컴퓨터 구조 프로젝트

전기전자공학부 2016170994 김다민

1. **3x3 Inverse matrix**

Data 영역에 미리 데이터를 저장하고 역행렬을 구하는 방식으로 진행했다..

입력 예시 행렬:

1 1 0

2 1 -1

0 1 2

.data

a11: .word 1

a12: .word 1

a13: .word 0

a21: .word 2

a22: .word 1

a23: .word -1

a31: .word 0

a32: .word 1

a33: .word 2

no: .asciiz "There is no inverse matrix"

space: .asciiz " "

new\_line: .asciiz "\n"

그 밖에도 역행렬이 없을 시 출력할 문구를 no에 저장하고 스페이스바와 줄바꿈 문자열을 각각 space와 new\_line에 저장하였다.

메인 코드이다.

역행렬을 계산하기 데이터영역에 있는 입력 행렬의 각 성분 a11~a32을 $s0~$s7에 저장하고 a33을 $t8 레지스터에 저장하였다.

.text

main:

$t9에 determinat를 저장하기위해 초기화한다.

li $t9 0

#data 에서 입력값을 받아옴

lw $s0, a11

lw $s1, a12

lw $s2, a13

lw $s3, a21

lw $s4, a22

lw $s5, a23

lw $s6, a31

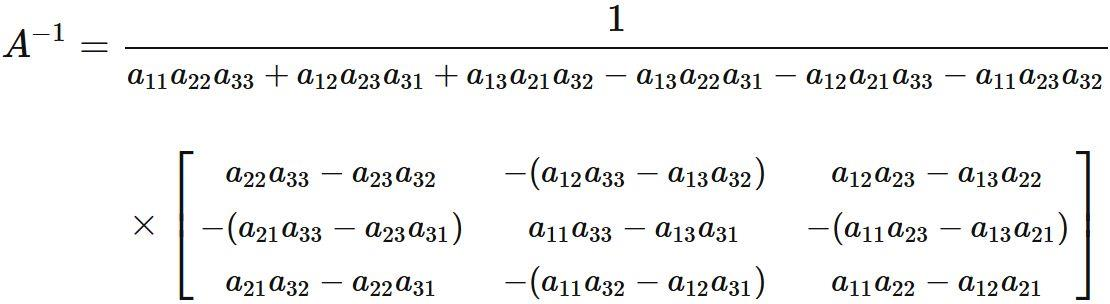
lw $s7, a32

lw $t8, a33

아래는

역행렬을 구하는 공식이다. 먼저 determinant 에 해당하는 a11a22a33+ a12a23a31 + a13a21a32 -a13a22a31 - a12a21a33 - a11a23a32 를 구하고 그 값이 0이면 역행렬이 존재하지 않으므로 “there is no inverse matrix”라는 문자열을 출력한다.

0이 아닐 시 위의값의 determinant의 역수인 값을 $t9 에 저장한다.



