**Assessed Exercise 2: Assembly Language Programming  
Computer Systems 2  
Semester 1, 2017/18**

***Stuart Reilly  
2258082R***

49 recursive

79 iter

50 println

The two implmentations solve the 0-1 Knapsack problem successfully. Since the time complexity of the recursive solution is , it doesn’t depend on the weight limit, only the number of elements. This causes the recursive solution to be more efficent at producing a solution for large weight limits, but, less efficent at solving solutions for large number of elements than the dynamic solution. With a time complexity of , the dynamic solution scales linearly with in N and W, but the recursive soltuion scales polynomially with N and doesn’t scale with W.

In order to test the solutions, a Rust implmentation of the recusrive solution was created to find the correct result for each data set. The following is the implmentation:

fn knapsack(n: usize, w:usize, weights: &[usize], values: &[usize], i: usize) -> usize {

if w == 0 || i >= n {

0

} else if weights[i] > w {

knapsack(n, w, weights, values, i + 1)

} else {

let take = values[i] + knapsack(n, w - weights[i], weights, values, i + 1);

let leave = knapsack(n, w, weights, values, i + 1);

::std::cmp::max(take, leave)

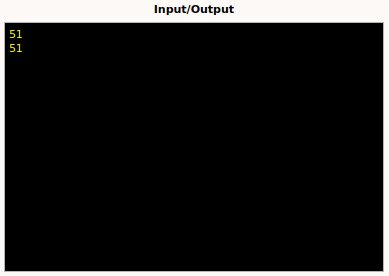
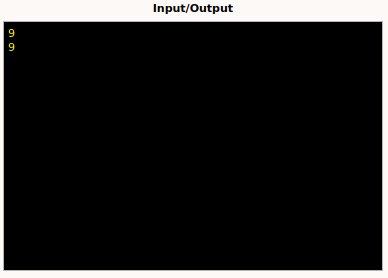
}

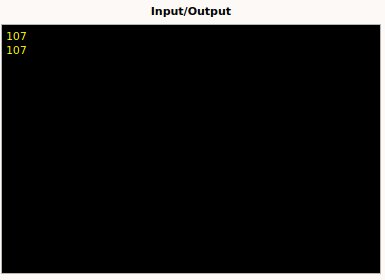
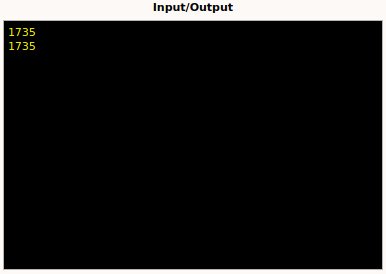
}

This implmentaion produced the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | W | Weights | Values | Result |
| 3 | 5 | 1,2,3 | 4,3,5 | 9 |
| 5 | 16 | 12,7,11,8,9 | 24,13,26,15,16 | 51 |
| 7 | 170 | 41,50,49,59,55,57,60 | 442,525,511,593,546,564,617 | 1735 |
| 7 | 50 | 31,10,20,19,4,3,6 | 70,20,39,37,7,5,10 | 107 |

Executing using the same arguments in the same order, the assembly solutions produced the following output:





All of the test outputs match the correct results given by the Rust implmentation, therefore the assembly implmentations are correct.

The following is the full assembly implmentation:

; A Sigma16 assembly program that implements both the recursive and dynamic

; algorithms to solve the 0-1 Knapsack Problem.

;

; Given a set of items, each having a positive integer weight and a positive

; integer value, and a maximum weight capacity, compute the maximum value you

; can achieve by either including or excluding items from your solution.

