CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 4 Dictionaries & Arrays
Sep 6, 2016

Dictionaries & Arrays

Dictionaries

Dict - standard key value store

ObjectIdDict - keys are object identities

WeakKeyDict - keys are weak references, so can be garbage collected

In DataStructures.jl

- Ordered Dicts - keys are in insertion order
- Dictionaries with Defaults - default values for missing keys
- Sorted Dict - keys in sorted order
- Sorted Multi-Dict - can have multiple keys that are the same

Dictionary Operations

```
a = Dict(:a => 1, :b => 2)
a[:b]                                   # 2
a[:c] = 3
a[:c]                                # 3
a[:d]                                   # KeyError: key :d not found
haskey(a, :d)                        #false
for (k,v) in a                                          # Key c Value 3
  println("Key $k Value $v")                    # Key a Value 1
end                                                          # Key b Value 2

get!(a, :e, 5)                      # 5
a[:e]                                      # 5
```

```
b = Dict(1 => "one", 2 => "two", "one" => 1)

e = Dict([(1, "one"), (2, "two")])

f = Dict{Int64,String}([(1, "one"), (2, "two")])
```

Arrays

Arrays (Column Vectors)

Creating

		Contents
		1, 3, 5, 7
		1, “cat”, 3.45
		3, 5
		0, -64, 25
		1, 2, 3, 4, 5
		1.0, 1.5, 2.0, 2.5, 3.0
		0, 0, 0, 0, 0
		1.0f0, 1.0f0, 1.0f0
		1.0, 1.0, 1.0

Arrays (Column Vectors)

vector1 = [1,3,5,7]

4-element Array{Int64,1}:
 1
 3
 5
 7

Previous slide shows them as row vectors just to save space

Array Initialization

Array(Int8, 3) 

Array elements are not zeroed out for you

Array Operations

```
vector = [7, 5, 3, 1]
vector[1]                       # returns 7
vector[1] = 2                   # vector == [2,5,3,1]
vector[2:4]                     # returns [5, 3, 1]
vector[2:4] = [1,2,3]           # vector == [2, 1, 2, 3]
vector[2:4] = 8                 # vector == [2, 8, 8, 8]
vector[2:4] = [3,4]             # DimensionMismatch
vector = [7, 5, 3, 1]
push!(vector,9)                 # vector == [7, 5, 3, 1, 9]
pop!(vector)                    # returns 9, vector == [7, 5, 3, 1]
append!(vector,[2,4])           # vector == [7, 5, 3, 1, 2, 4]
prepend!(vector,[0])            # vector == [0, 7, 5, 3, 1, 2, 4]
resize!(vector, 3)              # vector == [0, 7, 5]
resize!(vector, 8)              # vector == [0, 7, 5, 0, 1, 2, 4, 37507606382]
```

Array Operations

```
vector = [7, 5, 3, 1]
insert!(vector,2,9)                 # vector == [7, 9, 5, 3, 1]
deleteat!(vector,5)                 # vector == [7, 9, 5, 3]
deleteat!(vector,[1:2])             # vector == [5, 3]
in(4, [1, 2, 3, 5])                 # false
∈(4, [1, 2, 3, 5])                  # false
∉(4, [1, 2, 3, 5])                  # true
findfirst([2,3,1,3,4],3)            # 2 (index of location of 3)
findnext([4,3,1,3,4],4,2)           # 5
sum([7, 5, 3, 1])                   # 16
prod([7, 5, 3, 1])                  # 105
maximum([7, 5, 3, 1])               # 7
minimum([7, 5, 3, 1])               # 1
reverse([1,2,3,4])                  # returns [4, 3, 2, 1]
reverse([1,2,3,4],2,3)              # returns [1, 3, 2, 4]
shuffle([1,2,3,4])                  # returns [2, 3, 4, 1]   (different each time)
[1,2] + [3, 4]                      # returns [4, 6]
[1,2] .* [3, 4]                     # returns [3, 8]
```

Array Operations

```
mean([19, 7, 5, 3, 1])          # 7.0
median([19, 7, 5, 3, 1])        # 5.0
std([19, 7, 5, 3, 1])           # 7.07106781186547 55
var([19, 7, 5, 3, 1])           # 50.0
```