#3x3 determiant 계산과정

mul $t0, $s0, $s4

mul $t0, $t0, $t8

add $t9, $t9, $t0

mul $t0, $s1, $s5

mul $t0, $t0, $s6

add $t9, $t9, $t0

mul $t0, $s2, $s3

mul $t0, $t0, $s7

add $t9, $t9, $t0

mul $t0, $s2, $s4

mul $t0, $t0, $s6

sub $t9, $t9, $t0

mul $t0, $s1, $s3

mul $t0, $t0, $t8

sub $t9, $t9, $t0

mul $t0, $s0, $s5

mul $t0, $t0, $s7

sub $t9, $t9, $t0

#determiant의 분모가 0이면 역행렬이 존재하지 않으므로 따로 처리

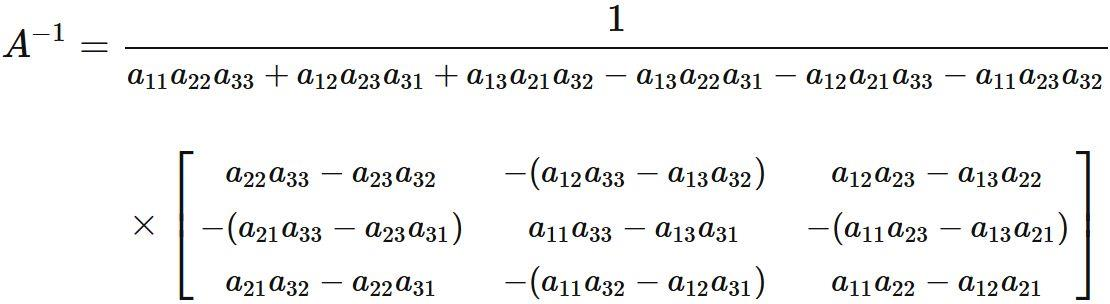
beq $t9, 0, no\_inverse

#determiant 를 $t9에 저장

addi $t0 $zero, 1

div $t9, $t0, $t9

**추후에 “there is no inverse”를 출력하는 no\_inverse코드에 대한 설명이 나온다.**



이제 위 공식에서 determinant의 역수에 해당하는 값이 $t9에 저장되어 있다.

다음으로 각성분을 구할 차례이다.

위공식을 보면 공통점이 나오는데 a행렬의 두성분을 곱한것 끼리의 뺄셈이라는 점이다.

따라서 성분두개의 곱을 각각 $t0, $t1에 저장하고, 서로의 뺄셈과 determinant의 역수 곱을 하는 부분은 중복되므로 나중에 처리해준다.

예를 들면 첫번째 성분은 a22a33 – a23a32이므로 $t0에 a22a23을 저장하고 $t1에 a23a32를 저장한 뒤 다음과정으로 넘긴다. 두번째 성분은 –(a12a33-a13a32) = a13a32 -a12a33 이므로 $t0에 a12a33 $t1에 a13a32를 저장한다.

서로의 뺄셈연산과 detertminant와 곱해지는 과정은 중복되므로 print\_num:에서 처리해주는게 코드가 효율적이고 짧아진다. 따라서 $t0 와 $t1에 값을 저장한후에 jal를 이용해 서로뺄셈을 실행하고 determinant의 역수와 곱을 해서 역행렬성분을 완성하고 바로 print를 시행한다.

$t6 레지스터는 3번째마다 스페이스가 아닌 줄바꿈을 실행하기위해 카운트를 해주는 변수로 이용 된다.

li $t6, 0

#역행렬 B의 성분을 구하고 하나씩 구하고 출력

#b11 구하고 출력하기

mul $t0, $s4, $t8

mul $t1, $s5, $s7

jal print\_num

#b12

mul $t0, $s2, $s7

mul $t1, $s1, $t8

jal print\_num

#b13

mul $t0, $s1, $s5

mul $t1, $s2, $s4

jal print\_num

#b21

mul $t0, $s5, $s6

mul $t1, $s3, $t8

jal print\_num

#b22

mul $t0, $s0, $t8

mul $t1, $s2, $s6

jal print\_num

#b23

mul $t0, $s2, $s3

mul $t1, $s0, $s5

jal print\_num

#b31

mul $t0, $s3, $s7

mul $t1, $s4, $s6

jal print\_num

#b32

mul $t0, $s1, $s6

mul $t1, $s0, $s7

jal print\_num

#b33

mul $t0, $s0, $s4

mul $t1, $s1, $s3

jal print\_num

j finish

. 마지막 b33을 출력한 뒤에는 프로그램 종료를 위해 finish로 점프시킨다.

$t0, $t1에 저장된 성분을 서로 빼주고 determinat의 역수를 곱해 역행렬의 성분을 완성시킨다. 완성시킨 성분은 출력을 위해 바로 $a0에 저장한다. 그 다음 integer출력을 위해 $v0에 1을 넣어주고

Syscall을 하여 역행렬의 각성분을 출력한다.

print\_num:

sub $t0, $t0, $t1

mul $a0, $t0, $t9

li $v0, 1

syscall

숫자를 출력한후에는 “ ”나 “\n”을 출력하기 위한 코드가 실행된다. 문자열 출력을 위해 $v0 에 4를 넣어준다. 그후에 3번째마다 “\n”을 3번째가 아닐 경우에는 “ ”를 출력시킨다. 각각 “ ”나 “\n” 을 출력한 뒤에는 jal과정에서 저장된 $ra의 주소로 다시 점프하여 다음 성분을 출력하는 코드를 실행한다.

