Munin: Execution-Time Space Complexity Analysis

Fall 2023 CS 335 Final Project

Eleftheria Beres

Dec 2023

1 Introduction

Analyzing the time complexity is a common task in theory computer science, computational complexity theory, and the design of algorithms. Just as important, although less well-understood, is analyzing algorithms' space complexity. That is, understanding how much memory is used in the execution of algorithms compared to the size of their inputs.

Space complexity analysis is often theoretically performed with multi-tape Turing Machines—space complexity has the nice feature that multi-tape and single-tape Turing Machines computing the same algorithm fall into the same space complexity class. Using multi-tape Turing Machines also allows for sublinear space usages; i.e., algorithms that use a less-than-linear amount of memory in their execution as a function of their input sizes. Consider a Turing Machine trying to determine if a binary integer x is a palindrome that does the following:

1. Writes down the length l of x
2. Starts a counter i at 0
3. Stores the index $l - i - 1$ as j
4. Writes down the ith bit of x as a
5. Writes down the jth bit of x as b
6. Compares a and b; if these are not equal, we reject; otherwise, we continue
7. Increments i by 1
8. Compares i to l; if i is less than l, go to line 3; otherwise, we've checked the entire binary integer and found it to be a palindrome so we accept

We will give this Turing Machine five tapes. The first tape will have the input written on it. The second will be where we write down l, the third where we write down i, the fourth where we write down j, and the fifth will be where we write down a and b. Note that the fifth tape always used 2 bits. Tapes two through four will write down numbers between 0 and the length of x, l. As the length of l in binary is bounded by $\log(l)$, so is the length of what's written on these tapes. Therefore, the space complexity of this algorithm—not counting tape one—is $4\log(l) + 2$. In other words, this algorithm uses sublinear—logarithmic in this case—space.

Measuring such usage in actual code is either very difficult or nearly impossible. For one, there is often no way to separate the size of the input from the amount of memory used by the algorithm. Additionally, since, in most languages, values are stored in variables with constant sizes—for example, u32s in Rust—the memory usage of the code will depend not on the size of the inputs, but instead on the size of the variables chosen. This makes measuring space complexity a much more challenging endeavor than measuring time complexity despite its value for understanding the execution-time requirements of algorithms and for teaching theoretical computer science.

Therefore, I present Munin, a tool for measuring the execution-time space complexity of algorithms. Munin works by separating inputs from the rest of its memory and by storing values in raw bit-vectors, enabling it to easily measure the exact number of bits used by an algorithm during its execution.

Here, I describe Munin and use it to analyze the space complexity of four algorithms:

1. the algorithm for deciding if an input is

a palindrome described above: PAL

2. an algorithm for deciding if $x + y = z$ in linear space: LIN-ADD
3. an algorithm for the same problem in sublinear logarithmic space: ADD
4. an algorithm for deciding if $x + y$ is a palindrome in logarithmic space: PAL-ADD

2 Munin

Munin is a Rust program that comes with the ability to run complexity analysis on 4 different Munin assembly programs. Munin can be downloaded from the GitHub repository at `http://www.github.com/ellifteria/munin`. The repository contains instructions for installing and running Munin.

Munin creates a virtual machine with sets of input, single-bit, and regular variables and three operating phases: IDLE, INPUT, and EXECUTION. The default and start phase is the IDLE phase. During the IDLE phase, the Munin device can take four actions. It can be moved into either the INPUT or EXECUTION phase, it can print out the most recently stored variable values, it can load a program, or it can clear the device memory. However, no memory can be written during the IDLE phase.

If the device is moved into the INPUT phase, users can either manually set input variables or run an input program that sets input variable values. This is the only phase of the Munin device in which input variables can be written; during EXECUTION, the device will use these input variables as its inputs. During the INPUT phase, users can also read or write to the bit and regular variables. However, these values will be cleared once the device enters the EXECUTION phase.

Munin programs are written in the Munin assembly language provided in Appendix C. Additional information on the Munin assembly language is provided in the Munin GitHub repository.

In the EXECUTION phase, the device runs the program loaded by the user in the IDLE phase. While the program runs, it can read from the input, bit, and regular variables but can only write to the bit and regular variables. This allows Munin to separate the input and execution-time memory usage, allowing users to determine exactly how much input and execution-time memory is used separately. Once a program has finished execution, the device can count up the total number and size of variables to determine how much space was used by the program; the sizes or variables are determined by the maximum number of bits that were stored in the variable throughout program execution.

