Physics 215 (1st Semester AY 2022-2023)

Richelle Jade L. Tuquero

Session 4 Fast Function Calls (OKR)

OBJECTIVE: Compare benchmark times of different implementation of functions that can be expressed as a recursion relation.

- [X] **KR1:** Benchmarked at least two (2) different implementation of the same function or process (e.g. raising each element of an array to some power `p`, random array may be used) that utilizes some parameter that can be considered a constant or declared globally. Typical methods: (1) Global variable, (2) Constant global variable, and (3) Named parameter variable.
- [X] **KR2:** Replicated the naive implementation of the polynomial in the textbook.
- [X] **KR3:** Replicated the naive implementation of the Horner's method for the same polynomial.
- [X] **KR4:** Replicated the macro implementation of the Horner's method of the same polynomial.
- [X] **KR5:** Table showing how many *minutes* will the function evaluations in both KR3 and KR4 be reduced if KR2 requires 24hours of runtime.

In this session, we will implement and achieve the necessary key results. Moreover, the methods that were replicated are from chapter 4 of the book Julia High Performance by Alan Edelman.

In [1]:
```julia
using Pkg;
Pkg.activate(".")
Pkg.add("BenchmarkTools")
Pkg.add("DataFrames")

Pkg.update()
Pkg.status()
```

```
  Activating project at `~/Desktop/Physics 215/Submission/Session 4`
    Updating registry at `~/.julia/registries/General.toml`
   Resolving package versions...
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Project.toml`
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Manifest.toml`
   Resolving package versions...
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Project.toml`
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Manifest.toml`
    Updating registry at `~/.julia/registries/General.toml`
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Project.toml`
  No Changes to `~/Desktop/Physics 215/Submission/Session 4/Manifest.toml`
Status `~/Desktop/Physics 215/Submission/Session 4/Project.toml`
  [6e4b80f9] BenchmarkTools v1.3.2
  [a93c6f00] DataFrames v1.4.3
```

KR1

Benchmarked at least two (2) different implementation of the same function or process (e.g. raising each element of an array to some power p, random array may be used) that utilizes some parameter that can be considered a constant or declared globally. Typical methods: (1) Global variable, (2) Constant global variable, and (3) Named parameter variable.

First we load `BenchmarkTools` and `DataFrames` in order to compare and show the benchmarks of different methods.

In [2]:
```julia
using BenchmarkTools
using DataFrames
```

In here we aim to create a function with different implementation depending on the variables. The function we will implement aims to take the sum of the elements of the array raised to a defined variable.

Global variable

We first implement a global variable by stating the value of a variable. In this implementation, we will refer to this as `p`.

In [3]:
```julia
p = 2; # Global variable

# Function using the global variable p.
function raise_sum(x::Vector)
    sum = zero(eltype(x))
    for i in x
        sum += i^p
    end
    return sum
end
```

Out[3]:
```
raise_sum (generic function with 1 method)
```

To benchmark the function using the global variable, we generate an array called as `data`.

In [4]:
```julia
data = rand(200_000);
```

Next, we implement `@benchmark` to get the benchmark of the function using the generated `data` as input and the global variable `p`. We also included another method which will just give the time for the benchmark called as `@btime`. However, we will only use `@benchmark` in the comparison of the time for the different methods.

In [5]:
```julia
mark1 = @benchmark raise_sum($data)
```

Out[5]:
```
BenchmarkTools.Trial: 669 samples with 1 evaluation.
 Range (min … max):  6.958 ms …  12.933 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     7.166 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   7.476 ms ± 750.423 μs  ┊ GC (mean ± σ):  3.47% ± 6.55%

 6.96 ms       Histogram: log(frequency) by time      10.6 ms <

 Memory estimate: 9.16 MiB, allocs estimate: 600000.
```

In [6]:
```julia
time1 = @btime raise_sum($data)
```

Out[6]:
```
  6.934 ms (600000 allocations: 9.16 MiB)
66823.36380191674
```

Based on the above results, it takes a median time of 7.166 ms to evaluate the function using a global variable.

