를 선언해주자. 그 후 호너 방법을 통해 p, t를 계산한 후 텍스트의 각 위치별로 비교를 수행한다. p와 t가 같을 때, 즉 hash값이 동일한 경우에 한해서 문자하나하나씩 세부적으로 비교한다. 또한, 다음 위치의 비교를 위해 점화식을 이용해서 다음 t값을 계산한다.

KMP 알고리즘은 suffix/prefix 매칭을 통해 불필요한 비교 반복을 없앤다고 한다. 주요 골자는 먼저 하나하나씩 비교한 후, 다른게 나오면 한칸 뒤로 가서 suffix/prefix 매칭을 한 후, 패턴의 prefix다음에 오는 글자와 텍스트이 suffix 다음글자를 비교한다. 이 때 두 글자가 다르면 이 워드는 다른 워드이므로 다음 워드로 넘어가서 처음부터 반복한다.

이를 위해 매칭 전에 suffix와 prefix가 어디서 일치하는지 알기 위한 최대 접두부 테이블을 만들어야 한다. 최대 접두부 테이블은 j와 i, 그리고 index를 이용한다. j는 일단 인덱스0에 위치하고 i는 j다음부터 즉, 인덱스1부터 위치한다. j와 i의 문자가 다르면 SP에는 0이 기록되고 i는 다음칸으로 이동한다. j와 i의 문자가 같을때마다 SP에는 하나씩 증가시켜 기록하고 j가 다음칸으로 이동한다. 이렇게 SP를 채우다가 이전에 나왔던 문자가 pattern에서 또 나온다면 j를 한 칸뒤로 보내고 그때의 SP값을 본 후 이 SP값에 해당하는 index로 j를 보낸다.

구현을 보자. 최대 접두부 테이블 SP가 만들어졌으면 이제 비교를 수행한다. pattern과 text의 문자가 같을때는 인덱스를 증가시키면서 넘어가되, 다른게 나왔으면 text의 suffix/ pattern prefix 다음의 문자를 비교하여 j를 뒤로 건너뛰어 준다.

**추가과제**

※프로그램 실행시 100,000,000.txt파일을 제외한 txt파일은 모두 삭제하고 실행시켜주세요.

추가과제는 앞에서 사용한 스트링 매칭알고리즘을 사용하여 10,000~100,000,000개로 랜덤한 숫자로 이루어진 스트링을 5,10,15,…,30개의 랜덤한 숫자로 이루어진 패턴으로 매칭하는 것이다.(10부터 시작하기엔 text의 길이가 1억개가 아닐 때 출력되는 경우가 너무 적었다) Rabin-Karp를 수행할 때 mod 연산에 필요한 해시값은 2로 주었다.

먼저 길이가 1억인 스트링을 저장하고 있는 100,000,000.txt를 만든다. make\_text\_file()함수가 사용되었다. 그 후, 100,000,000.txt에서 1천만개, 1백만개, 1십만개, 1만개씩 끊어서 만든 substring을 가지는 파일을 각각 생성한다. 그러면 모든 텍스트파일에는 공통적으로 들어가는 길이가 30이하인 스트링이 있을 것이다. 편의를 위해 substring은 100,000,000.txt파일의 처음부터 읽어왔다. 패턴은 파일에 존재하는 스트링으로 만들것이다. 이렇게 하는 이유는 매칭의 편리함을 위해서이다. 본래 의도했던 것은 txt파일에도 전부 랜덤하게 스트링을 만들고, 패턴도 전부 랜덤하게 하려고 했다. 하지만 이럴 경우 txt파일마다 매칭해야하는 패턴이 달랐고, 파일마다 모두 다른 스트링을 가지고 있어서 실험의 불편함이 존재했다. 패턴을 파일에 존재하는 스트링으로 만들지 않으면 길이10이상의 패턴들은 파일에 나타나지 않는 경우가 있엇다.