By separating memory in this way, Munin can separately determine the size of the inputs and write memory. Since all variables are either a bit or a vector of bits—as opposed to being constant-sized values such as u32s in Rust—Munin is able to count exactly the number of bits required by algorithms without the overhead required by other programming languages.

3 Space complexity analysis using Munin

To validate Munin, I analyzed the PAL algorithm described in the introduction. Since this algorithm is known to operate in logarithmic space, we would expect this to be reflected in the Munin output. The Munin assembly code for this algorithm is given in Appendix A.1. Running this through Munin gives the following output:

```
COMPLEXITY ANALYSIS

INPUT LENGTH | MEMORY USED
-------------+------------
1            | 10
-------------+------------
2            | 13
-------------+------------
4            | 16
-------------+------------
8            | 19
-------------+------------
16           | 22
```

Plotting this shows the following; note that the left plot uses a linear scale for both the x- and y-axis and the right plot uses a

linear scale for the y-axis and a logarithmic scale for the x-axis (this convention will be followed for all plots in this section):

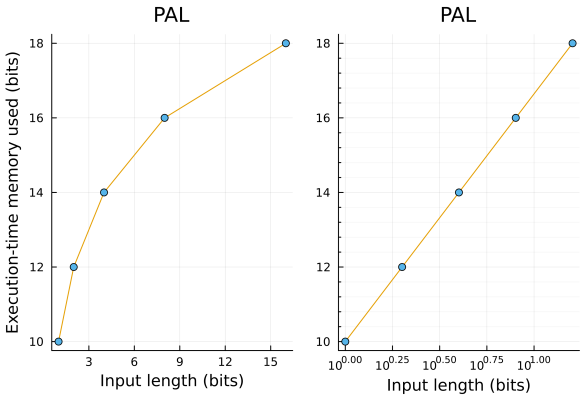

Figure 1: Execution-time memory usage (bits) vs. input length (bits) for the LSPACE PAL algorithm given in Appendix A.1

From these plots, it is clear that the PAL algorithm runs in logarithmic space relative to the input lengths. This is the expected result.

I next tested Munin on an algorithm known to use linear space with respect to its input size. This algorithm is a linear space ADD algorithm (LIN-ADD) that decides if $x + y = z$. The algorithm is shown in Appendix A.2. It is known to be a linear space algorithm since it copies the inputs x, y, and y. Then, the algorithm adds x and y and compares it to z. Running LIN-ADD through Munin gives the following output:

COMPLEXITY ANALYSIS

INPUT LENGTH	MEMORY USED
1	8
2	10
4	14
8	22
16	38

This gives the following plots which validate that the algorithm runs in linear space, as expected:

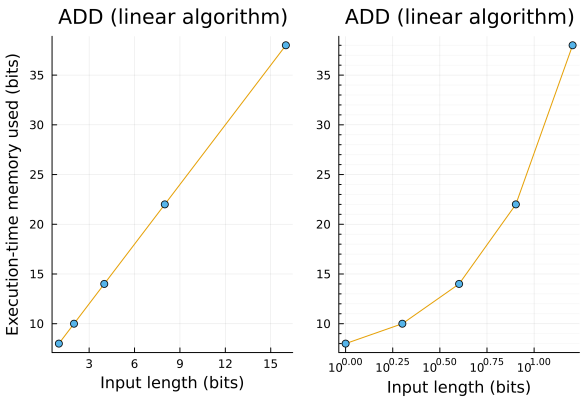

Figure 2: Execution-time memory usage (bits) vs. input length (bits) for the linear space ADD algorithm given in Appendix A.2

An additional two algorithms are analyzed: PAL-ADD and ADD. These are both algorithms that run in logarithmic space, proofs of this are written in B.