Julia is designed for math and data analysis

So has large library of functions on arrays

Array operations

http://docs.julialang.org/en/release-0.4/manual/arrays/

Math operations

http://docs.julialang.org/en/release-0.4/stdlib/math/

Row Vectors

```
column_vector = [4, 3, 2, 1]    # Vector Int64
row_vector = [4 3 2 1]          #1x4 Array{Int64,2}: 1  1  1  1
row_vector[2]                   # 3
row_vector[1,2]                 # 3
```

[4 3 2 1] is a two dimensional array

Common enough that they allow row_vector[2]

typealias Vector{T} Array{T,1} so Vector is an alias for a 1 dimensional array

Mulit-dimensional Arrays

array1 = [1 2 3]	1x3 Array{Int64,2}:	1 2 3
array2 = [1 2 3; 4 5 6]	2x3 Array{Int64,2}:	1 2 3 4 5 6
Array(Int8,2,3)	2x3 Array{Int8,2}:	100 115 114 101 99 105
cell(2)	Vector Any, 2	#undef #undef
cell(2,2)	2x2 Array{Any,2}	#undef #undef #undef #undef
zeros(Int64,2,4)	2x4 Array{Int64,2}:	0 0 0 0 0 0 0 0
rand(Int8,2,3)	2x3 Array{Int8,2}:	109 53 -99 104 56 -12
fill('a',3,2)	3x2 Array{Char,2}:	'a' 'a' 'a' 'a' 'a' 'a'
eye(3)	3x3 Array{Float64,2}	1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0
copy([1 2; 3 4])		

Destructuring Arrays

```
a,b,c  = [1, 2, 3, 4]
assert(a == 1)
assert(b == 2)
assert(c == 3)
```

```
d,e,f = [5 6 7 8 9]
assert(d == 5)
assert(e == 6)
assert(f == 7)
```

Array Operations

array = [1 2; 3 4; 5 6]	
array[1,2]	returns 2
array[2,1] = 9	1 2 9 3 5 6
array[2,[1:2]]	returns 9 4
array[2,[1:2]] = [7, 8]	1 2 7 8 5 6
array[3,1:2] = 0	1 2 7 8 0 0
[1 2; 3 4; 4 5 6]	:ArgumentError

Array Operations

```
findmax([1 2 3; 4 5 9])          # (9, 6)
findmax([1 9 3; 4 5 6 ])         # (9, 3)

sum([1 2 3; 4 5 6])              # 21

maximum([1 2 7; 3 4 6])          # 7
maximum([1 2 7; 3 4 6],1)        # 3 4 7
maximum([1 2 7; 3 4 6],2)        #= 7
                                        6 =#

in(4, [1 2 3; 4 5 6])            # true

findfirst([1 2 3; 4 3 6], 3)     # 4
```

Vectors only

push!

pop!

append!

prepend!

resize!

insert!

deleteat!

resize

reverse

shuffle

Scalar & Array operations

	0.841470984807897O
	1.0
	0.841471 0.909297 0.141112 -0.756802

abs abs2 angle cbrt
airy airyai airyaiprime airybi airybiprime airyprime
acos acosh asin asinh atan atan2 atanh
acsc acsch asec asech acot acoth
cos cospi cosh sin sinpi sinh tan tanh sinc cosc
csc csch sec sech cot coth
acosd asind atand asecd acscd acotd
cosd sind tand secd cscd cotd
besselh besseli besselj besselj0 besselj1 besselk bessely bessely0 bessely1
exp erf erfc erfinv erfcinv exp2 expm1
beta dawson digamma erfcx erfi
exponent eta zeta gamma
hankelh1 hankelh2
ceil floor round trunc
isfinite isinf isnan
lbeta lfact lgamma
log log10 log1p log2
copysign max min significand
sqrt hypot

Vectorizing Scalar Functions

```
function foo(x)
  x + 1
end


foo(2)                    # 3
foo([1,2,3])          # Error
```

```
@vectorize_1arg Number foo

foo([1 2 3])          # [2 3 4]
foo([1 2; 3 4])      # [2 3
                                  4 5]
```

Matrix Operations

3* [1,2]	Vector Int64, 2	3 6
3*[1 2;3 4]	2x2 Array{Int64,2}:	3 6 9 12
[1 2 3]*[4,5,6]	Vector Int64, 1	32
[4,5,6]*[1 2 3]	3x3 Array{Int64,2}:	4 8 12 5 10 15 6 12 18
[1 2; 3 4]*[5,6]	Vector Int64, 1	17 39

