프로젝트 #3 (개인)

컴퓨터학부 암호학

2023년 10월 2일

문제

결정적(deterministic) 밀러라빈(Miller-Rabin) 알고리즘을 사용하여 길이가 최대 64 비트인 소수를 찾는 프로그램을 구현한다. 베이스(base) 값 a를 무작위로 선택하는 확률적 밀러라빈 알고리즘과는 달리 결정적 밀러라빈 알고리즘은 매우 작은 집합의 정해진 베이스 값만 검증한다. 그 이유는 $n < 2^{64}$이면 베이스 값 a를 집합 $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$에서만 검증하면 충분하다는 것이 밝혀졌기 때문이다.

함수 구현

학생들이 구현할 함수의 프로토타입은 아래에 열거되어 있다. 각 함수에 대한 요구사항은 다음과 같다.

- `int miller_rabin(uint64_t n)` – 64비트 음이 아닌 정수 `n`이 소수이면 `PRIME`을, 그렇지 않으면 `COMPOSITE`을 넘겨준다. `n`이 2^{64} 보다 작으므로 결정적 밀러라빈 알고리즘을 사용한다.
- `uint64_t mod_add(uint64_t a, uint64_t b, uint64_t m)` – $a + b \bmod m$을 계산하여 넘겨준다. a와 b가 각각 m보다 작다는 가정하에서 a와 b의 합이 m보다 크거나 같으면 결과에서 m을 빼줘야 한다. 이 경우 실제 계산은 $a - (m - b)$로 하는 것이 오버플로를 피할 수 있는 좋은 방법이다. 또한 $a + b >= m$을 검사하는 과정에서 오버플로가 발생할 수 있으므로 b를 오른쪽으로 넘겨 $a >= m - b$를 검사하는 것이 오버플로를 피할 수 있는 현명한 방법이다.
- `uint64_t mod_sub(uint64_t a, uint64_t b, uint64_t m)` – $a - b \bmod m$을 계산하여 넘겨준다. a와 b가 각각 m보다 작다는 가정하에서 a가 b보다 작으면 결과가 음이 되므로 m을 더해준다. 즉, $a + (m - b)$으로 계산한다. 그렇지 않으면 원래대로 $a - b$로 계산한다.
- `uint64_t mod_mul(uint64_t a, uint64_t b, uint64_t m)` – $ab \bmod m$을 계산하여 넘겨준다. 오버플로가 발생할 수 있기 때문에 프로그래밍 언어가 제공하는 곱셈으로는 계산이 올바르지 않을 수 있다. 앞에서 정의한 `mod_add()`가 오버플로를 고려했다는 점과 곱셈을 덧셈을 사용하여 빠르게 계산할 수 있는 "double addition" 알고리즘을 사용하면 문제를 해결할 수 있다. 그 알고리즘은 다음과 같다.

```
r = 0;
while (b > 0) {
    if (b & 1)
        r = mod_add(r, a, m);
    b = b >> 1;
    a = mod_add(a, a, m);
}
return r;
```

- `uint64_t mod_pow(uint64_t a, uint64_t b, uint64_t m)` – $a^b \bmod m$을 계산하여 넘겨준다. 오버플로가 발생할 수 있기 때문에 이 역시 프로그래밍 언어가 제공하는 지수함수로는 계산이 올바르지 않을 수 있다. 앞에서 정의한 `mod_mul()`이 오버플로를 고려했다는 점과 지수연산을 곱셈을 사용하여 빠르게 계산할 수 있는 "square multiplication" 알고리즘을 사용하면 문제를 해결할 수 있다. 그 알고리즘은 다음과 같다.

```
r = 1;
while (b > 0) {
    if (b & 1)
        r = mod_mul(r, a, m);
    b = b >> 1;
    a = mod_mul(a, a, m);
}
return r;
```

OpenMP 설치

OpenMP (Open Multi-Processing)는 다중코어 환경에서 병렬프로그래밍을 지원하는 API이다. 이 과제에서는 소수를 빨리 찾기 위해 OpenMP 라이브러리를 사용한다. 학생들이 과제를 수행하는데 OpenMP에 대한 지식이 꼭 필요한 것은 아니지만 검증 프로그램을 돌리기 위해서는 OpenMP를 설치해야 한다.

