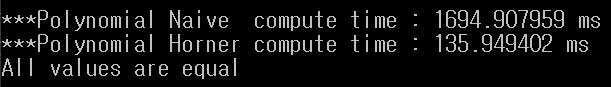
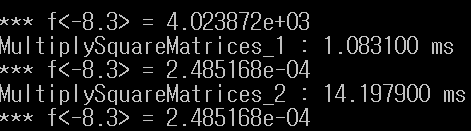
<실습>

1. 행렬 계산의 속도 향상을 위하여 (1) 자신이 적용한 방법과 (2) 어떠한 근거로 자신 이 적용한 방법이 더 효율적일지, 그리고 (3) 어떤 m 값에 대해 loop unrolling 방법이 가장 효과적이었는지를 요약하여 프로그램과 함께 제출할 것.
2. MultiplySquareMatrices\_1의 경우 행렬의 곱셈을 응집성 없이 수행한 것이라서 시간이 많이 든다. MultiplySquareMatrices\_2의 경우 시간 단축을 위해 전치행렬을 사용하여 응집성을 갖는 행렬의 곱셈을 수행한다. MultiplySquareMatrices\_3~4은 loop unrolling을 각각 8.32로 수행하여 행렬의 곱셈을 완료한다.
3. C++에서 행렬은 row major로 접근된다. 그러나 MultiplySquareMatrices\_1의 경우는 column major로 메모리에 접근하므로 캐시 메모리에 fault가 일어나 여러 번 블록에 접근해야한다. 따라서 MultiplySquareMatrices\_1보다 MultiplySquareMatrices\_2가 더 효율적이다. 또한, loop unrolling을 사용하게 되면 반복문 내에서 수행되는 비교 연산의 횟수가 줄어 시간적인 측면에서 효율적이다.
4. M=8, 32에 대해서 수행해본 결과 m=8인 경우가 더 효과적이었다.
5. Horner’s rule을 사용하여 다항식의 값을 계산하는 것이 더 효율적이라는 것을 알 수 있었다. 일반적으로 Int, float, double의 경우 덧셈과 뺄셈보다 곱셈을 수행하는데 드는 시간이 더 길고, 곱셈보다는 나눗셈의 수행 시간이 더 길다. 실습 2번에서는 다항식의 값을 계산하는 과정에서 항을 하나씩 계산하여 더하는 방법과 Horner’s rule을 사용하여 계산하는 방법에 대해 비교해 볼 수 있었다. Horner’s rule을 사용한 경우가 훨씬 소요시간이 적게 걸렸는데 이는 이 방법을 사용했을 때 곱셈의 횟수가 줄어들기 때문이다. Horner’s rule을 사용하는 경우는 y[i]를 구할 때, x[i]에 대해 거듭제곱 연산을 수행하지 않아 곱셈을 덜 할 수 있다. 다음은 실습 2번의 수행 결과이다.



1. 실습에서 사용한 x의 값은 음수이다. 따라서 테일러 급수식에 맞춰 무한 급수를 전개해보면, 양수 + 음수 + 양수 + … 꼴이다. 이 때 다음 항을 계산할수록 소수점 아래 자리수는 늘어나는데 유효숫자 자리는 정해져 있어 정확한 값을 나타낼 수 없다. Float 대신 double 자료형을 선택하면 유효 숫자의 자리 수가 늘어나 조금 더 정확한 값을 얻을 수 있다. 다음은 실습 3번의 수행 결과이다. 마지막으로 위에서 살펴봤듯 Horner’s rule을 사용하는 경우에 곱셈 횟수가 줄어 소요시간이 덜 걸릴것이다. 다음은 실습 3번의 수행 결과이다.

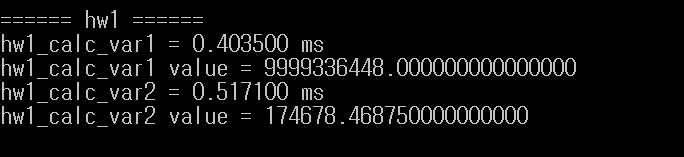


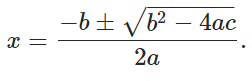
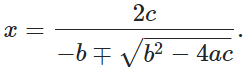
<과제>

1. 분산을 어떻게 계산한 것이 더 정확한 것으로 판단되는가? 충분히 큰 n에 대하여 두 방법 중 어떤 방법이 더 빠르게 분산 값을 계산하는가?
2. 2) 

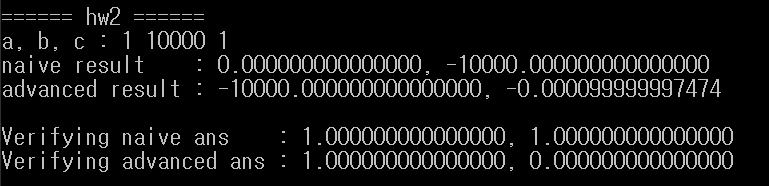
1)의 식을 이용해 분산을 계산한 것이 더 정확하다. 2)의 방법을 사용하여 분산을 구할 경우 두 식의 값이 매우 커져 정확도가 낮아질 수도 있고, 값이 비슷해져 비슷한 수의 뺄셈을 할 가능성이 있다. 수행 결과를 보면 1)번 방법을 이용해서 구할 때 더 빠른 것을 확인할 수 있었다.

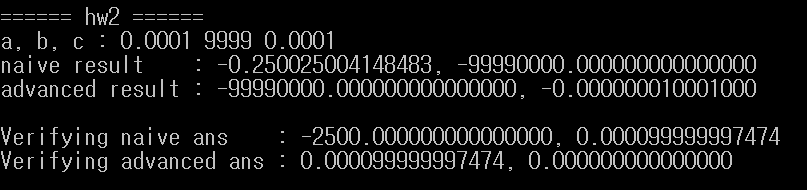
다음은 과제 1번의 수행 결과이다.