Concatenation

cat(k, A...) concatenate input n-d arrays along the dimension k

```
a = [1 2; 3 4]
b = [5 6; 7 8]
```

```
cat(1, a, b)

4x2 Array{Int64,2}:
1  2
3  4
5  6
7  8
```

```
cat(2, a, b)

2x4 Array{Int64,2}:
 1  2  5  6
 3  4  7  8
```

```
cat(3, a, b)

2x2x2 Array{Int64,3}:
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
 5  6
 7  8
```

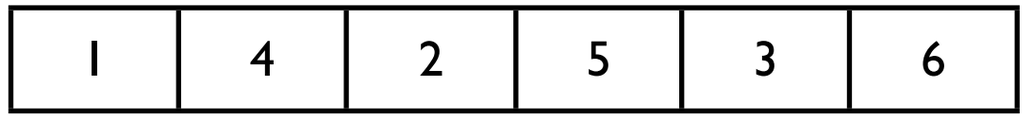
cat(1, a, b)
vcat(a, b)
[a;b]

cat(2, a, b)
hcat(a, b)
[a b]

Array Storage - Column wise

In memory as

[1 2 3; 4 5 6]



findfirst([1 2 3; 4 5 6],3) # 5

reshape([1 2 3; 4 5 6],1,6) # [1 4 2 5 3 6]

reshape([1 2 3; 4 5 6],6) # [1, 4, 2, 5, 3, 6]

Column storage - Why we need to know

```
function sum_by_rows(array)
  sum = zero(Int64)
  rows, columns = size(array)
  for row in 1:rows
    for column in 1:columns
      sum += array[row,column]
    end
  end
  sum
end
```

```
function sum_by_columns(array)
  sum = zero(Int64)
  rows, columns = size(array)
  for column in 1:columns
    for row in 1:rows
      sum += array[row,column]
    end
  end
  sum
end
```

large = fill(1,10_000,10_000)

	Time	Times Faster
@time sum_by_rows(large)	4.90 secs	
@time sum_by_columns(large)	0.14 secs	35.0
@time sum(large)	0.08 secs	1.7

Column storage - Why we need to know

Turning off bound checking
on array access

```
function sum_by_columns_nb(array)
  sum = zero(Int64)
  rows, columns = size(array)
  @inbounds for column in 1:columns
    for row in 1:rows
      sum += array[row,column]
    end
  end
  sum
end
```

large = fill(1,10_000,10_000)

	Time
@time sum_by_rows(large)	4.90 secs
@time sum_by_rows_nb(large)	4.40 secs
@time sum_by_columns(large)	0.14 secs
@time sum_by_columns_nb(large)	0.08 sec
@time sum(large)	0.08 secs

sum += array[row,column] What work is done

Compute the offset from begining of array

offset = column_length * (column - 1) + row

Add offset to start of array

location = offset + arraylocation

Fetch value at location & add to sum

How long does it take to do the addition?

```
function just_addition(n)
  sum = zero(Int64)
  for k in 1:n*n
    sum += 1
  end
  sum
end
```

```
function calculate_array_index(n)
  column_length = n
  location = zero(Int64)
  for row in 1:n, column in 1:n
    location += column_length * (column - 1) + row
  end
  location
end
```

	Time
@time sum_by_rows_nb(large)	4.40 secs
@time sum_by_columns_nb(large)	0.08 secs
@time sum(large)	0.08 secs
@time just_addition(10_000)	0.000004 secs
@time calculate_array_index(10_000)	0.000008 secs

What is left is accessing memory

Pages & 10_000 * 10_000 array

Int64 uses 8 bytes

10_000 * 10_000 * 8 = 800_000_000 bytes

```
large = fill(1,10_000, 10_000)
sizeof(large)                          # 800_000_000
```

800_000_000 bytes = 800MB or 796.9MB

Page size = 4096 bytes

800_000_000 / 4096 = 195_312.5 pages

All the time is spent getting the data to the processor!