# ' '나 '\n'를 출력

# jal로 a0에 3번째마다 '\n' 아니면 ' '의 주소를 담음

li $v0, 4

#space\_newline, new\_line:' '나 '\n' 판단

addi $t6, $t6, 1

beq $t6, 3, newline

la $a0, space

syscall

jr $ra

newline:

la $a0, new\_line

li $t6, 0

syscall

jr $ra

determinant가 0일경우 역행렬이 없으므로 “there is no inverse matrix”라는 문구를 출력하고 이어서 바로 프로그램종료를 위한 코드가 실행된다.

#determiant의 분모가 0일때 역행렬 없다고 출력, 이어서 종료(finish: 실행)

no\_inverse:

li $v0, 4

la $a0, no

syscall

프로그램 종료 코드는 모든 성분이 출력된 후에도 실행될 수 있도록 위에서 모든성분이 출력된후 j finish로 이어받아 종료코드를 실행시킨다. 종료하는 코드는 $v0에 10을 넣고 syscall을 하면 된다.

#출력 후 프로그램 종료

finish:

li $v0, 10

syscall

**출력예시**

Case1 역행렬이 있는 경우:

입력행렬:

1 1 0

2 1 -1

0 1 2

출력결과

스크린샷이(가) 표시된 사진

자동 생성된 설명

직접 계산을 통해 맞게 출력되었음을 확인해보았다.

Case2 역행렬이 없는 경우 :

Data영역을 다음과 같이 바꾸어 진행하였다.

.data

a11: .word 1

a12: .word 1

a13: .word 1

a21: .word 1

a22: .word 1

a23: .word 1

a31: .word 1

a32: .word 1

a33: .word 1

no: .asciiz "There is no inverse matrix"

space: .asciiz " "

new\_line: .asciiz "\n"

입력행렬 :

1 1 1

1 1 1

1 1 1

출력결과:

스크린샷, 조류이(가) 표시된 사진

자동 생성된 설명

Determinant가 0인경우에 역행렬이 없다고 제대로 출력되었다.

1. **Heap sort**

보통 heapsort로 내림차순 하여 배열에 저장하기 위해선 minheap을 이용한다. 그러나 mips에서는 데이터 저장을 위해 stack을 이용하므로 maxheap으로 정렬하여 큰 값이 top에 오게 만든 후 그 값을 pop을 하고 출력하면 출력을 내림차순으로 할 수 있다. 따라서 maxheap을 이용해 heap sort를 진행하였다. 아래는 과정의 예시이다.

Ex)

1. Max heap 구성

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 20 | 15 | 16 | 6 | 7 | 8 | 9 | 3(top) |

1. Heap sort.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 6 | 7 | 8 | 9 | 15 | 16 | 20(top) |

1. **pop & print**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 6 | 7 | 8 | 9 | 15 | 16(top) |

**출력 :20**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 6 | 7 | 8 | 9 | 15(top) |

**출력 :20 16**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 6 | 7 | 8 | 9(top) |

**출력: 20 16 15**

* ● ●

|  |
| --- |
| 3 |

**출력: 20 16 15 9 8 7 6**

전부 pop

**최종 출력: 20 16 15 9 8 7 6 3**

**이런 식으로 내림차순으로 출력하였다.**

**문제에 의하면 입력으로 원소를 받아 마지막 노드에 추가 후 즉시 heapsort를 진행한다고 하였다. Buildheap의 경우 연산수가 insertion에 비해 더 많으므로 이 경우 매 입력마다 buildheap을 시행하지 말고 insertion 알고리즘을 이용해 한번의 insertion으로 maxheap을 유지한다면 다음 시행에 대해서도 전에 만들어 놓았던 maxheap을 계속 이용할 수 있어서 연산횟수가 줄어 더 효율적이다. 따라서 buildheap과정을 사용하지 않고 insertion알고리즘을 이용해 maxheap을 구성하였다.(연속시행에서 이전시행때 구성된 maxheap에 insertion알고리즘 실행) 다시 말하면 매 시행시 말단노드에 추가된 값을 insertion 알고리즘에 따라 진행하면 아래서 위로 heapify를 하면 maxheap으로 유지되므로 insertion만 제대로 이루어지면 따로 buildheap 과정을 하여 maxheap을 새로 구성할 필요가 없다. 그 이후 maxheap을 정렬하기 위해선 위에서 아래로 heapify를 재귀적으로 이용하여 정렬을 한다.**

Code

Data : heap을 저장할 배열 공간을 만들고, 출력에 필요한 문자열들을 저장해 놓는다.

