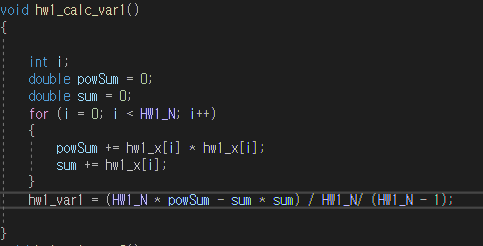
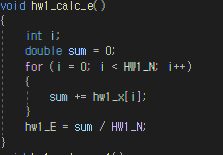
[고소실\_10주차과제]2반\_20161595\_배성현

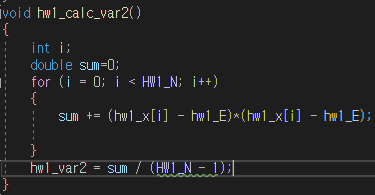
**숙제 1.**



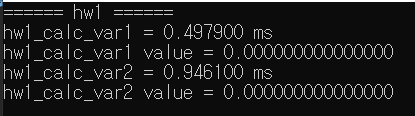
먼저 위의 왼쪽코드는 분산을 구하는 방법 중 하나인 오른쪽 위의 식을 사용한 것이다. 데이터들의 합을 저장하는 변수 sum과 데이터의 제곱의 합들을 저장하는 변수 powSum은 double type으로 선언하여 주었는데 그 이유는 샘플데이터들을 더하는 경우나, 제곱하여 더하는 경우 값이 float이 저장할 수 있는 범위를 넘어 갈수 있기 때문에 double을 사용하여 주었다.

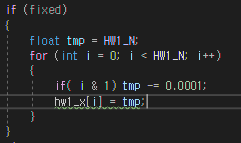
위의 왼쪽 코드는 오른쪽 위의 식을 사용하여 init\_hw1에 의해 만들어진 샘플데이터에 대하여 평균을 구하는 코드이다. 위의 분산식의 powSum과 sum을 double타입으로 선언한 것과 같은 이유로 여기서의 변수 sum도 double type으로 선언하여 주었다.

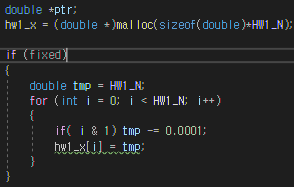
그리고 위의 왼쪽 코드는 오른쪽 위의 식을 사용하여 샘플데이터에 대하여 분산을 구하는 코드이다. 이 때도 위의 경우에서와 같은 이유로 double변수 sum을 사용하여 주었다.



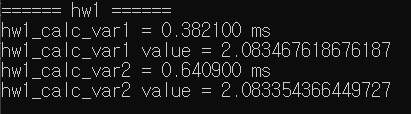
위의 결과는 1을 init\_hw1()의 매개변수로 넘겨주게 되었을 때 나온 결과이다. 두 경우 다 분산이 0이 나왔다.



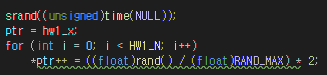
위의 코드를 보았을 때 1이 init\_hw1에 들어가게 되면 위의 if문 안의 내용이 수행이 되는데 hw1\_x[i]에 tmp를 계속해서 0.0001씩 뺀 값을 집어넣게 된다. 하지만 tmp가 float이기 때문에 반복문에서 조건을 만족하는 경우에 tmp-=0.0001을 해주어도 tmp 값은 계속 HW1\_N의 값인 100000을 유지하게 된다. 따라서 hw1\_x[i]에 들어있는 모든 데이터가 100000값을 가지고 있기 때문에 위의 결과에서 분산이 0이 나오게 된다.



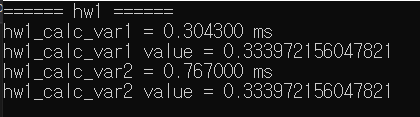
이를 정확하게 나오게 하기 위해서는 위와 같이 100000의 값을 저장하기 위해 값이 저장되는 변수들을 double형으로 바꿀 필요가 있다.