;

; Main program

; Sets up registers, calls knapsackRP(), knapsackDP(), prints results and terminates

;

; Register usage

; R1: N / result

; R2: W

; R3: pointer to weights

; R4: pointer to values

; R5: constant 0

; R13: return address

; R14: stack pointer

; Structure of stack frame

; 1[R14] origin of next stack frame

; 0[R14] 0 (no previous stack frame)

main ;Line 23

lea R14,stack[R0] ; Initialise stack pointer

store R0,0[R14] ; No previous stack frame

store R14,1[R14] ; Pointer to beginning of current frame

load R1,N[R0] ; R1 := N

load R2,W[R0] ; R2 := W

lea R3,weights[R0] ; R3 := weights

lea R4,values[R0] ; R4 := values

add R5,R0,R0 ; R5 := constant 0 (= i for knapsackRP)

lea R14,1[R14] ; Push stack pointer

jal R13,knapsackRP[R0] ; Call the recursive knapsack solution function

lea R14,1[R14] ; Push stack pointer

jal R13,println[R0] ; Call the println() function; param in R1

load R1,N[R0] ; Restore N into R1 after it was overwritten by knapsackRP

lea R14,1[R14] ; Push stack pointer

jal R13,knapsackDP[R0] ; Call the dynamic programming knapsack solution function

lea R14,1[R14] ; Push stack pointer

jal R13,println[R0] ; Call the println() function; param in R1

trap R0,R0,R0 ; Terminate

;

; function knapsackRP(N:R1, W:R2, weights:R3, values:R4, i:R5) -> return value:R1

;

; The basic recursive algorithm is:

; int knapsackRP(int N, int W, int weights[], int values[], int i) {

; if (W == 0 || i >= N) // If out of capacity or out of items

; return 0;

; if (weights[i] > W) // If weight of current item above weight limit

; return knapsackRP(N, W, weights, values, i + 1); // Skip to next item

;

; // Compute solution if the item is included

; int takeit = values[i] + knapsackRP(N, W - weights[i], weights, values, i + 1);

;

; // Compute solution if the item is excluded

; int leaveit = knapsackRP(N, W, weights, values, i + 1);

;

; return max(takeit, leaveit);

; }

;

; Stack frame stucture:

; 9[R14] origin of next stack frame

; 8[R14] solution if taken

; 7[R14] current N

; 6[R14] current i

; 5[R14] save R6

; 4[R14] save R7

; 3[R14] save R8

; 2[R14] save R12

; 1[R14] return address

; 0[R14] address to previous stack frame

; Register usage:

; R1: N/result

; R2: W

; R3: pointer to weights

; R4: pointer to values

; R5: i

; R6: weights[i]'s value

; R7: solution if taken

; R8: values[i]'s value

; R12: comparison results

; R13: return address

; R14: stack pointer

knapsackRP

; init stack frame

store R13,1[R14]

store R12,2[R14]

store R8,3[R14]

store R7,4[R14]

store R6,5[R14]

; If out of capacity or out of items, return 0

cmpeq R12,R2,R0

jumpt R12,return0[R0]

cmplt R12,R5,R1

jumpf R12,return0[R0]

; If weight of item above limit, skip to next item

load R6,weights[R5]

lea R5,1[R5] ; increment i

cmpgt R12,R6,R2

jumpf R12,try[R0]

store R14,9[R14] ; skip current item by making a recursive call with i+1

lea R14,9[R14]

jal R13,knapsackRP[R0]

jump return[R0]

try

; Compute solution if item is taken

sub R2,R2,R6

store R1,7[R14] ; store current N

store R5,6[R14] ; store current i

store R14,9[R14]

lea R14,9[R14] ; push stack frame

jal R13,knapsackRP[R0]

; restore values for next recursive call and store solution if taken

add R7,R1,R0 ; move result

load R1,7[R14] ; get current N

load R5,6[R14] ; get current i

lea R5,-1[R5]

load R6,weights[R5]

add R2,R2,R6 ; get W from W-weight[i]

load R8,values[R5]

add R7,R7,R8 ; solution if included

store R7,8[R14]

; compute solution if item is left

lea R5,1[R5] ; increment i

store R14,9[R14]

lea R14,9[R14] ; push stack frame

jal R13,knapsackRP[R0]

; find the max between taken and left solutions, storing it in R1 to return

load R7,8[R14]

cmpgt R12,R7,R1

jumpf R12,return[R0]

add R1,R7,R0 ; move solution if taken into R1 as is larger then if left

return

; cleanup stack frame and return

load R13,1[R14]

load R12,2[R14]

load R8,3[R14]

load R7,4[R14]

load R6,5[R14]

load R14,0[R14]

jump 0[R13]

return0

; set the result to 0 and return, used when out of items/capacity or weight of item exceeds weight limit

add R1,R0,R0

jump return[R0]

;

; function knapsackDP(N:R1, W:R2, weights:R3, values:R4) -> return value:R1

;

; The dynamic programming algorithm is:

; int knapsackDP(int N, int W, int weights[], int values[]) {

; int S[N+1][W+1]; // Allocate array as local variable

;

; for (i = 0; i <= W; i++)

; S[0][i] = 0; // Solution for 0 items is 0

;

; for (i = 1; i <= N; i++) // Loop over all items...