To get a better look of the actions in the compiler and possible sources of error, we implement `@code_warntype`.

In [7]:
```julia
@code_warntype raise_sum(data)
```

```
MethodInstance for raise_sum(::Vector{Float64})
  from raise_sum(x::Vector) in Main at In[3]:4
Arguments
  #self#::Core.Const(raise_sum)
  x::Vector{Float64}
Locals
  @_3::Union{Nothing, Tuple{Float64, Int64}}
  sum::Any
  i::Float64
Body::Any
1 ─ %1  = Main.eltype(x)::Core.Const(Float64)
│         (sum = Main.zero(%1))
│   %3  = x::Vector{Float64}
│         (@_3 = Base.iterate(%3))
│   %5  = (@_3 === nothing)::Bool
│   %6  = Base.not_int(%5)::Bool
└──       goto #4 if not %6
2 ┄ %8  = @_3::Tuple{Float64, Int64}
│         (i = Core.getfield(%8, 1))
│   %10 = Core.getfield(%8, 2)::Int64
│   %11 = sum::Any
│   %12 = (i ^ Main.p)::Any
│         (sum = %11 + %12)
│         (@_3 = Base.iterate(%3, %10))
│   %15 = (@_3 === nothing)::Bool
│   %16 = Base.not_int(%15)::Bool
└──       goto #4 if not %16
3 ─       goto #2
4 ┄       return sum
```

From the output of the `@code_warntype`, we found that the compiler was unable to assign the type for `sum`. In here, it was noted that the type is `Any` resulting to slower time of the evaluation. Hence, one of the disadvantages of the global variable is that the compiler assigns the type `Any` since it is unable to assign a specific type.

Constant global variable

Next, we consider declaring `const` to a global variable `p2`.

In [8]:
```julia
const p2 = 2; # constant global variable
```

We also show that a constant global variable can change values but their type does not change.

In [9]:
```julia
p2 = 3
```

```
WARNING: redefinition of constant p2. This may fail, cause incorrect answers, or produce other errors.
```

Out[9]:
```
3
```

In [10]:
```julia
p2 = 3.0
```

```
invalid redefinition of constant p2

Stacktrace:
 [1] top-level scope
   @ In[10]:1
 [2] eval
   @ ./boot.jl:368 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1428
```

Then, we implement the same process as that in the **global variable** section to benchmark the result for constant global variable.

In [11]:
```julia
const pconst = 2;

# function for using constant global variable p2.
function const_raise_sum(x::Vector)
    sum = zero(eltype(x))
    for i in x
        sum += i^pconst
    end
    return sum
end
```

Out[11]:
```
const_raise_sum (generic function with 1 method)
```

In [12]:
```julia
mark2 = @benchmark const_raise_sum($data)
```

Out[12]:
```
BenchmarkTools.Trial: 8656 samples with 1 evaluation.
 Range (min … max):  572.417 μs … 720.541 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     572.875 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   575.614 μs ±   7.212 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 572 μs        Histogram: log(frequency) by time        607 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

From the result of the benchmark, we observe that using a global constant variable results to faster evaluation than using a global variable. Note that the time median is 572.875 μs. We also check for possible warnings and the type of the result using `@code_warntype`.

In [13]:
```julia
@code_warntype const_raise_sum(data)
```