The output and plot of running ADD through Munin is the following:

COMPLEXITY ANALYSIS

INPUT LENGTH	MEMORY USED
1	10
2	12
4	14
8	16
16	18

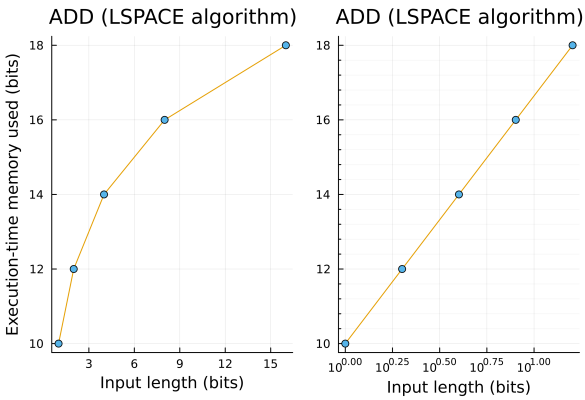

Figure 3: Execution-time memory usage (bits) vs. input length (bits) for the LSPACE ADD algorithm given in Appendix A.3

This demonstrates that ADD runs in logarithmic space or LSPACE.

The output and plot of running PAL-ADD through Munin is the following:

COMPLEXITY ANALYSIS

INPUT LENGTH	MEMORY USED
1	20
2	25
4	30
8	35
16	40

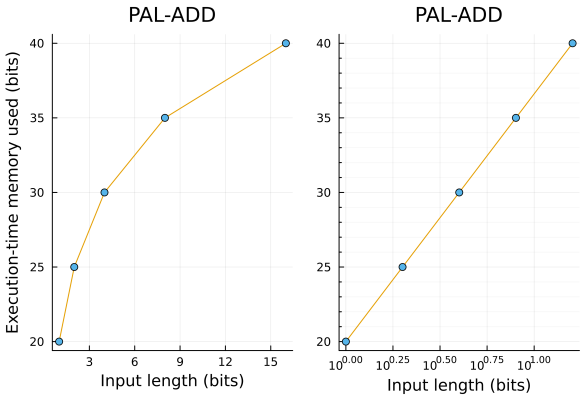

Figure 4: Execution-time memory usage (bits) vs. input length (bits) for the LSPACE PAL-ADD algorithm given in Appendix A.4

Likewise, this demonstrates that PAL-ADD runs in LSPACE.

4 Discussion

Here, I present Munin, a tool for measuring the execution-time memory usage of algorithms. I show that Munin can be used to demonstrate and determine the theoretical space complexity of various programs. I also demonstrate that LIN-ADD decides in linear space while PAL, ADD, and PAL-ADD decide in LSPACE.

The roadmap for future work on Munin first includes building a web-based application allowing users to write, run, and analyze Munin programs in the browser. This will provide users a platform for easily exploring space complexity through writing and testing actual algorithms. The next item on the roadmap is to expand the Munin assembly language to include ‘push‘ and ‘pop‘ operators and a function stack allowing users to write programs that include functions. A final roadmap item is the creation of a scripting language, MuninScript, that compiles to the Munin assembly language. This will allow users to more easily write Munin programs and study the space complexity of algorithms.

A Munin example algorithms

A.1 LSPACE PAL algorithm

A.1.1 Pseudo-code

```
1     function pal(x)
2         len = size_of(x)
3         i = 0
4         while i < len
5             j = len - i - 1
6             a = get_nth_bit_of(x, i)
7             b = get_nth_bit_of(x, j)
8             if a != b
9                 return false
10            end
11            i += 1
12        end
13        return true
14    end
```

A.1.2 Munin assembly code

```
1  set b00 0
2  set-to-length-of v00 i00
3  set v01 0
4  compare v01 v00
5  jump-over-next-if less
6  jump-to 16
7  set v02 v00
8  int-subtract v02 v01
9  int-subtract v02 1
10 set-to-nth-bit v03 i00 v01
11 set-to-nth-bit v04 i00 v02
12 compare v03 v04
13 jump-over-next-if equal
14 end
15 int-add v01 1
16 jump-to 3
17 set b00 1
18 end
```

A.2 Linear-space ADD algorithm

A.2.1 Pseudo-code

```
1     function lin_add(x, y, z)
2         a = x
3         b = y
4         c = z
5         a += b
6         if a != c
```

```
7               return false
8           end
9           return true
10      end
```

A.2.2 Munin assembly code

```
1  set v0 i0
2  set v1 i1
3  set v2 i2
4  int-add v0 v1
5  set b0 0
6  compare v0 v2
7  jump-over-next-if not-equal
8  set b0 1
9  end
```