; for (w = 0; w <= W; w++) // and over all intermediate weight limits

; if (weights[i-1] > w) // If the current item doesn't fit...

; S[i][w] = S[i-1][w]; // then skip it...

; else

; // otherwise, compute the maximum of taking it or leaving it

; S[i][w] = max(S[i-1][w], S[i-1][w-weights[i-1]] + values[i-1]);

; return S[N][W];

; }

;

; Stack frame structure:

; <fill in your stack frame's structure...>

; 0[R14] pointer to previous stack frame

;

; Register usage:

; R1: N/result

; R2: W

; R3: pointer to weights

; R4: pointer to values

; R5: i | addr of S[N][W]

; R6: w

; R7: addr of S[i][w] | addr of S[i-1][w] | w-weights[i-1] | S[i-1][w-weights[i-1]] + values[i-1] | new S[i][w] value

; R8: value at S[i-1][w] | addr of S[i-1][w-weights[i-1]] i| S[i-1][w]

; R10: i-1

; R11: pointer to S

; R12: comparison results

; R13: return address

; R14: stack pointer

knapsackDP

; init stack frame

store R11,1[R14]

store R12,2[R14]

store R8,3[R14]

store R7,4[R14]

store R6,5[R14]

store R5,6[R14]

lea R11,7[R14] ; init pointer to S as next memory after saved registers

lea R5,-1[R0] ; set i = -1

; zero the array

loopZero

lea R5,1[R5]

cmpgt R12,R5,R1

jumpt R12,breakZero[R0]

add R6,R0,R0

loopZeroInner

cmpgt R12,R6,R2

jumpt R12,loopZero[R0]

mul R7,R5,R2

add R7,R7,R6

add R7,R7,R11

store R0,0[R7]

lea R6,1[R6]

jump loopZeroInner[R0]

breakZero

add R5,R0,R0 ; set i=0

loopn

lea R5,1[R5] ; inc i

cmpgt R12,R5,R1

jumpt R12,breakn[R0]

lea R6,-1[R0] ; set w=-1

loopw

lea R6,1[R6] ; inc w

cmpgt R12,R6,R2

jumpt R12,loopn[R0]

lea R7,-1[R5] ; i-1

add R7,R7,R3 ; addr of weights[i-1]

load R7,0[R7] ; weights[i-1]

cmpgt R12,R7,R6

jumpf R12,noskip[R0]

lea R7,-1[R5] ; i-1

mul R7,R7,R2 ; W(i-1)

add R7,R7,R6

add R7,R7,R11 ; addr of S[i-1][w]

load R8,0[R7] ; S[i-1][w]

lea R7,-1[R5] ; i-1

mul R7,R7,R2 ; W(i-1)

add R7,R7,R6 ;

add R7,R7,R11 ;

store R8,0[R7];

jump loopw[R0] ; goto next iteration

; item fits so test it

noskip

lea R10,-1[R5] ; i-1

add R7,R10,R3 ; addr of weights[i-1]

load R7,0[R7] ; weights[i-1]

sub R7,R6,R7 ; w-weights[i-1]

mul R8,R10,R2 ; W(i-1)

add R8,R7,R8

add R8,R8,R11 ; addr of S[i-1][w-weights[i-1]]

load R7,0[R8] ; S[i-1][w-weights[i-1]]

add R8,R10,R4 ; addr of values[i-1]

load R8,0[R8] ; values[i-1]

add R7,R7,R8 ; S[i-1][w-weights[i-1]] + values[i-1]

mul R8,R10,R2 ; W(i-1)

add R8,R6,R8

add R8,R8,R11

load R8,0[R8] ; S[i-1][w]

cmpgt R12,R8,R7

jumpf R12,store[R0]

add R7,R8,R0

store

mul R8,R5,R2

add R8,R6,R8

add R8,R8,R11

store R7,0[R8]

jump loopw[R0]

breakn

mul R5,R1,R2

add R5,R2,R5

add R5,R5,R11

load R1,0[R5]

load R11,1[R14]

load R12,2[R14]

load R8,3[R14]

load R7,4[R14]

load R6,5[R14]

load R5,6[R14]

load R14,0[R14]

jump 0[R13]