```
MethodInstance for const_raise_sum(::Vector{Float64})
  from const_raise_sum(x::Vector) in Main at In[11]:4
Arguments
  #self#::Core.Const(const_raise_sum)
  x::Vector{Float64}
Locals
  @_3::Union{Nothing, Tuple{Float64, Int64}}
  sum::Float64
  i::Float64
Body::Float64
1 ─ %1  = Main.eltype(x)::Core.Const(Float64)
│         (sum = Main.zero(%1))
│   %3  = x::Vector{Float64}
│         (@_3 = Base.iterate(%3))
│   %5  = (@_3 === nothing)::Bool
│   %6  = Base.not_int(%5)::Bool
└──       goto #4 if not %6
2 ┄ %8  = @_3::Tuple{Float64, Int64}
│         (i = Core.getfield(%8, 1))
│   %10 = Core.getfield(%8, 2)::Int64
│   %11 = sum::Float64
│   %12 = (i ^ Main.pconst)::Float64
│         (sum = %11 + %12)
│         (@_3 = Base.iterate(%3, %10))
│   %15 = (@_3 === nothing)::Bool
│   %16 = Base.not_int(%15)::Bool
└──       goto #4 if not %16
3 ─       goto #2
4 ┄       return sum
```

We found that using the constant global variable results to no errors. Moreover, the type of the result is also assigned as `Float64` unlike the method using global constant which is `Any`.

Named Parameter Variable

Finally, we consider the method using a named parameter variable. For this implementation, we simply assign the value as a parameter for the function instead of using global variables.

In [14]:
```julia
# Named parameter variable par or function arguement
function par_raise_sum(x::Vector; par = 2)
    sum = zero(eltype(x))
    for i in x
        sum += i^par
    end
    return sum
end
```

Out[14]:
```
par_raise_sum (generic function with 1 method)
```

Then, we benchmark the function for named parameter variable.

In [15]:
```julia
mark3 = @benchmark par_raise_sum($data)
```

Out[15]:
```
BenchmarkTools.Trial: 8658 samples with 1 evaluation.
 Range (min … max):  572.417 μs … 714.625 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     572.959 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   575.353 μs ±   6.082 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 572 μs        Histogram: log(frequency) by time        604 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

We also check the function using `@code_warntype`.

In [16]:
```julia
@code_warntype par_raise_sum(data)
```

```
MethodInstance for par_raise_sum(::Vector{Float64})
  from par_raise_sum(x::Vector; par) in Main at In[14]:2
Arguments
  #self#::Core.Const(par_raise_sum)
  x::Vector{Float64}
Body::Float64
1 ─ %1 = Main.:(var"#par_raise_sum#1")(2, #self#, x)::Float64
└──      return %1
```

The output shows that there are no errors for using the named parameter variable method.

Parameter global variable

We benchmark the global variable as a named parameter variable.

In [17]:
```julia
mark4 = @benchmark par_raise_sum($data, par = p)
```

Out[17]:
```
BenchmarkTools.Trial: 8562 samples with 1 evaluation.
 Range (min … max):  572.625 μs …   1.591 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     575.459 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   582.136 μs ± 28.744 μs   ┊ GC (mean ± σ):  0.00% ± 0.00%

 573 μs        Histogram: log(frequency) by time        646 μs <

 Memory estimate: 48 bytes, allocs estimate: 3.
```

Parameter constant global variable

Lastly, we check the method using the constant global variable as a parameter variable.

In [18]:
```julia
mark5 = @benchmark par_raise_sum($data, par = pconst)
```

Out[18]:
```
BenchmarkTools.Trial: 8644 samples with 1 evaluation.
 Range (min … max):  572.416 μs … 697.291 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     572.958 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   576.245 μs ±   8.239 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 572 μs        Histogram: log(frequency) by time        610 μs <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

We are done benchmarking the functions for different methods. We now proceed on comparing the time it takes to evaluate the functions by including the ratio of the time for the global variable and the other methods.

In [19]:
```julia
speedup1 = median(mark1.times)/median(mark1.times)
speedup2 = median(mark1.times)/median(mark2.times)
speedup3 = median(mark1.times)/median(mark3.times)
speedup3a = median(mark1.times)/median(mark4.times)
speedup3b = median(mark1.times)/median(mark5.times)