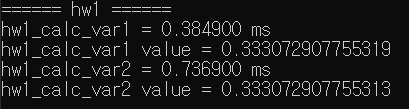


위의 결과는 값이 저장되는 변수들을 모두 double로 바꾸어 연산하였을 때 얻은 분산이다. double을 사용하였기 때문에 좀 더 정확하게 수를 표현할 수 있어 0이 아닌 위와 같은 값이 나오게 되었고, 두 분산 값이 일치하는 것을 확인 할 수 있다.



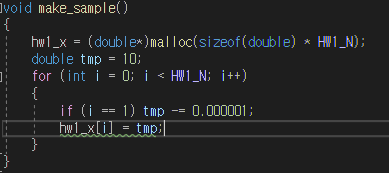
0이 init\_hw1에 들어가게 되면 위의 내용이 수행이 되는데 rand()값을 RAND\_MAX로 나누어 2를 곱하기 때문에 생성되는 데이터들은 0~2사이의 값을 가지게 된다.

float연산

 double연산

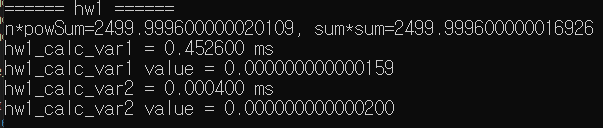
위의 결과는 이러한 0을 init\_hw1()의 매개변수로 넘겨주고 float 연산과 double연산을 이용하였을 때 나온 각각의 결과이다. Float과 double 모두 첫 번째 방법과 두 번째 방법을 통해 구한 분산이 일치하게 나온 것을 확인할 수 있다.

위 결과들의 시간을 보았을 때 모든 경우에 있어서 를 통하여 구한 분산의 시간이 오래 걸린 것을 확인할 수 있다. 그 이유는 코드를 보면 알 수 있는데 의 방법에서는 평균을 구하는 과정에서 HW1\_N만큼의 실수 덧셈을 반복하게 되고 또 평균을 이용하여 분산을 구하는 과정에서 3번의 실수 덧셈, 한 번의 실수 곱셈을 HW1\_N만큼 반복하게 된다. 그에 반해 의 방법에서는 2번의 실수 덧셈, 한 번의 실수 곱셈을 HW1\_N만큼 반복하게 되므로 상대적으로 의 방법의 실수 연산이 더 적다. 따라서 의 방법이 의 방법에 비하여 빠르다. 그리고 위의 분산 결과들에 있어서는 의 방법과 의 결과가 같게 나왔는데 이는 비슷한 수끼리의 뺄셈이 발생하지 않았기 때문이다. 일반적으로 의 방법에서 비슷한 수끼리의 뺄셈이 발생하여 조금 더 위험하다고 알려져 있는데 그 이유는 의 방법에서 와 가 비슷한 수가 될 수 있기 때문이다.

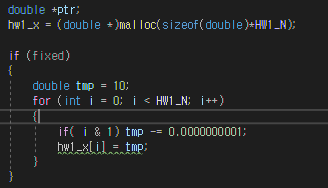
 

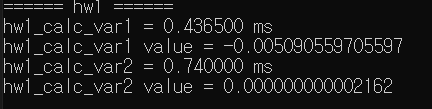
(main 함수)

비슷한 수끼리의 뺄셈의 경우를 확인하기 위해 위의 코드와 같이 HW1\_N을 5로 바꾸고, 데이터들의 표준편차가 작게 만들어지도록 make\_sample()이라는 함수를 만들어 주었다. 그리고 해당 경우를 테스트할 때에는 위와 같이 HW1\_N을 5로 바꾸고, 메인 함수에서 init\_hw1()함수를 주석 처리하여 주고, make\_sample()함수를 주석에서 풀어주었다.