.data

space: .asciiz " "

inform: .asciiz "-9999 입력시 종료"

input: .asciiz "\n\n입력:"

output: .asciiz "출력:"

#heap 배열 할당

heap: .align 2 #word를 할당 (int형 자료)

.space 4000 #heap 배열 정수 1000개까지로 할당

**프로그램 종료 조건을 넣기위해 -9999입력시 프로그램을 종료하도록 하였다. 따라서 맨처음에 안내문구인 -9999입력시 종료를 넣었다.**

.text

main:

# "-9999 입력시 종료"를 출력

li $v0, 4

la $a0, inform

syscall

**heap배열의 현재크기를 저장하기위해 변수를 하나 할당해 이용하였다.**

#heap의 현재주소를 저장하기 위해 변수 할당

la $s7, heap

#heap의 현재 크기를 저장하기위한 변수 할당

li $s0, 0

**heap배열의 insertion시 마지막 노드에 추가될 값을 입력 받는다.**

자세한내용은 주석으로 달아 놓았다.

#입력 값을 읽어 들이고 heap배열에 저장함

read:

#"\n\n입력:"을 출력함

li $v0, 4

la $a0, input

syscall

li $v0, 5

syscall

# -9999입력시 종료시킴

beq $v0, -9999, finish

# 읽어들인 값 heap배열에 저장

addi $s7, $s7, 4

addi $s0, $s0, 1

sw $v0, 0($s7)

**입력받은 값에 대해 insertion을 실행한다. 입력 받은 값을 마지막 노드에 추가하고 insertion 알고리즘을 적용하여 즉시 maxheap을 구성한다.**

#heap 의 max index와 current index 초기화

addi $t7, $s0, 0

addi $t9, $s0, 0

#새로운 입력이 들어오면 insert 알고리즘에의해 maxheap 구성하는 재귀적 과정(아래서 위로 heapify)

## heap insertion 에서 사용될 register

# $s0 max index(말단 노드 index)

# $t8 parent index

# $t7 current index

# $t6 parent value

# $t5 current value

**아래는 마지막 노드에 추가된 값을 insertion 알고리즘에 사용되는 재귀적 과정을 담은 코드이다. heap배열에 저장된 부모의 값과 자식의 값을 비교하기 위해 부모의 값과 주소, 자식의 값과 주소를 레지스터에 저장해 놓고 이용하였다. 위의 주석에 어떤 레지스터에 어떤 값을 저장해 놓았는지 적어 놓았다. 편의상 roof의 index를 1로 설정해 진행하였으며 따라서 자식과 부모의 교환이 일어날 경우 다음 시행에서 부모가 현재 값으로 바뀐다.**

**이때 현재 노드의 index가 n이면 부모의 index는 왼쪽 자식일 경우 n/2 혹은 오른쪽 자식 일 경우(n-1)/2이다 이때 n을 오른쪽으로 1bit shift를 오른쪽자식이라 하더라도 오버 플로우가 무시되게 때문에 하면 왼쪽 오른쪽에 상관없이 부모의 index값을 갖을 수 있게 된다.**

insert:

**insertion 종료조건은 자가자신이 roof일때 즉 index가 1일때 종료된다.**

**부모의 index는 자기자신을 오른쪽으로 1비트 shift하여 구한다.**

beq $t7, 1, load\_heap

srl $t8 $t7, 1

**부모의 값을 load하는과정**

sub $t1, $s0, $t8

mul $t1, $t1, 4

sub $t1, $s7, $t1

lw $t6, 0($t1)

**현재 값을 load하는과정**

sub $t2, $s0, $t7

mul $t2, $t2, 4

sub $t2, $s7, $t2

lw $t5, 0($t2)