table = DataFrame("Method"=>["Global"], "Speedup" => [speedup1]);
push!(table, ["Constant", speedup2]);
push!(table, ["Parametrized value", speedup3]);
push!(table, ["Parametrized global", speedup3a]);
push!(table, ["Parametrized const.imp", speedup3b]);

print(table)
```

5×2 DataFrame

Row	Method String	Speedup Float64
1	Global	1.0
2	Constant	12.5086
3	Parametrized value	12.5068
4	Parametrized global	12.4525
5	Parametrized const.imp	12.5068

Based on the results of the table, the use of the constant global variable is the fastest method in evaluating the function. Overall, the slowest is the global variable which as we previously mentioned results to an `Any` type for the result. Note that from the table, we found that all methods are approximately 12.5 times faster than using the global variable.

KR2

Replicated the naive implementation of the polynomial in the textbook.

The expression for a polynomial as given in the textbook is

$$p(x) = \sum_{i=0}^{n} a_i x^i. \tag{1}$$

Since the key result 2 aims to replicate the implementation of the polynomial in the textbook, we replicate the function `poly_naive()` in the textbook. For here, we refer to it as `naive_poly()` due to personal preference.

In [20]:
```julia
function naive_poly(x, a...)
    p = zero(x)
    for i in eachindex(a)
        p = p + a[i]*x^(i-1) # Equation of the polynomial
    end
    return p
end
```

Out[20]:
```
naive_poly (generic function with 1 method)
```

Just like in the textbook, we try to compute the function

$$f(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + 6x^5 + 7x^6 + 8x^7 + 9x^8 \tag{2}$$

which is referred to as `f_naive(x)` for the naive implementation of the function.

In [21]:
```julia
f_naive(x) = naive_poly(x,1,2,3,4,5,6,7,8,9)
```

Out[21]:
```
f_naive (generic function with 1 method)
```

Then, we benchmarked the function for a given value of `x` to compare with the other methods in KR5.

In [22]:
```julia
x = 8.5
markp0 = @benchmark f_naive($x)
```

Out[22]:
```
BenchmarkTools.Trial: 10000 samples with 995 evaluations.
 Range (min … max):  29.815 ns … 57.915 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     30.570 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   30.603 ns ±  0.895 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

 29.8 ns         Histogram: frequency by time        32.6 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

KR3

Replicated the naive implementation of the Horner's method for the same polynomial.

Horner's method implements the recursive relation

$$b_n = a_n \tag{3}$$
$$b_{n-1} = a_{n-1} + b_n x \tag{4}$$
$$b_{n-2} = a_{n-2} + b_{n-1} x \tag{5}$$
$$\vdots$$
$$b_0 = a_0 + b_1 x \tag{6}$$

where the polynomial $p(x)$ is given by b_0. Thus, we update the value of b until $n = 0$ or $n' = 1$ in Julia since Julia starts at 1.

Note that since we are asked to replicate the Horner's method, the function is given below as based on the textbook. Additionally, we refer to it as `horner_poly()` instead of `poly_horner()` due to personal preference.

In [23]:
```julia
function horner_poly(x, a...)
    b = zero(x)
    for i in reverse(eachindex(a))
        b = a[i]+b*x
    end
    return b
end
```

Out[23]:
```
horner_poly (generic function with 1 method)
```

Next, we evaluate Eq. (2) using the Horner's method.

In [24]:
```julia
f_horner(x) = horner_poly(x, 1,2,3,4,5,6,7,8,9)
```

Out[24]:
```
f_horner (generic function with 1 method)
```

Then, we get the benchmark for the Horner's method to be used in KR5.