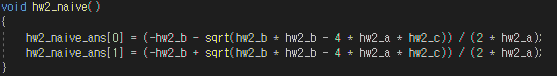
그리고 실제로 이를 기준으로 테스트 해보았을 때 위와 같은 결과를 얻을 수 있었는데 이를 보게 되면 두 개의 분산 값이 다른 것을 확인 할 수 있다. 그리고 위와 같이 의 식에서의 (n\*powSum)와 (sum\*sum)에 해당하는 값을 출력하여 보았을 때 여기서 비슷한 수끼리의 뺄셈이 발생했다는 것을 확인할 수 있었다. 최종적으로 실제 분산 값을 계산해 보면 2E-13 값이 나오므로 의 식은 잘못 구한 것을 확인할 수 있고, 의 식은 잘 구한 것을 확인 할 수 있다. 따라서 의 식이 더 위험하다는 것을 알 수 있다.



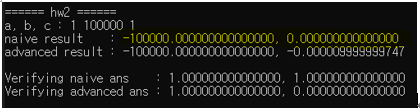


또한 위와 같이 init\_hw1의 코드를 살짝 바꾸어서 double로 계산하였을 때, 의 방법을 이용하는 경우에 분산 값이 음수가 나온 것을 확인 할 수 있다. 이는 와 의 계산에 있어 각각의 반올림 오차가 발생하게 되고 그로 인해 두 수가 비슷한 경우, 의 값이 더 크게 나올 수 있기 때문이다.

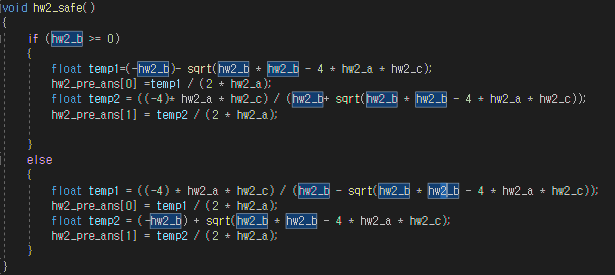
**숙제 2.**



위의 코드는 중학교 때 배운 아래와 같은 근의 공식을 사용하여 짜 주었다.



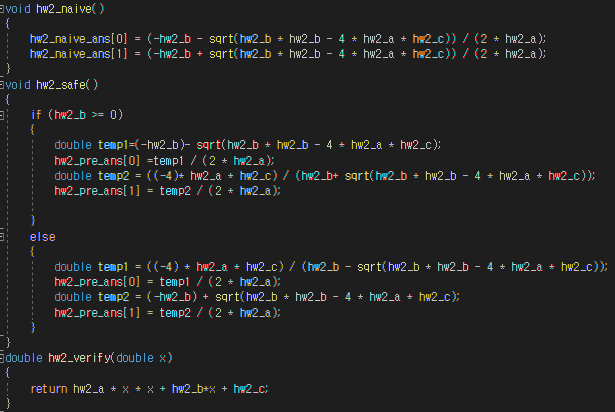
위의 naïve 코드를 이용하여 근을 찾게 되면 먼저 b가 양수일 때, b의 값이 굉장히 크고 a\*c의 값이 굉장히 작은 경우에 의 식에서 루트안의 식이 b의 값과 굉장히 유사해져 비슷한 수끼리의 뺄셈이 일어나게 된다. 따라서 위의 결과의 형광펜 부분의 오른쪽 해와 같이 나오게 되어 문제가 발생하게 된다. 또한 위의 결과의 형광펜 부분의 왼쪽 해의 경우에는 비슷한 수끼리의 뺄셈 문제 때문이 아닌 자료형이 float이기 때문에 data를 정확하게 담을 수 없어 문제가 발생하게 되고 따라서 Verifying naïve ans를 보게 되면 잘못된 것을 확인할 수 있다.