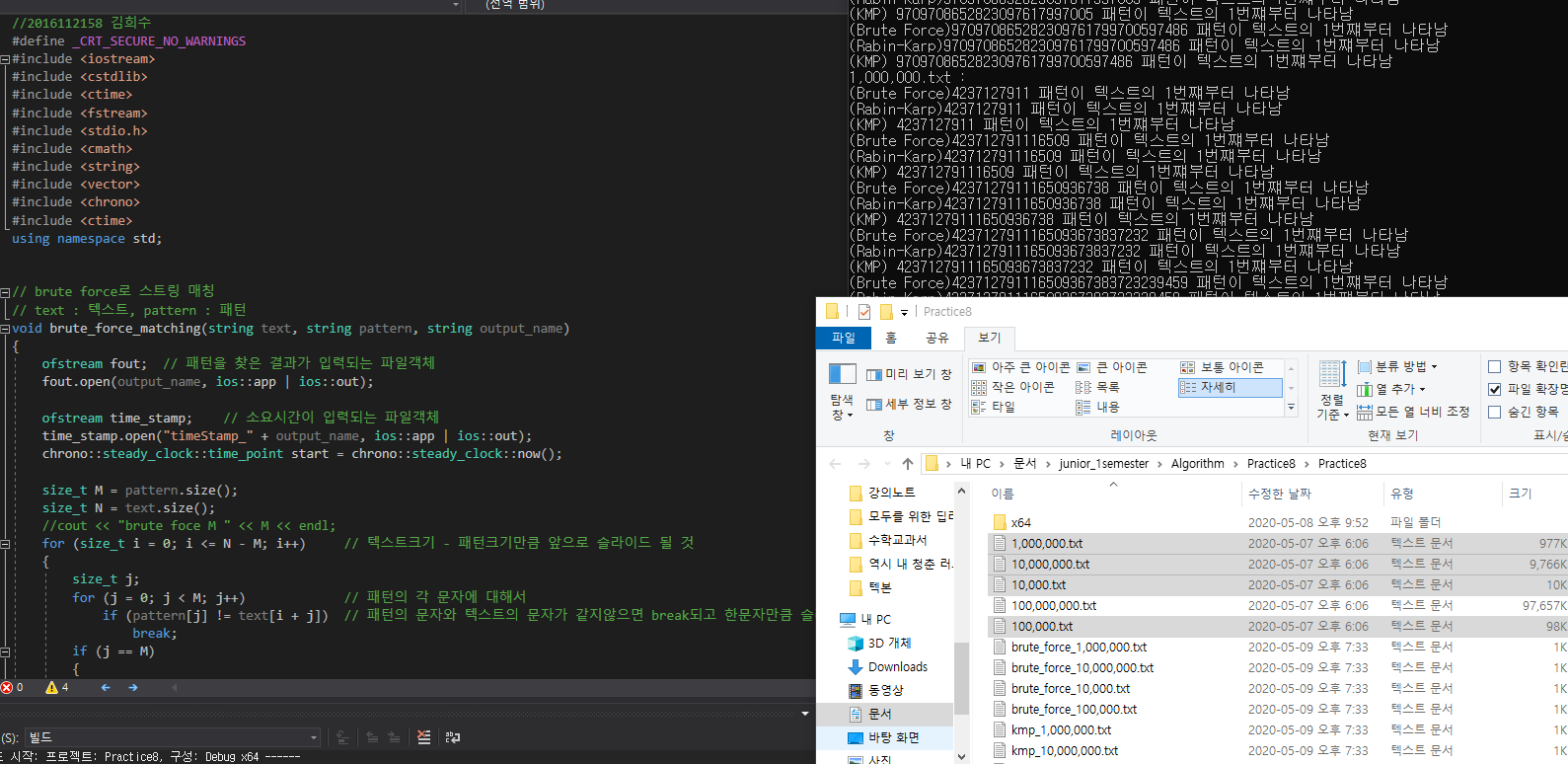
만약 100,000,000.txt의 서브스트링으로 10,000,000.txt이하의 텍스트파일을 만드는게 본래 문제에서 요구하던 것이 아니라면 그냥 make\_text\_file()함수를 이용해서 10,000,000.txt이하를 만들어주면 된다. 그렇게 해도 get\_result()함수에선 패턴을 얻을 땐 그 txt파일에 분명 존재하는 문자열을 패턴으로 저장한다. 따라서 비교가 제대로 이루어지는지 실험을 제대로 할 수 있다.