This is not big data yet

Problem gets more challenging

	Time
@time sum_by_rows_nb(large)	4.40 secs
@time sum_by_columns_nb(large)	0.08 secs
@time sum(large)	0.08 secs
@time just_addition(10_000)	0.000004 secs
@time calculate_array_index(10_000)	0.000008 secs

Better written as

```
function sum_by_rows(array)
  sum = zero(Int64)
  rows, columns = size(array)
  for row in 1:rows
    for column in 1:columns
      sum += array[row,column]
    end
  end
  sum
end
```

```
function sum_by_rows(array)
  sum = zero(Int64)
  rows, columns = size(array)
  for row in 1:rows, column in 1:columns
      sum += array[row,column]
  end
  sum
end
```

Parallelizing the Code

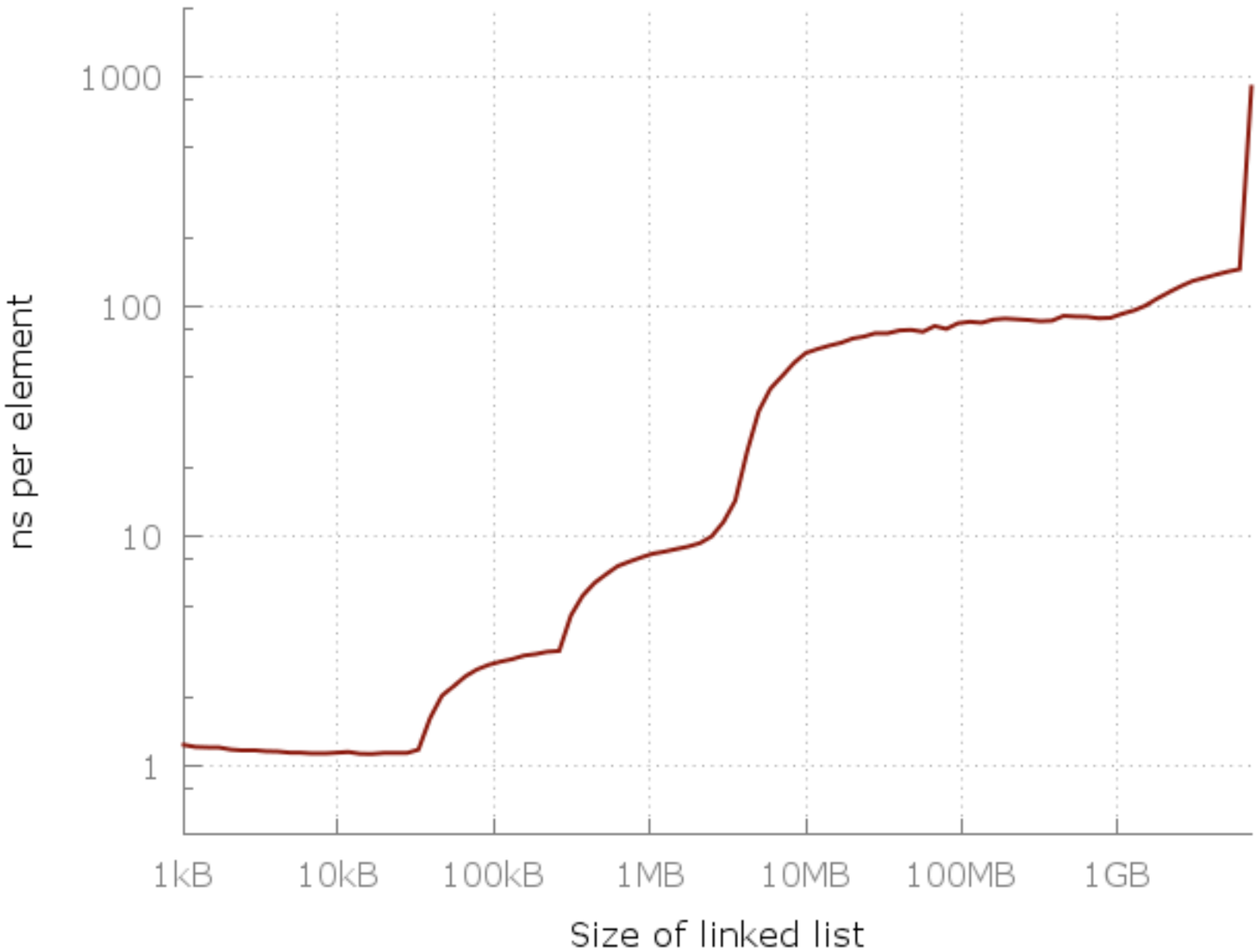
```
addprocs(1)       # only have two cores
function sum_by_columns_p(array)
  sum = zero(Int64)
  rows, columns = size(array)
   @parallel for column in 1:columns
      for row in 1:rows
      sum += array[row,column]
    end
  end
  sum
end
```