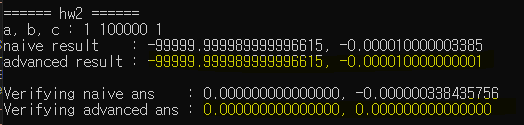
그리고 먼저 비슷한 수끼리의 뺄셈에 대한 문제의 해결방법으로 위의 코드와 같이 유리화한 식을 사용해주는 방법이 있다. 유리화의 경우, 먼저 b가 양수인 경우에, 의 식에서 비슷한 수끼리의 뺄셈 문제가 발생할 수 있기 때문에 이를 로 바꾸어주게 되고, b가 음수인 경우에는, 에서 문제가 발생할 수 있기 때문에 이를 로 바꾸어 주게 된다. 그리고 위의 결과 그림의 advanced result의 오른쪽 해를 보게 되면 naive에서는 해가 0.00000~이었던 것이, -0.000009999999747로 바뀐 것을 확인 할 수 있고 또 이를 대입한 결과인 Verifying advanced ans의 오른쪽 값을 보게 되면 0.0으로 값이 잘 나온 것을 확인할 수 있다. 따라서 위와 같이 유리화를 통하여 근의 공식에서의 비슷한 수끼리의 뺄셈에 의해 발생하는 문제를 해결할 수 있다.

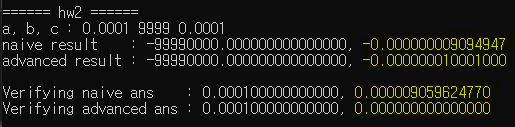




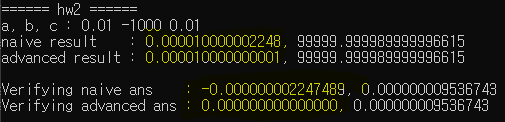


그리고 나서 float에 의해 data손실이 발생했던 문제를 해결해주기 위하여 위와 같이 모든 부분을 double로 바꾸어 주었고 같은 a, b, c 값에 대하여 계산하였을 때 아래와 같은 결과를 얻게 되었다. 그리고 아래의 결과를 보게 되면, 앞의 float에 의한 문제가 해결되어 Verifying advanced부분에서 두 해에 대한 값 모두 0이 나온 것을 확인할 수 있다.



다른 경우

또 다른 경우를 보게 되었을 때 naïve result의 오른쪽 해를 구하는 과정에서 비슷한 수끼리의 뺄셈이 발생하게 되고, 따라서 verifying naïve ans를 보았을 때 0값이 아닌 것을 확인 할 수 있다. 그리고 이러한 문제를 해결하도록 해준 advanced result의 오른쪽 해의 경우에는 Verifying advanced ans의 경우 0값이 잘 나온 것을 확인 할 수 있다. 그리고 왼쪽 해의 경우에는 naïve와 advanced 둘 다 제대로 된 값을 구할 수 없었는데 그 이유는 위의 이유와 마찬가지로 double에는 값을 담을 수 없는 데이터의 손실 문제 때문이라고 할 수 있다.

다른 경우

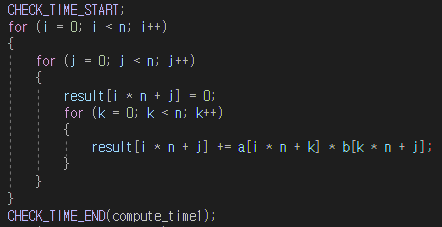
3번째 경우로서 b가 음수인 경우를 살펴보았다. 이번 경우에서는 에 의하여 비슷한 수끼리의 뺄셈 문제가 발생하기 때문에 naïve result의 왼쪽 해를 보면 알 수 있는데 왼쪽 해에 따른 Verifying naïve ans값이 0이 아닌 값을 나온 것을 확인할 수 있다. 그리고 이를 해결하도록 해준 advanced result의 왼쪽 해의 경우 Verifying advanced ans값이 0으로 잘 나온 것을 확인할 수 있다.