각 알고리즘마다 fout과 time\_stamp라는 파일스트림을 생성해 fout에는 콘솔창에 나타나는 것과 동일하게 txt파일에 써지도록 하였고, time\_stamp에는 각 알고리즘이 수행되는데 소요된 시간을 출력하였다.

make\_text\_file함수로 스트링의 길이를 파라미터로 받아 그 만큼의 길이를 가지는 스트링을 파일에 입력하였고 get\_result함수에선 txt파일의 이름을 파라미터로 받아서 inFile이라는 파일객체에 해당 파일을 읽어와 text 스트링객체에 저장했다. 하지만 make\_text\_file로 만들어지는 파일은 100,000,000.txt파일 뿐이다.

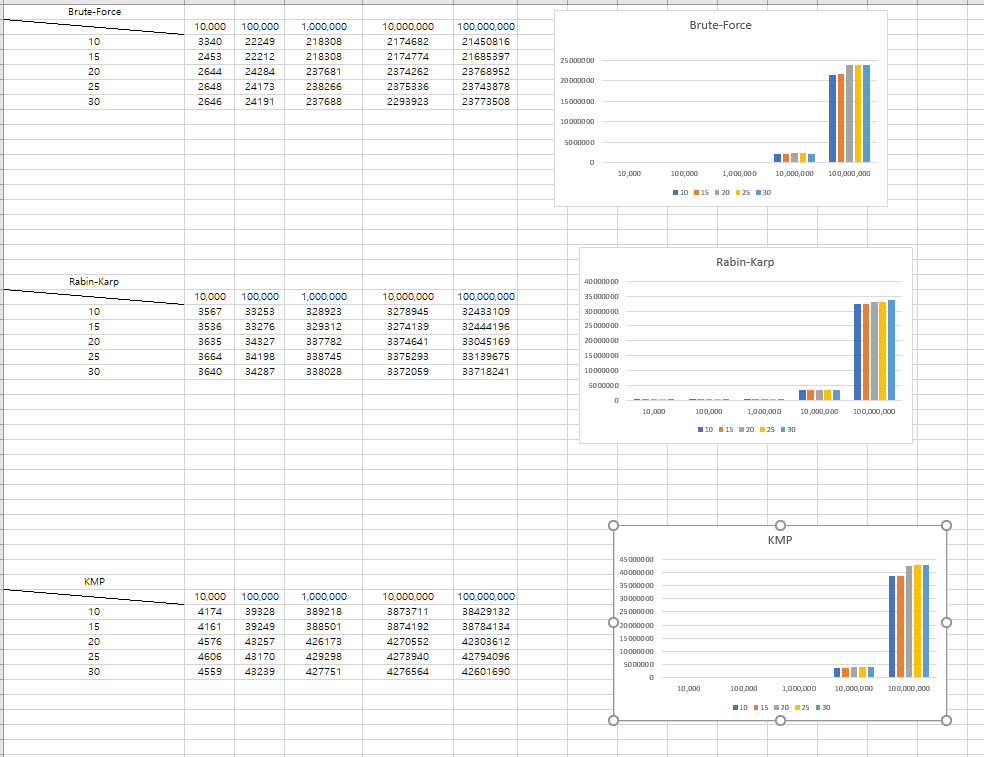
패턴은 랜덤하게 만들려다가 랜덤하게 만들경우 그 패턴이 txt파일에 존재하지 않는 경우 알고리즘의 소요시간 비교를 하기가 애매해지므로, txt파일에 존재하는 패턴을 사용하였다.

(실행화면)



※ 밑의 차트와 프로젝트에 저장된 txt파일의 데이터는 다소 다를 수 있습니다.

밑의 차트를 작성할 당시 단위를 마이크로초로 하였으나, 현재 코드내에선 초로 수정함



엑셀로 데이터를 표와 그래프로 만들었다. 하지만 의문점이 있다. 위의 그래프에선 Brute-Force의 경우가 소요시간이 가장 짧다고 나오는데 이는 일반적인 인식과 다르다. 배운대로라면 Rabin-Karp나 KMP가 Brute-Force보다 소요시간이 짧아야한다. 하지만 더 오래 걸린다. 예상되는 원인으론 Rabin-Karp나 KMP알고리즘의 소요시간에 이전에 수행된 알고리즘의 소요시간이 누적됬을 가능성이 있다. 즉, Rabin-Karp알고리즘의 소요시간으로 출력된 것이 사실은 (Brute-Force의 소요시간+Rabin-Karp의 진짜 소요시간)라는 것이다. 하지만 코드상에서 그런 문제는 보이지 않는다….