A.3 LSPACE ADD algorithm

A.3.1 Pseudo-code

Note: it is assumed that the carry_flag is set automatically when binary addition is performed and is set to false at the start of each function.

```
1       function add(x, y, z)
2           len = size_of(z)
3           i = size_of(x)
4           if i > len
5               return false
6           end
7           i = size_of(y)
8           if i > len_z
9               return false
10          end
11          i = 0
12          while i < len
13              x_bit = get_nth_bit_of(x, i)
14              y_bit = get_nth_bit_of(y, i)
15              if carry_flag
16                  x_bit = add_bits(x_bit, 1)
17              end
18              x_bit = add_bits(x_bit, y_bit)
19              z_bit = get_nth_bit_of(z, i)
20              if x_bit != z_bit
21                  return false
22              end
23              i += 1
24          end
25          return true
26      end
```

A.3.2 Munin assembly code

```
set-to-length-of v00 i02
set-to-length-of v01 i00
compare v01 v00
jump-over-next-if greater
jump-to 7
set b00 0
end
set-to-length-of v01 i01
compare v01 v00
jump-over-next-if less-or-equal
jump-to 5
set v01 0
set-to-nth-bit v02 i00 v01
set-to-nth-bit v03 i01 v01
set-to-nth-bit v04 i02 v01
bit-add-with-carry v02 v03
compare v04 v02
jump-over-next-if not-equal
jump-to 21
set b00 0
end
int-add v01 1
compare v01 v00
jump-over-next-if greater-or-equal
jump-to 12
set b00 1
end
```

A.4 LSPACE PAL-ADD algorithm

A.4.1 Pseudo-code

Note: it is assumed that the carry_flag is set automatically when binary addition is performed and is set to false at the start of each function.

```
    function pal_add(x, y)
        len = size_of(x)
        len_tmp = size_of(y)
        if len_tmp > len
            len = len_tmp
        end
        carry_on_last = carry_on_last_of_sum(x, y)
        i = 0
        while i < len
            i_bit = get_nth_bit_of_sum(x, y, i)
            j = len - i - 1 + carry_on_last
            j_bit = get_nth_bit_of_sum(x, y, j)
            if i_bit != j_bit
                return false
            end
            i += 1
```

```
17          end
18          return true
19      end
20
21      function get_nth_bit_of_sum(x, y, n)
22          i = 0
23          while i < n
24              x_bit = get_nth_bit_of(x, i)
25              y_bit = get_nth_bit_of(y, i)
26              if carry_flag
27                  x_bit = add_bits(x_bit, 1)
28              end
29              x_bit = add_bits(x_bit, y_bit)
30              i += 1
31          end
32          return x_bit
33      end
34
35      function carry_on_last_of_sum(x, y)
36          len = size_of(x)
37          len_tmp = size_of(y)
38          if len_tmp > len
39              len = len_tmp
40          end
41          i = 0
42          while i < len
43              x_bit = get_nth_bit_of(x, i)
44              y_bit = get_nth_bit_of(y, i)
45              if carry_flag
46                  x_bit = add_bits(x_bit, 1)
47              end
48              x_bit = add_bits(x_bit, y_bit)
49              i += 1
50          end
51          if carry_flag
52              return 1
53          else
54              return 0
55          end
56      end
```

A.4.2 Munin assembly code

```
1  set-to-length-of v00 i00
2  set-to-length-of v01 i01
3  compare v01 v00
4  jump-over-next-if less
5  set v00 v01
6  set v02 0
7  jump-to 32
8  set v06 0
```

```
 9 compare v06 v00
10 jump-over-next-if less
11 jump-to 30
12 set v07 0
13 set v08 v06
14 set v09 0
15 jump-to 45
16 set v13 v09
17 set v08 v00
18 int-subtract v08 v06
19 int-subtract v08 1
20 int-add v08 v02
21 set v07 1
22 jump-to 45
23 set v14 v09
24 compare v13 v14
25 jump-over-next-if not-equal
26 jump-to 28
27 set b00 0
28 end
29 int-add v06 1
30 jump-to 8
31 set b00 1
32 end
33 set v03 0
34 set-to-nth-bit v04 i00 v03
35 set-to-nth-bit v05 i01 v03
36 bit-add-with-carry v04 v05
37 int-add v03 1
38 compare v03 v00
39 jump-over-next-if greater-or-equal
40 jump-to 33
41 set v02 0
42 jump-over-next-if no-carry
43 set v02 1
44 clear-flags
45 jump-to 7
46 clear-flags
47 set v10 0
48 set-to-nth-bit v11 i00 v10
49 set-to-nth-bit v12 i01 v10
50 bit-add-with-carry v11 v12
51 int-add v10 1
52 compare v10 v08
53 jump-over-next-if greater
54 jump-to 47
55 set v09 v11
56 compare v07 0
57 jump-over-next-if not-equal
58 jump-to 15
59 jump-to 22
```

