BufferedReader는 한 줄을 통째로 입력받는 방법으로 주로 쓰인다.

ㄴ readline() 메서드는 값을 읽어올 때, String 값으로 개행 문자(엔터값)을 포함해 한줄을 전부 읽어오는 방식

ㄴ read() 메서드는 값을 읽어올때, 한 문자만 int값으로 변형하여 읽어오는 방식이다.(ASCII형식으로 읽어온다)

(공백 아스키 : 32, 숫자 '1' 아스키 : 49, '\n' 아스키 : 10, A 아스키 : 65 )

간단한 최적화

* 입력처리에는 Scanner보다 BufferedReader가 효율적이다.

(Scanner : space 모두 키값으로 인식, BufferedReader: enter값만 인식, String값으로 입력값 고정)

* 한 줄 입력이 여러 번 들어오는 경우, split보다 StringTokenizer를 사용하여 파싱하는 것이 좋다
* 여러 번 출력해야 하는 경우에는 StringBuilder를 사용해 한 번에 출력하는 것이 좋다.
* Array보다는 ArrayList다 효율적이다.
* ArrayList의 정렬에는 Collections.sort() (TIM 정렬(삽입+합병))을 사용하자

Collections.sort와 Arrays.sort()는 다른 것!

* 배열을 초기화 할 때는 java.util.Arrays의 Arrays.fill(배열, 값)을 사용하자
* 연속으로 있는 숫자를 입력받기 위해서는 1번 대신에 2번을 사용하자

1011010101

1. String[] inputs = br.readLine().split(“”);
2. br.read() – ‘0’;

* 반복문의 출력에는 System.out.println 대신에 StringBuilder를 사용하자

System...println은 내부코드에 synchronized가 있어 동기화 시, 대기시간이 존재하게 된다.(매 번 객체생성)

StringBuilder는 String 값의 확장을 도와주는 기능을 한다.

즉, 반복문에서 매번 출력하는 코드나, String 변수 간의 덧셈으로 최종출력값을 얻는 방법보다,

StringBuilder를 통한 최종값 도출 및 한번의 출력을 통해 많은 오버헤드를 줄일 수 있다.

자바의 최적화에 대하여

* 코드 최적화는 무엇일까?

: 불필요한 수행 작업을 줄이고 자원을 적게 쓰도록 시스템을 수정하는 방법을 의미한다.

* 필요없는 자원의 소모를 줄이자

: 만약 100개의 정수를 입력 받아 배열로 만든다고 가정하자, 대게 들어오는 정수의 값을 특정지을 수 없다면 long형을 사용할 것이다. but, int범위 내로 받을 수 있는 것을 long형을 사용했다면 굉장한 자원낭비 일 수 있다.(long형 대신 int형 쓸 경우 8byte에서 4byte로 줄어드므로 반 이상 용량을 아낄 수 있다.

* 재귀함수와 반복문

: 두 방식 모두 수행 횟수 자체는 차이가 없지만, 메모리 구조에 대한 이해가 있다면 바로 알 수 있듯이, 스함수 호출 자체 비용이 비싸다는 것을 알 것이다. 스택메모리에 함수의 호출 반환 위치, 변수 등이 ‘매 번 호출할 때마다 쌓인다’

* 수행 작업(알고리즘) 최적화

: 1~N까지의 합을 구하고자 할 때, 보통의 경우는 반복문을 활용하여 작성한다.

but, 조금의 수학지식이 있다면 N\*(N+1)/2 와 같이 한 번의 연산만을 이용하여 끝낼 수 있다.

* 코드의 가독성을 해칠 정도의 사소한 코드 최적화는 필요치 않다

: 산술 연산(i+1) 보다 비트연산(i++)이 더 빠르다는 것은 일반적으로 사실이다.(하지만 컴파일러와 캐시 등과 같은 문제로 항상 빠르다고 볼 수는 없다)

* (불필요한) 임시 변수 생략

: 예를 들어 a와 b의 합을 계산하려 할 때, tmp변수를 생성할 것이다. 이를 생략하여 tmp=a+b; return tmp가 아니라 return a+b; 로 만들면 메모리에 훨씬 부담이 적다.

* 자바 this 키워드와 연쇄 방법

: 메소드 호출 시, 메소드 앞에 어떤 클래스의 메소드인지를 파악할 수 있게 하면 최적화된다.

ex) Resource myAppResources = this.getResources();

InputStream is = myAppResources.openRawResource(R.drawable.icon);

// 위와 같이 this를 활용하자

* 패키지 import

: 필요없는 패키지까지 모두 import하는 것은 시간도 공간도 모두 소요시킨다.

상세하게 import하자

코드 최적화를 위한 변환 기법

1. Common subexpression elimination : 공통된 코드를 중복 실행하지 않고 한 번 실행한 값을 활용해 최적화하는 방법이다. ex) tmp = a\*b; a= tmp + g; b=tmp+d;
2. Loop Unrolling : 루프 코드는 대체로 실행에 많은 시간이 소요된다. 특히 다중루프의 경우 더 많은 시간이 소요될 수 밖에 없다. 따라서 지정한 범위에 도달했는지 검사하기 위해 값을 비교하고 증가시키는 등의 작업이 진행되는데, 코드에서 이런 작업 과정을 줄일 수 있다면 실행 속도를 보다 빠르게 할 수 있다.

ex)for(i=0;i<3;i++)ab[i]=i; ---🡪 ab[0]=0; ab[1]=1; ab[2]=2;

// 실제로 해당 배열에 들어가는 숫자가 적고, 단순하다면 조건에 맞는 지 테스트 하는 과정이 생략되어 빠르게 동작한다.

1. Function inlining : 함수를 호출하면 복귀할 위치를 스택에 저장하고 해당 함수로 점프한다. 만약 해당 함수에서 일반적으로 사용하는 레지스터를 스택에 보관하고 함수를 실행을 마치면, 레지스터의 값을 복원하고 보관해둔 주소로 점프하는 과정을 거친다. 이런 시간을 줄이기 위해서는 작은 코드인 함수를 직접 삽입해 사용하는 방법이 있다.

ex) int max(); -> max(); //호출방법

ex) max()메서드의 내용물을 그대로 main()에 넣어버림

ㄴ> 이를 통해 단순한 코드일수록 함수로 만들면 오히려 역효과가 난다는 것을 알 수 있음

1. Code motion : 코드를 이동해 보다 효율적인 실행 코드를 생성한다.

ex) for(i=0;i<n;i++){ab[i]=7\*i+x\*x;} 🡺 t = x\*x; for(i=0; i<n; i++){ab[i] = 7\*I + t;

// 위의 코드를 보면 4번의 최적화를 했지만, 그로 인해 1번 최적화를 거부한 것이 된다.

// 공간을 쓸 것인지, 실행 시간을 쓸 것인지를 정해서 최적화하는 것이 좋다(일반적으로는 시간을 단축)

1. Type-based alias analysis : 데이터 타입이 다르다면 별도의 선언이 없는 경우, 서로 다른 위치에 데이터가 존재한다.
2. static 또는 global 변수가 사용하는 함수에서 보다 효율적인 처리를 위해서는 해당 변수들이 위치를 변경한다.
3. Instruction scheduling : 여러 개의 파이프라인을 사용하는 경우, 앞의 파이프라인에서 결과가 나오지 않으면 뒤의 파이프라인이 기다리는 상태가 된다. 이를 Stall이라고 말하며, Instruction scheduling은 이런 Stall 현상을 최소화해 명령어를 재배열하는 방법을 말한다.