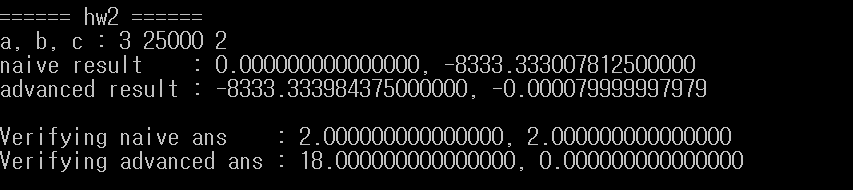


1) 2)

중학교 때 배운 근의 공식을 사용하여 이차방정식을 풀 때 b^2 >> |ac|라면 비슷한 숫자 간의 뺄셈으로 인해 정확도가 떨어진다. 다음과 같은 상황을 피하기 위해서 1)의 근의 공식을 유리화하여 다음과 같이 바꾼다. 이 식에 대입하게 되면 비슷한 숫자 사이의 뺄셈이 일어나지 않기 때문에 정확한 근을 구해낼 수 있다. 과제에서는 타입을 float로 정해두어 표현할 수 있는 유효숫자가 많지 않았다. 그래서 유리화해서 구한 새로운 근도 서로 다른 실근 두개를 정확히 구하지 못하고, 하나만 구할 수 있었다. Verifying advanced ans 부분을 확인해보면 항상 두 실근 중 한 개는 원래의 식에 대입했을 때 0의 값을 올바르게 출력한다. 다음은 과제 2의 수행 결과이다.







1. 1) 나눗셈 연산을 시프트 연산으로 바꾸기

나눗셈 연산은 다른 사칙연산에 비해 시간이 많이 소요되므로 최대한 사용을 안 하는 것이 좋다. 따라서 나눗셈 대신 시프트 연산을 사용하면 코드 최적화를 할 수 있다. 다음은 수행 시간을 비교한 결과이다. (original : 나눗셈, optimized : 시프트)



1. 비트 연산을 활용하여 메모리 낭비 줄이기(비트 하나당 하나의 정보)

저장하고자 하는 정보가 1bit짜리라면 아무리 메모리를 적게 차지하는 char, bool 형태로 변수를 선언해도 8bit가 필요하다. 이 경우 정보 하나당 7bit가 낭비되는 것인데 만약 정보의 개수가 많아진다면 낭비도 심해질 것이다. 따라서 이를 방지하기 위해 비트 연산을 이용하여 int의 비트 하나당 저장하고자 하는 정보를 담아 코드 최적화를 할 수 있다. 다음은 사용되는 메모리 크기를 비교한 결과이다.(original : int arr[16], optimized : int tmp)

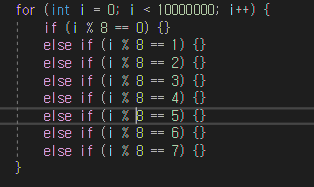
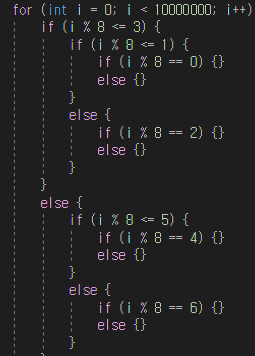


1. 홀수와 짝수의 구분을 비트로 하기

보통 홀수와 짝수를 구분하기 위해서는 if(num % 2 == 1)과 같이 사용하는 경우가 많다. 그러나 이 경우 1)에서 언급했던 것처럼 나눗셈을 사용하게 되어 여러 번 반복할수록 코드 소요시간이 길어진다. 이 부분을 해결하기 위해서 비트 연산을 사용한다. 예를 들어 if(num&1 == 1)이라면 num은 홀수인 것으로 %연산을 사용한 것과 같은 효과를 갖지만, 연산의 속도는 더 빠르다. 다음은 홀수와 짝수의 구분을 비트연산을 이용한 결과이다. (Original : ‘%’, optimized : ‘&’)



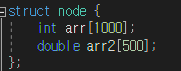
1. If문 쪼개기(binary breakdown)

A . B 

A과 같은 경우를 생각해보자. 최악의 경우에는 if문을 한번 도는데 8번의 비교가 필요하다. 만약 반복의 횟수가 커지거나, 비교 횟수가 더 커진다면 시간적인 측면에서 상당히 비효율적이다. 이를 해결하기 위해서 if 문을 쪼갤 수 있다. B와 같이 코드를 작성하면 최악의 경우에도 3번의 비교만이 필요하다. 다음은 코드 수행 결과이다.(Original : A의 경우, optimized : B의 경우)



1. 파라미터 전달 시 포인터를 사용해서 메모리 낭비 줄이기



다음과 같은 구조체(node)가 있다고 하자. 이 구조체를 매개변수로 하는 함수가 있다고 했을 때 이를 그대로 넘겨준다면 구조체의 모든 변수(arr, arr2)가 복사되어야 한다. 이는 시간적인 측면에서 뿐만 아니라 공간적인 측면에서도 상당한 낭비임을 알 수 있다. 구조체 내의 변수가 차지하는 공간이 커질수록, 또는 매개변수로 전달하는 노드가 배열을 이루는 경우에는 이와 같은 상황은 더욱 심해질 것이다. 그러나 매개변수로 구조체 자체가 아닌 해당 구조체의 포인터를 넘겨준다면 포인터 크기 만큼의 공간을 차지하는 것이 전부이다. 다음은 메모리 사용 측면에 대한 코드 수행 결과이다. (original : 구조체 전체를 넘겨줌, optimized : 포인터를 넘겨줌)