- **Linux** 환경에서는 설치하지 않는다. 컴파일러에 내장되어 있다.
- **macOS** 환경에서는 먼저 Homebrew를 설치해야 한다. 이미 Homebrew가 설치되어 있다면 이 부분은 건너뛴다. Homebrew를 설치하려면 터미널을 열고 다음 명령어를 실행한다.

  ```
  % /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
    install/master/install.sh)"
  ```

  Homebrew 설치가 완료되면 다음 명령어를 실행하여 OpenMP 라이브러리를 설치한다.

  ```
  % brew install libomp
  ```

  애플 실리콘이 탑재된 맥에서는 `brew` 명령어를 찾지 못하는 오류가 발생할 수 있다. `brew` 명령어가 `/usr/local/bin` 밑에 있지 않고 `/opt/homebrew/bin` 밑에 있기 때문이다. `/opt/homebrew/bin`을 시스템 기본 실행 경로에 포함시켜야 한다. 사용자 홈디렉토리로 이동하여 문자편집기로 `.zshrc` 파일을 열거나 새로 생성한다. 아래처럼 `/opt/homebrew/bin` 경로를 추가하고 저장한 후 터미널을 다시 시작한다.

  ```
  export PATH=/opt/homebrew/bin:$PATH
  ```

  설치된 OpenMP의 헤더 파일과 라이브러리 파일 경로도 `gcc` 컴파일러 환경변수에 추가해야 한다. 먼저 헤더 파일과 라이브러리 파일이 어디에 설치되었는지 확인한다. 설치 과정에서 출력되는 메시지에 경로 정보가 있을 수 있으므로 잘 관찰한다. 예를 들어, OpenMP 헤더 파일과 라이브러리 파일이 `/opt/homebrew/include`와 `/opt/homebrew/lib` 디렉토리 밑에 설치되었다고 가정하자. 이 경우 앞에서와 마찬가지로 사용자 홈디렉토리로 이동하여 문자편집기로 `.zshrc` 파일을 열고, 아래처럼 설치된 경로를 추가하고 저장한 후 터미널을 다시 시작한다.

  ```
  export C_INCLUDE_PATH=/opt/homebrew/include:$C_INCLUDE_PATH
  export LIBRARY_PATH=/opt/homebrew/lib:$LIBRARY_PATH
  ```

골격 파일

구현이 필요한 골격파일 `miller_rabin.skeleton.c`와 함께 프로그램을 검증할 수 있는 `test.c`, 헤더파일 `miller_rabin.h`, 그리고 `Makefile`을 제공한다. 이 가운데 `test.c`를 제외한 나머지 파일은 용도에 맞게 자유롭게 수정할 수 있다.

제출물

과제에서 요구하는 함수가 잘 설계되고 구현되었다는 것을 보여주는 자료를 보고서 형식으로 작성한 후 PDF로 변환하여 `이름_학번_PROJ3.pdf`로 제출한다. 여기에는 다음과 같은 것이 반드시 포함되어야 한다.

- 본인이 작성한 함수에 대한 설명
- 컴파일 과정을 보여주는 화면 캡처
- 실행 결과물의 주요 장면과 그에 대한 설명, 소감, 문제점
- 프로그램 소스파일 (`miller_rabin.c`, `miller_rabin.h`) 별도 제출
- 프로그램 실행 결과 (`miller_rabin.txt`) 별도 제출

평가

- Correctness 50%: 프로그램이 올바르게 동작하는 지를 보는 것입니다. 여기에는 컴파일 과정은 물론, 과제가 요구하는 기능이 문제없이 잘 작동한다는 것을 보여주어야 합니다.
- Presentation 50%: 자신의 생각과 작성한 프로그램을 다른 사람이 쉽게 이해할 수 있도록 프로그램 내에 적절한 주석을 다는 행위와 같이 자신의 결과를 잘 표현하는 것입니다. 뿐만 아니라, 프로그램의 가독성, 효율성, 확장성, 일관성, 모듈화 등도 여기에 해당합니다. 이 부분은 상당히 주관적이지만 그러면서도 중요한 부분입니다. 컴퓨터과학에서 중요하게 생각하는 best coding practices를 참조하기 바랍니다.

HK