	Time
@time sum_by_columns(large)	0.14 secs
@time sum_by_columns_p(large)	1.5 sec

```
large = fill(1,10_000,10_000)
sum_by_columns_p(large)          # 0
```

Myth of Ram Access Being O(1)

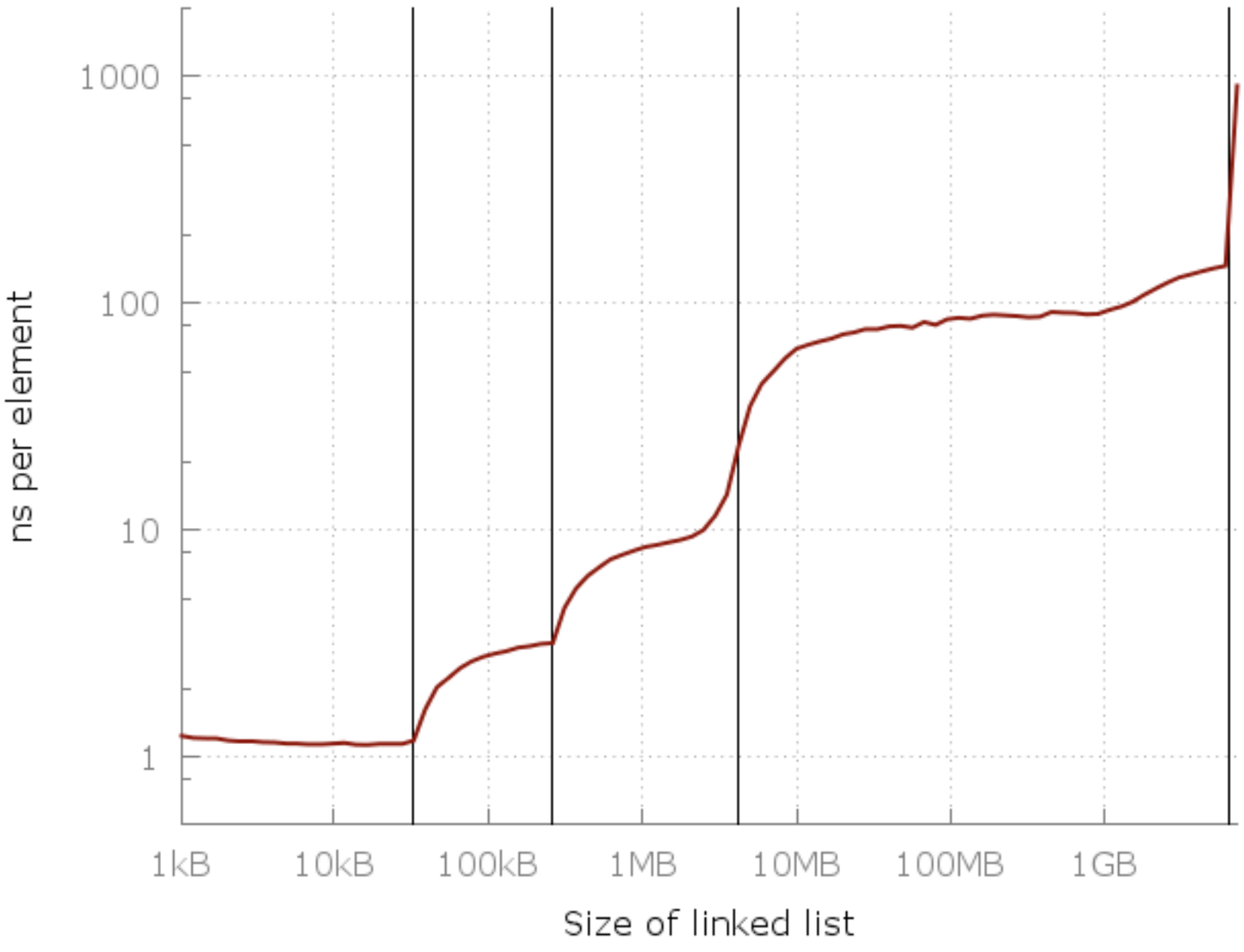
http://goo.gl/JwtF5v



Source: http://www.ilikebigbits.com/blog/2014/4/21/the-myth-of-ram-part-i

Myth of Ram Access Being O(1)

