<2021 Spring Algorithm>

HW1 – Randomized and deterministic selection

자유전공학부 2012-13311  
안 효 지

**1. 구현방법**

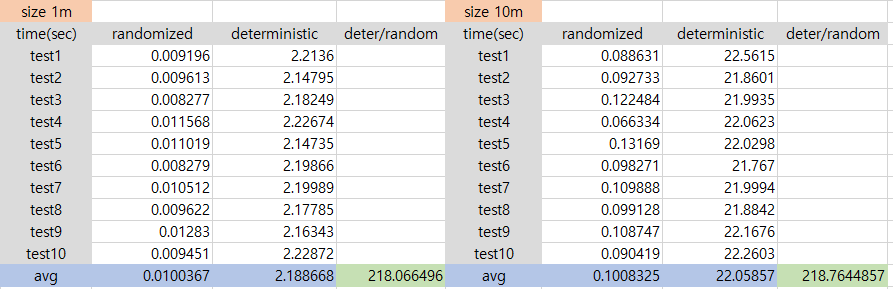
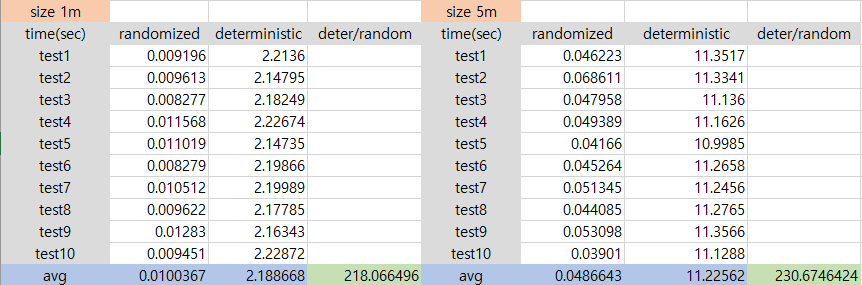
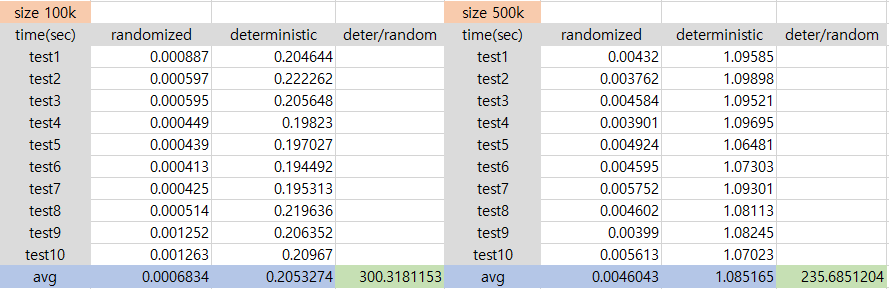
쉽게 배우는 알고리즘 책을 참고하여 cpp로 구현하였다. **①Randomized selection**은 quick sort에서 사용했던 partition을 이용하였는데, randomized라는 이름에 더욱 걸맞도록 srand()를 이용해 pivot을 random하게 정하였다.

**②** **Deterministic selection**을 구현하는 과정에서, 원소가 5개 이하인 경우에는 다른 코드를 더 구현하지 않고 ①Randomized selection을 사용해 중간값을 구하도록 하였다. 또한, 처음에는 어짜피 median of medians 값만 구하면 되므로 원소가 5개인 그룹들의 medians를 전체 arr의 앞 인덱스부터 차례차례 채우려고 했었다. 하지만 재귀를 거듭할수록 partition이 망가지는 것을 알게 되어 medians를 담는 vector를 만들어 거기에 push를 해주었다.

**③** **Check()** 기능은 내가 짠 알고리즘이 내놓은 output이 정말 i번째로 작은 원소가 맞는지를 확인하면 된다. Array를 for문으로 한 번 반복하면서 output보다 ‘작은 원소의 개수(smaller)’들과 output과 ‘같은 값의 원소의 개수(equal)’를 따로 센 후, smaller < ith <= smaller+equal 인지 확인하면 된다.  
 가령, -1 0 0 0 1 의 5개 원소의 배열에서 2번째~4번째로 작은 원소는 전부 0이며, 5번째로 작은 원소는 1이다. 우리는 0이 두 번째로 작은 원소가 맞느냐는 질문에도 yes, 세 번째, 네 번째로 작은 원소냐는 물음에도 yes 라고 답을 해야한다. 따라서, 0보다 작은 원소가 배열에 1개 있고, 0과 동일한 원소는 3개가 있으므로, 0은 1 < ith <= 1+ 3(작은 원소 개수 + 동일 원소 개수)의 범위를 충족하여 2, 3, 4번째로 작은 숫자들이 된다.  
 이 check 알고리즘은 크기가 n인 배열에서 for문을 한 번만 돌기 때문에 \Theta(n) 시간이 걸린다.

**2. 시간측정**

시간은 size 100k, 500k, 1m, 5m 10m일 때, 총 10번씩 실행하여 평균을 내었다. 결과는 아래와 같다.



randomized selection은 worst case인 경우에는 \Theta(n^2)의 시간복잡도를 보이나, 이번 실험에서는 선형의 시간복잡도를 보였다. R^2의 값이 1에 근사한 것을 통해 알 수 있다.

deterministic selection은 worst case인 경우에도 \Theta(n)의 시간복잡도를 보장하는데, 이를 위해 \Theta(n)의 오버헤드를 사용한 다음 입력의 크기가 in/10인 문제를 재귀적으로 호출한다. 균형을 맞추는 오버헤드가 너무 커져버리면 linear time complexity를 보장할 수 없으나, 본 실험에서는 선형 시간복잡도를 보임을 R^2값을 통해 알 수 있다.

1차항 계수의 비를 계산하면 217정도로 아주 크게 나오는데, 이는 deterministic selection의 오버헤드 때문이다. 이 오버헤드의 크기에 따라 deterministic의 사용 여부가 정해진다.

**3. Example running**

Compile & run 방법은 README.md에 작성해두었다. 하지만 기존 스켈레톤코드와 Makefile을 그대로 사용하였기 때문에 특별한 것은 없다. 똑같이 make, make run을 통해 실행이 가능하다.

아래는 실행 예시이다. $ make run을 하면 아래와 같이 나온다. Size500k, 1m은 시간이 약간 걸린다. 나머지 size5m, 10m은 시간이 오래 걸려 Makefile에는 넣어놓지 않았다.