**숙제 3.**

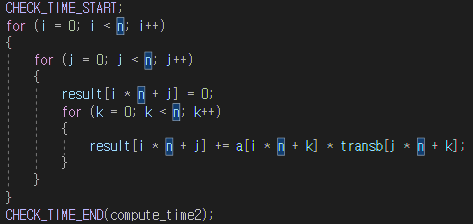
코드 최적화 방법: locality, code motion, strength reduction, dead code elimination, common subexpression elimination

1. Locality

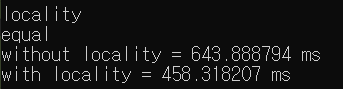
가장 먼저 아래와 같이 locality를 이용한 방법이 있다. c언어의 경우에는 row major order로 배열이 저장되기 때문에 배열에 어떠한 순서로 접근하는지에 따라 locality에 의해 속도 영향을 받게 된다.

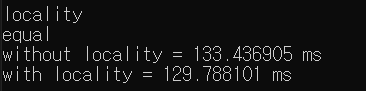


위의 코드의 경우는 locality를 고려하지 않은 것이다. 행렬의 곱셈을 함에 있어서 오른쪽에 있는 행렬, 즉 b의 경우에는 row를 따라 탐색하는 것이 아닌 column을 따라 값을 가져오기 때문에 cache miss가 더 자주 발생하고 따라서 속도가 아래의 코드에 비해 상대적으로 느리다.



위의 코드의 경우는 locality를 고려한 코드이다. 이는 오른쪽 행렬의 전치행렬을 미리 계산해서 배열에 저장하여 놓고 이 배열을 행렬의 곱셈에 이용함으로써 오른쪽 행렬 역시 row order로 접근되게 하여 조금 더 cache hit가 많이 발생할 수 있도록 하였다. 따라서 이렇게 locality를 고려한 코드가 고려하지 않은 코드보다 빠르다.

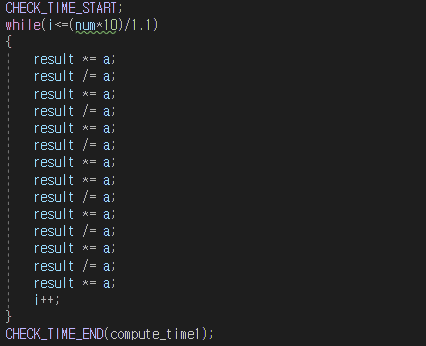
Debug mode

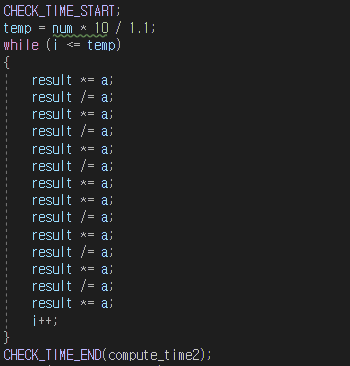
Release mode

Debug mode와 Release mode 둘 다 locality를 고려하지 않은 것에 비해 locality를 고려한 것이 속도가 더 빠르게 나왔다. 다만 release모드에서는 컴파일러가 기계어로 변환하면서 소스코드를 최적화를 진행하여 실행 파일의 크기를 줄이고, 속도 역시 빨라지기 때문에 속도 차이가 많이 보이지 않았다. 또한 Debug mode에 비해 최적화된 Release mode에서 locality를 고려하지 않은 코드와 고려한 코드 둘다 빨라진 것을 확인 할 수 있다.

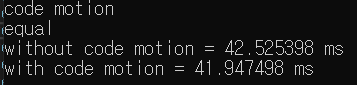
1. Code motion

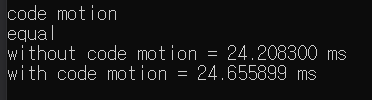
Code motion방법은 프로그램의 알고리즘을 수정하지 않고 프로그램의 코드의 일부를 옮김으로써 최적화하는 것을 말한다.