B Proofs of space complexity

B.1 ADD

Theorem B.1. *ADD* $= \{\langle x, y, z\rangle : x + y = z\} \in L$

Proof. Let x, y, z be arbitrary binary integers. Here, it will be shown that if $x + y = z$ can be decided in LOG-SPACE.

Consider the ADD algorithm given in Appendix A.3.

This algorithm determines if the sum of two binary integers x, y is a third binary integer z.

To show that it does so in LOG-SPACE, consider the variables used by add. There are 8 such variables. x_bit, y_bit, z_bit, and carry_flag are all just a single bit each. len_x, len_y, len_z, and i can all hold values between 0 and $n = max(n_x, n_y, n_z)$ where n_x, n_y, n_z are the sizes of x, y, z respectively in bits. Therefore, these four variables all require at most $\log_2(n)$ bits. Thus, the total memory used by pal_add is $4\log_2(n) + 4$ bits meaning add is a LOG-SPACE algorithm.

Therefore, whether or not $x + y = z$ can be decided in LOG-SPACE. Because x, y, z are arbitrary this is the case for any binary integers.

Thus, ADD $= \{\langle x, y, z\rangle : x + y = z\} \in L$. ■

B.2 PAL-ADD

Theorem B.2. *PAL-ADD* $= \{\langle x, y\rangle : x + y$ *is a palindrome*$\} \in L$

Proof. Let x, y be arbitrary binary integers. Here, it will be shown that if $x+y$ is a palindrome can be decided in LOG-SPACE.

Consider the PAL-ADD algorithm given in Appendix A.4.

This algorithm determines if the sum of two binary integers is a palindrome.

To show that it does so in LOG-SPACE, consider the variables used by pal_add. There are 7 such variables. i_bit , j_bit , and carry_on_last are all just a single bit each. len, len_tmp, i, and j can all hold values between 0 and $n = max(n_x, n_y)$ where n_x, n_y are the sizes of x, y respectively in bits. Therefore, these four variables all require at most $\log_2(n)$ bits. Thus, the total memory used by pal_add is $4\log_2(n) + 3$ bits meaning pal_add is a LOG-SPACE algorithm.

get_nth_bit_of_sum and carry_on_last_sum use the same program flow as add from Theorem B.1 and use no more variables than add does. Since Theorem B.1 shows that add decides in LOG-SPACE, these do as well. Thus, all algorithms used in computing pal_add use a logarithmic amount of space relative to the size of the input.

Therefore, whether or not $x + y$ is a palindrome can be decided in LOG-SPACE. Because x, y are arbitrary this is the case for any binary integers.

Thus, PAL-ADD $= \{\langle x, y\rangle : x + y$ is a palindrome$\} \in L$. ■

C Munin assembly reference

Munin assembly reference				
Operator	Operand 1	Operand 2	Operand 3	Description
Assignment operaions				
set	D	S		Sets variable D equal to the value of S
stl	D	S		Sets variable D equal to the length of S
stnb	D	S	N	Sets variable D equal to the Nth bit of S
Integer arithmetic operaions				
iadd	D	S		Sets variable D equal to the value of D + S
isub	D	S		Sets variable D equal to the value of D - S
Binary arithmetic operaions				
badd	D	S		Sets one-bit variable D equal to the value of the binary sum of one-bit D and one-bit S; sets carry flag
bsub	D	S		Sets one-bit variable D equal to the value of the binary subtraction of one-bit D and one-bit S; sets the underflow flag
bsr	D	S		Sets variable D equal to the value of D « S
bsl	D	S		Sets variable D equal to the value of D » S
Comparison operaions				
cmp	A	B		Sets the equal flag if A == B ; sets the greater flag if A > B
clf				Clears all flags
Program flow operaions				
jmp	L			Jumps to line L
jon	C			Jumps over the next instruction if the condition C is true
end				Ends the program