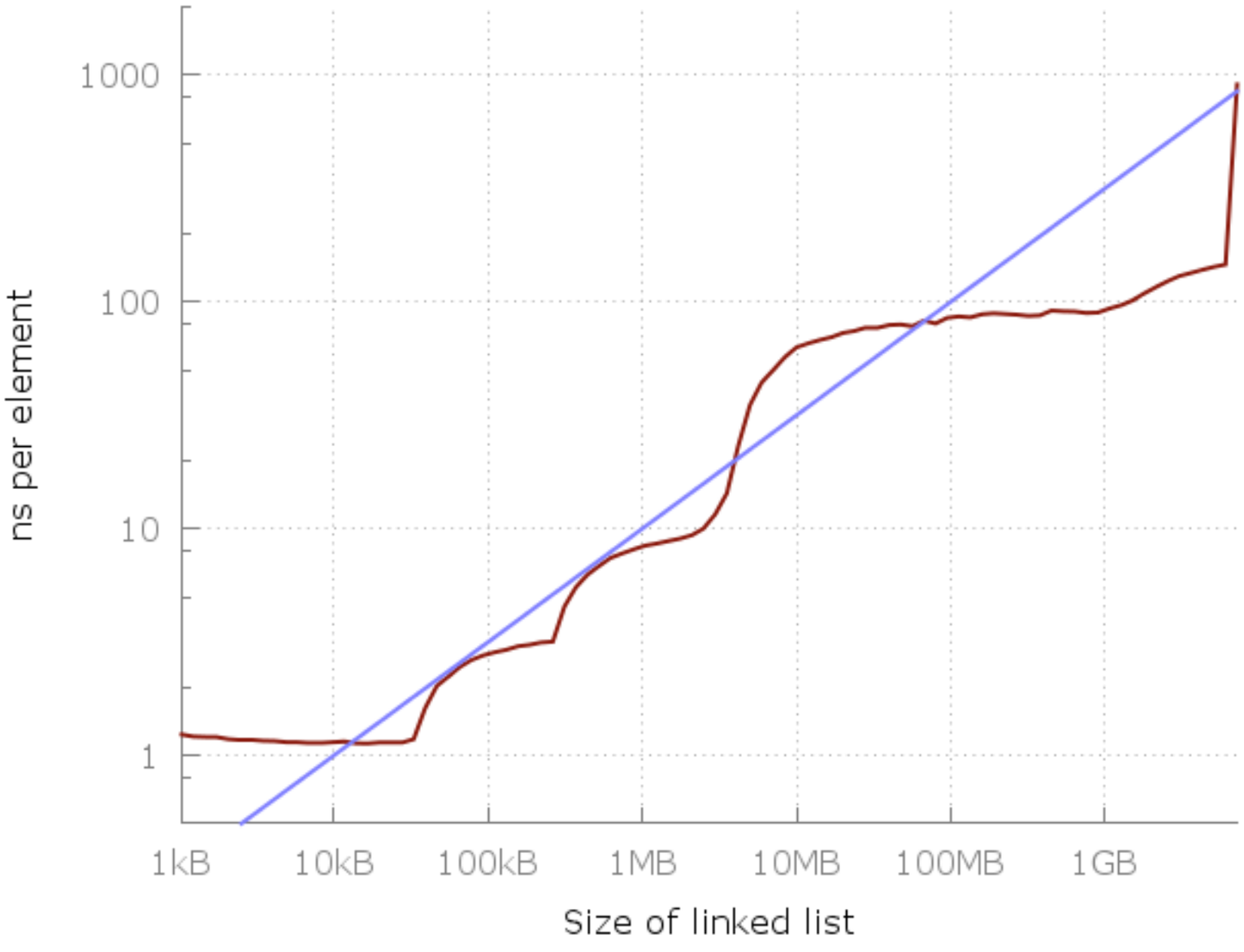
Lines - L1=32kiB, L2=256kiB, L3=4MB and 6 GiB of free RAM



Source: http://www.ilikebigbits.com/blog/2014/4/21/the-myth-of-ram-part-i

Myth of Ram Access Being O(1)

Blue Line = O(√N)



Tuesday, September 6, 16

Source: http://www.ilikebigbits.com/blog/2014/4/21/the-myth-of-ram-part-i