위의 코드 중 첫 번째 코드는 code motion을 적용하기 전의 코드이고, 두 번째 코드는 code motion을 적용한 후의 코드이다. 첫 번째 코드의 while문 안의 조건을 체크하는 num\*10/1.1부분은 값의 변화가 일어나지 않지만 체크를 위해서 loop를 돌 때마다 수행되게 되는데 두 번째 코드에서 이를 while문 밖으로 옮기는 code motion을 취해주어 코드의 사이즈를 줄이고 따라서 이를 통해 속도의 향상을 이루어 낼 수 있다.

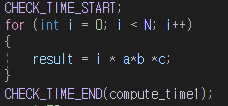
Debug motion

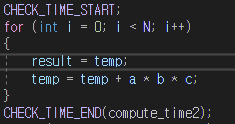
Release mode

Debug mode를 보면 code 모션을 진행한 경우에 있어서 속도가 미세하게 빨라진 것을 확인할 수 있다. 다만 release모드에서는 최적화가 되기 때문에 code motion을 한 것을 최적화한 것보다 code motion을 안 한 것을 최적화한 코드가 더 빠르게 나왔다. 그리고 Debug mode에 비해 최적화된 Release mode에서 code motion을 한 경우와 하지 않은 경우 둘다 빨라진 것을 확인 할 수 있다.

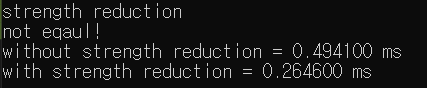
1. Strength reduction

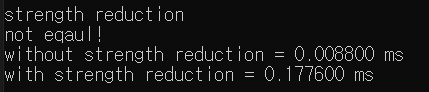
Strength reduction방법은 cost가 많이 드는 operator를 더 cost가 적게 들되 프로그램의 변화가 없도록 하는 operator로 교체하는 것을 의미한다.





위의 코드 중 첫 번째 코드는 strength reduction을 적용하기 전의 코드이고, 두 번째 코드는 strength reduction을 적용한 후의 코드이다. 첫 번째 코드의 i\*a\*b\*c부분에서 (i)\*(a\*b\*c)의 \*는 cost가 큰 operator이기 때문에 두 번째 코드에서는 더 연산의 cost가 적은 +를 사용함으로써 프로그램의 속도 향상을 이루어 낼 수 있다.

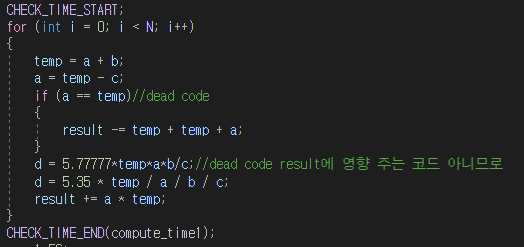
Debug mode

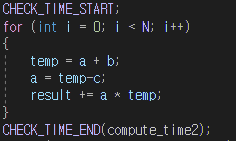
Release mode

Debug mode를 보면 Strength reduction을 진행한 경우에 있어서 속도 현저하게 빨라진 것을 확인 할 수 있다. 다만 release모드에서는 최적화가 되기 때문에 strength reduction을 한 것을 실행한 것보다 strength reduction을 안 한 것을 실행한 코드가 더 빠르게 나왔다. 그리고 Debug mode에 비해 최적화된 Release mode에서 strength reduction을 한 경우와 하지 않은 경우 둘 다 빨라진 것을 확인 할 수 있다. 또한 strength reduction의 경우에는 result의 결과값이 서로 다르게 나온 문제가 발생했는데 float은 수를 완전히 표현할 수 없어 근사하여 표현하게 되고 부동소수점 반올림 오차가 발생하게 되는데 strength reduction에서는 곱셈을 하던 것을 덧셈으로 하기 때문에 이러한 반올림 오차가 누적되어 결과값이 서로 달라지게 된다.