**부모와 현재 값을 비교하여**

**현재 값이 더 작을 경우 재귀문 종료**

slt $t0, $t5, $t6

bne $t0, 0, load\_heap

**부모보다 현재값이 클 경우 교환이 일어남**

sw $t6, 0($t2)

sw $t5, 0($t1)

**교환 시 부모의 index가 다음시행의 현재 index가 된다.**

addi $t7, $t8, 0

**재귀반복을 위한 점프문**

j insert

**위에서 마지막노드에 추가한 값을 바로 insertion을 통해 maxheap으로 저장하였으므로 현재 heap 배열에는 maxheap이 저장되어 있다.**

##정렬을 위해 heap array에서 stack으로 데이터를 가져옴

**Heapsort를 시행하기위해 저장된 Max heap을 스택으로 불러오는 과정**

load\_heap:

# heap array[1]의 주소를 가져옴

li $t0, 0

mul $t1, $s0, 4

sub $t1, $s7, $t1

#heap array[1] 부터 끝까지 순차대로 stack에 값 저장

load\_val:

lw $t2, 0($t1)

addi $sp, $sp -4

sw $t2, 0($sp)

**현재heap의 크기만큼 load가 되면 전부 load된것이므로 load종료**

**그렇지 않으면 주소에 4를 더해(word의크기가 4비트이므로) 로드를 계속 진행**

beq $t0, $s0, sort\_initial

addi $t1, $t1, 4

addi $t0, 1

**반복을 위한 점프 구문**

j load\_val

**다음은 스택에 load된 Maxheap을 heapsort시키는 과정이다.**

**아래 주석은 과정에서 저장에 사용될 레지스터들의 목록이다.**

## heap sort 과정

##사용될 register 목록들

#s0 heap의 전체 크기

#t9 current max index

#t8 current index

#t7 current value

#t6 left value

#t5 right value

#t4 current address

#t3 left address

#t2 right address

**알고리즘은 말단노드와 roof의 값을 바꾼 후 heap의 크기가 1줄었다고 생각하고 주어진 모든 노드에서 roof부터 말단 노드까지 차례대로 위에서 아래로 heapify를 실행시켜 모든 노드에 대해 heapify시키면 Maxheap이 완성된다.**

**(이 문제에서는 insertion 알고리즘에 의해 buildheap과정이 불필요하므로 생략하였지만, 만일 정리가 안된 tree를 max heap으로 정렬해야 하는 경우에는 $s0에 저장된 heap전체크기를 절반으로 나눈 뒤 그 지점부터 roof까지 heapify해주면 된다. heapify알고리즘은 아래 구현되어 있다.)**

#stack에 가져온 heap데이터를 sort 하기위해 필요한 초기화 과정

**이 과정에서 부모와 말단 노드값을 바꾸고 heap의 크기를 1줄인다. 이과정에서의 종료조건은 현재 heap의 크기가 1이되는 순간이다.**

sort\_initial:

#heap의 크기가 1이 되면 heap sort 종료

beq $t9, 1 ,print\_heap

#첫번째 노드와 마지막 노드의 값을 바꾸는 과정

sub $t0, $s0, $t9

mul $t0, $t0, 4

add $t0, $sp, $t0

lw $t2 ,0($t0)

sub $t1, $s0, 1

mul $t1, $t1, 4

add $t1, $sp, $t1

lw $t3, 0($t1)

sw $t3, 0($t0)

sw $t2, 0($t1)

# heap의 크기를 1 줄인다.

sub $t9, $t9, 1

#현재 index $t8, roof에서 아래로 heapify하므로 1로 줌

li $t8, 1

**아래과정은 위에서 아래로 heapify하는 과정이다. roof에서 시작하여 아래로 heapify한다. 자식과 교환이 일어난 경우 재귀적으로 아래 노드에 대해 heapify를 실행한다. 교환이 일어나지 않는 경우에는 다시 위로 돌아가 roof와 말단노드의 값을 바꾸고 heap크기를 1줄이고 아래 과정을 다시 실행한다. 각 코드의 설명을 주석에 달아 놓았다.**

**연산수를 최소화 하기위해 자식의 값을 먼저 load하지 않고, 자식이 하나인경우, 둘인경우에 맞추어 필요에 맞추어 load하였다.**

sort\_recur:

#case1 자식이 없을 경우 다음 노드에서 실행 -> 자식 load 불팔요

#left child의 index $t1 = current index \*2 / $t1에 저장

mul $t1, $t8, 2

#left child index > current max index 보다 클경우 자식이 없음 다음 swap 진행

slt $t3, $t9, $t1

bne $t3, 0, sort\_initial

#case2 자식이 하나일 경우 -> 왼쪽자식에 대해 load필요

#current address를 $t4에 저장하고 /current value를 $t7에 load한다.

sub $t4, $s0, $t8

mul $t4, $t4, 4

add $t4, $sp, $t4

lw $t7, 0($t4)

#left child address를 $3에 저장하고 /left child value를 $t6에 load한다.

sub $t3, $s0, $t1

mul $t3, $t3, 4

add $t3, $sp, $t3

lw $t6, 0($t3)

beq $t1, $t9, one\_child #아래 one\_child:에서 값 비교 후 정렬 실행, 코드 효율성 증가와 jump를 줄이기 위해 아래에 배치함.

#case3 자식이 둘인 경우 -> 오른쪽 자식의 load필요

#right child의 index $t2 = left child의 index+1 $t2에 저장

addi $t2, $t1, 1

#right child address를 $2에 저장하고 /left child value를 $t5에 load한다.

sub $t2, $s0, $t2

mul $t2, $t2, 4

add $t2, $sp, $t2

lw $t5, 0($t2)

#t7 current value

#t6 left value

#t5 right value

#t4 current address

#t3 left address

#t2 right address

#자기자신과 left child 비교

slt $t0, $t7, $t6

bne $t0, 0, small

#자기 자신이 left child 보다 클때

slt $t0, $t7, $t5

bne $t0, 0, change\_right

j sort\_initial

#자기 자신이 left child보다 작을때

small:

slt $t0, $t6, $t5

bne $t0 , 0 ,change\_right

j change\_left

#case2의 비교,정렬 / 자식이 하나일 경우 부모가 자식보다 작으면 jump 없이 바로 change\_left 실행

one\_child:

slt $t0, $t7, $t6

beq $t0, 0, sort\_initial

#부모와 left child의 값을 바꾸고 current index를 left child의 index로 바꿈

change\_left:

sw $t7, 0($t3)

sw $t6, 0($t4)

mul $t8, $t8, 2

j sort\_recur

#부모와 right child의 값을 바꾸고 current index를 right child의 index로 바꿈

change\_right:

sw $t7, 0($t2)

sw $t5, 0($t4)

mul $t8, $t8, 2

addi $t8, $t8, 1

j sort\_recur

**아래과정은 heapsort가 모두 종료되고 stack에 저장된 값을 pop하면서 내림차순으로 출력하는 코드이다.**

##heap sort가 끝난 후 stack에 저장된 값을 pop하면서 내림차순으로 출력

print\_heap:

# "출력:"을 출력함

li $v0, 4

la $a0, output

syscall

#반복 counting에 필요한 변수 t0 할당

li $t0, 0

print\_num:

# 모든 출력이 끝난 후 다시 입력과정으로 이동하는 코드

beq $t0, $s0, read

addi $t0, $t0, 1

#숫자 출력 및 스택포인터 반환 (pop)

li $v0, 1

lw $a0, 0($sp)

syscall

addi $sp, $sp, 4

# 숫자사이 공백 출력

li $v0, 4

la $a0, space

syscall

#숫자 출력 반복하게 함

j print\_num

**위에서 -9999입력시 종료하도록 이곳으로 점프시킨다. 종료를 위해 $v0에 10을 넣고 syscall을 한다.**

#"-9999입력시 프로그램 종료"

finish:

li $v0, 10

syscall

**아래는 실제 실행결과이다.**

스크린샷이(가) 표시된 사진

자동 생성된 설명

문제에서 주어진 예시대로 시행해 보았다.

내림차순으로 잘 출력 되었으며, -9999입력시 종료도 잘 되는 것을 확인할 수 있었다.

그 밖에 0이나 음수, 중복되는 값을 넣어도 문제없이 실행되는 것을 확인할 수 있었다.