In [25]:
```julia
markp1 = @benchmark f_horner($x)
```

Out[25]:
```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  2.458 ns … 13.041 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.583 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.572 ns ±  0.132 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

 2.46 ns       Histogram: log(frequency) by time      2.67 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

KR4

Replicated the macro implementation of the Horner's method of the same polynomial.

The macro implementation of the Horner's method uses the function `muladd`. To get a better idea on what it does we use `?`.

In [26]:
```julia
? muladd
```

search: **muladd**

Out[26]:

```
muladd(x, y, z)
```

Combined multiply-add: computes `x*y+z`, but allowing the add and multiply to be merged with each other or with surrounding operations for performance. For example, this may be implemented as an `fma` if the hardware supports it efficiently. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See `fma`.

Examples

```
jldoctest
julia> muladd(3, 2, 1)
7

julia> 3 * 2 + 1
7
```

```
muladd(A, y, z)
```

Combined multiply-add, `A*y .+ z`, for matrix-matrix or matrix-vector multiplication. The result is always the same size as `A*y`, but `z` may be smaller, or a scalar.

!!! compat "Julia 1.6" These methods require Julia 1.6 or later.

Examples

```
jldoctest
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; z=[0, 100];

julia> muladd(A, B, z)
2×2 Matrix{Float64}:
   3.0    3.0
 107.0  107.0
```

From the discussion of what `muladd(a, b,c)`, we know that this multiplies `a` and `b`, then add it with `c`

$$a * b + c. \tag{7}$$

Comparing this with Eq. (4), we can implement the Horner's method by setting `muladd(b,x,a)`. Note that below is a replication of the macro implementation of the Horner's method.

In [27]:
```julia
macro horner(x, p...)
    ex = esc(p[end])
    for i in length(p)-1:-1:1
        ex = :(muladd(t, $ex, $(esc(p[i]))))
    end
    Expr(:block, :(t=$(esc(x))), ex)
end
```

Out[27]:
```
@horner (macro with 1 method)
```

Just like the rest of the key results, we evaluate Eq. (2) and get its benchmark for KR5.

In [28]:
```julia
f_horner_macro(x) = @horner(x, 1,2,3,4,5,6,7,8,9)
```

Out[28]:
```
f_horner_macro (generic function with 1 method)
```

In [29]:
```julia
markp2 = @benchmark f_horner_macro($x)
```

Out[29]:
```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  2.458 ns … 17.250 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.542 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.564 ns ±  0.198 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

 2.46 ns       Histogram: log(frequency) by time      2.67 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

KR5

Table showing how many minutes will the function evaluations in both KR3 and KR4 be reduced if KR2 requires 24hours of runtime.

We then generate a table to show the the ratio of the speed for the naive implementation and the other methods.

In [30]:
```julia
speedup0 = median(markp0.times) / median(markp0.times)
speedup1 = median(markp0.times) / median(markp1.times)
speedup2 = median(markp0.times) / median(markp2.times)

# Generating table
table = DataFrame("Method"=>["Naive"],"Speedup" => [speedup0]);
push!(table, ["Horner", speedup1]);
push!(table, ["Macro", speedup2]);

print(table)
```

3×2 DataFrame

Row	Method String	Speedup Float64
1	Naive	1.0
2	Horner	11.835
3	Macro	12.0259

We observe that the macro implementation of the Horner method provides the fastest implementation of evaluating the polynomial function. Meanwhile the naive implementation is the slowest.

Next, we include the time it takes to run in minutes when the naive implementation is ran for 24 hours.

In [31]:
```julia
transform!(table,:Speedup=>ByRow(x->24/x)=>:"Time(hours)",
    :Speedup=>ByRow(x->24*60/x)=>:"Time(minutes)")
print(table)
```

3×4 DataFrame

Row	Method String	Speedup Float64	Time(hours) Float64	Time(minutes) Float64
1	Naive	1.0	24.0	1440.0
2	Horner	11.835	2.02788	121.673
3	Macro	12.0259	1.99569	119.742

To get a better comparison we include the time in hours and minutes. For the 24 hours runtime using the naive implementation, we observe a significant decrease of the runtime using the Horner and most especially the macro Horner's method. The macro implementation of the Horner's method finishes within 119.742 minutes while the Horner's method is 121.673 minutes.