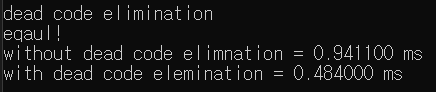
1. Dead code elimination

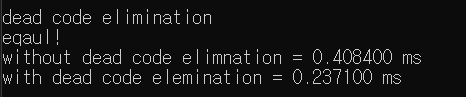
Dead code란 프로그램에서 쓰이지 않는 코드를 의미한다. 이러한 코드들은 제거될 수 있는데 Dead code elimination방법은 이러한 코드들을 제거하여 최적화하는 것을 말한다.





위의 코드 중 첫 번째 코드는 dead code elimination을 적용하기 전의 코드이고, 두 번째 코드는 dead code elimination을 적용한 후의 코드이다. 첫 번째 코드를 보면 a=temp-c부분에서 c는 0이 아니기 때문에 if문안의 내용은 쓰이지 않는 코드라는 것을 알 수 있고, 또 d에 값을 대입하는 if문 밑의 두 d에 관한 코드는 result에 아무런 영향을 주지 않으므로 쓰이지 않는 코드라는 것을 알 수 있다. 따라서 두 번째 코드에서는 이러한 쓰이지 않는 코드(dead code)들을 다 제거함으로써 프로그램의 속도 향상을 이루어 낼 수 있다.

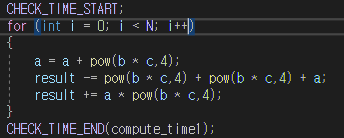
Debug mode

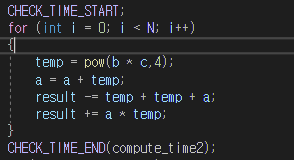
Release mode

Debug mode와 Release mode 둘 다 dead code elimination을 수행한 것이 속도가 더 빠르게 나왔다. 또한 Debug mode에 비해 최적화된 Release mode에서 dead code elimination을 한 것과 하지 않은 코드 둘 다 빨라진 것을 확인 할 수 있다.

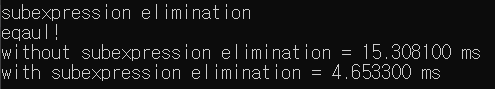
1. Common subexpression elimination

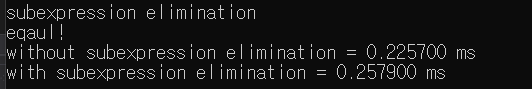
Common subexpression elimination방법은 공통된 코드를 중복 실행하지 않고 한 번만 실행하고 이 후부터 그 값을 이용하여 최적화하는 것을 말한다.





위의 코드 중 첫 번째 코드는 common subexpression elimination을 적용하기 전의 코드이고, 두 번째 코드는 common subexpression elimination을 적용한 후의 코드이다. 첫 번째 코드를 보면 pow(b \* c, 4) 부분은 loop안에서 굉장히 많이 나타나는 공통된 코드라는 것을 알 수 있다. 따라서 두 번째 코드에서 이를 먼저 계산하고 값을 temp에 저장한 뒤 이후의 같은 expression에 temp를 사용하여 공통된 코드를 줄임으로써 속도의 향상을 이루어 낼 수 있다.

Debug mode

Release mode

Debug mode를 보면 common subexpression elimination을 진행한 경우에 있어서 속도가 현저하게 빨라진 것을 확인 할 수 있다. 다만 release모드에서는 최적화가 되기 때문에 subexpression elimination한 것을 실행한 것보다 subexpression elimination을 안 한 것을 실행한 코드가 더 빠르게 나왔다. 그리고 Debug mode에 비해 최적화된 Release mode에서 subexpression elimination을 한 경우와 하지 않은 경우 둘 다 빨라진 것을 확인 할 수 있다.