; Function println(num:R1)

; Converts the value in R1 to printable decimal digits and prints it on screen

;

; Stack frame structure:

; (1+(num of base 10 digits))[R14]..(1+(2\*(num of base 10 digits)))[R14] charArray

; 8[R14]..(num of base 10 digits)[R14] charStack

; 1[R14]..7[R14] caller's R2-R6 and R10-R12 registers in that order

; 0[R14] pointer to previous stack frame

;

; Register usage:

; R1: input number | current charStack pointer for reversal

; R2: base to convert to (10) | ASCII LF char (10)

; R3: length of string

; R4: minimum charStack pointer

; R5: modulo value | constant 1 for decrementation of charStack pointer | constant 2 for print trap

; R6: current stack value

; R10: string array pointer

; R11: string stack pointer

; R12: comparison results

; R13: return address

; R14: stack pointer

println

; Setup stack frame

store R2,1[R14]

store R3,2[R14]

store R4,3[R14]

store R5,4[R14]

store R6,5[R14]

store R10,6[R14]

store R11,7[R14]

store R12,8[R14]

; Initialise variables for decoding

lea R11,9[R14] ;init string stack pointer

lea R2,10[R0]

; Iterate of all powers of ten less than input number, storing the storing the difference

; from input number and the power of ten at string stack point, subtracting said difference

; and incrementing the string length each iteration each iteration

decode

cmpgt R12,R2,R1

jumpt R12,endDecode[R0]

div R5,R1,R2 ;next 3 lines find R1 % R3

mul R5,R5,R2

sub R5,R1,R5

div R1,R1,R2 ;Remove found modulo from number

store R5,0[R11]

lea R11,1[R11] ;Push string stack pointer

jump decode[R0]

endDecode

store R1,0[R11];Append last digit

lea R11,1[R11] ;increment charStack pointer

sub R3,R11,R14 ;Length of string is the difference in the stack pointer and the pointer to the last digit + 8

lea R3,-8[R3]

lea R5,1[R0]

add R10,R0,R11 ;set initial string array pointer

add R1,R10,R0 ;set initial stack reversal into array array pointer

lea R4,9[R14] ;set minimum string stack pointer

; Iterate in reverse over the string stack, moving each value into string array: inverting the string

flip

cmpeq R12,R4,R11

jumpt R12,endFlip[R0]

sub R11,R11,R5

load R6,0[R11]

lea R6,48[R6]

store R6,0[R1]

add R1,R1,R5

jump flip[R0]

endFlip

; Append newline to string array, print string array and return

add R4,R10,R3

lea R6,-1[R4]

store R2,0[R6]

lea R5,2[R0]

trap R5,R10,R3

; Restore stack frame

load R2,1[R14]

load R3,2[R14]

load R4,3[R14]

load R5,4[R14]

load R6,5[R14]

load R10,6[R14]

load R11,7[R14]

load R12,8[R14]

load R14,0[R14]

jump 0[R13]

; Data segment

; should produce 9

;N data 3

;W data 5

;weights data 1

; data 2

; data 3

;values data 4

; data 3

; data 5

;stack data 0

; should produce 51

;N data 5

;W data 26

;weights data 12

; data 7

; data 11

; data 8

; data 9

;values data 24

; data 13

; data 23

; data 15

; data 16

;stack data 0

;N data 7

;W data 170

;weights data 41

; data 50

; data 49

; data 59

; data 55

; data 57

; data 60

;values data 442

; data 525

; data 511

; data 593

; data 546

; data 564

; data 617

;stack data 0

N data 7

W data 50

weights data 31

data 10

data 20

data 19

data 4

data 3

data 6

values data 70

data 20

data 39

data 37

data 7

data 5

data 10

stack data 0