Session 4: Fast Function Calls

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

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.

For this session, I will primarily follow the examples in the textbook

In [1]:
```julia
using BenchmarkTools
```

In [2]:
```julia
p = 2

function pow_array(x::Vector{Float64})
    s = 0.0
    for y in x
        s = s + y ^ p
    end
    return s
end
```

Out[2]:
```
pow_array (generic function with 1 method)
```

In [3]:
```julia
t = rand(100000)
@btime pow_array($t)
```

```
  5.851 ms (300000 allocations: 4.58 MiB)
```

Out[3]:
```
33260.95810623382
```

In [4]:
```julia
@code_warntype pow_array(t)
```

```
Variables
  #self#::Core.Const(pow_array)
  x::Vector{Float64}
  @_3::Union{Nothing, Tuple{Float64, Int64}}
  s::Any
  y::Float64

Body::Any
1 ─       (s = 0.0)
│   %2  = x::Vector{Float64}
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Float64, Int64}::Tuple{Float64, Int64}
│         (y = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│   %10 = s::Any
│   %11 = (y ^ Main.p)::Any
│         (s = %10 + %11)
│         (@_3 = Base.iterate(%2, %9))
│   %14 = (@_3 === nothing)::Bool
```

```
│    %15 = Base.not_int(%14)::Bool
└──        goto #4 if not %15
3 ─        goto #2
4 ┄        return s
```

In [5]:

```
const p2 = 2
function pow_array2(x::Vector{Float64})
    s = 0.0
    for y in x
        s = s + y^p2
    end
    return s
end
```

Out[5]:

```
pow_array2 (generic function with 1 method)
```

Speedup using a constant is very significant when setting p as a constant!

In [6]:

```
@btime pow_array2($t)
```

```
  111.183 μs (0 allocations: 0 bytes)
```

Out[6]:

```
33260.95810623382
```

In [7]:

```
@code_warntype pow_array2(t)
```

```
Variables
  #self#::Core.Const(pow_array2)
  x::Vector{Float64}
  @_3::Union{Nothing, Tuple{Float64, Int64}}
  s::Float64
  y::Float64

Body::Float64
1 ─       (s = 0.0)
│   %2  = x::Vector{Float64}
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Float64, Int64}::Tuple{Float64, Int64}
│         (y = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│   %10 = s::Float64
│   %11 = (y ^ Main.p2)::Float64
│         (s = %10 + %11)
│         (@_3 = Base.iterate(%2, %9))
│   %14 = (@_3 === nothing)::Bool
│   %15 = Base.not_int(%14)::Bool
└──       goto #4 if not %15
3 ─       goto #2
4 ┄       return s
```

In [8]:

```
function pow_array3(x::Vector{Float64})
    return pow_array_inner(x, p)
end

function pow_array_inner(x, pow)
    s = 0.0
    for y in x
        s = s + y ^ pow
    end
```

```
    return s
end
```

Out[8]:
```
pow_array_inner (generic function with 1 method)
```

In [9]:
```
@btime pow_array3($t)
```

```
  111.357 μs (1 allocation: 16 bytes)
```

Out[9]:
```
33260.95810623382
```

KR2: Replicated the naive implementation of the polynomial in the textbook.

In [10]:
```
function poly_naive(x, a...)
    p=zero(x)
    for i = 1:length(a)
        p = p + a[i] * x^(i-1)
    end
    return p
end
```

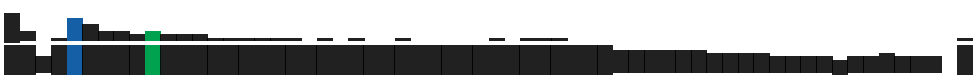
Out[10]:
```
poly_naive (generic function with 1 method)
```

In [11]:
```
f_naive(x) = poly_naive(x, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

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

In [12]:
```
x = 3.5
bench_naive = @benchmark f_naive($x)
```

Out[12]:
```
BenchmarkTools.Trial: 10000 samples with 709 evaluations.
 Range (min … max):  177.898 ns …   2.209 μs  ┊ GC (min … max): 0.00% … 90.85%
 Time  (median):     193.292 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   211.444 ns ± 67.260 ns   ┊ GC (mean ± σ):  0.44% ±  2.01%
```



```
  178 ns          Histogram: log(frequency) by time        394 ns <

 Memory estimate: 32 bytes, allocs estimate: 2.
```

KR3: Replicated the naive implementation of the Horner's method for the same polynomial.

In [13]:
```
function poly_horner(x, a...)
    b=zero(x)
    for i = length(a):-1:1
        b = a[i] + b * x
    end
    return b
end
```

Out[13]:
```
poly_horner (generic function with 1 method)
```

In [14]:
```
f_horner(x) = poly_horner(x, 1,2,3,4,5,6,7,8,9)
bench_horner = @benchmark f_horner($x)
```

Out[14]:
```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
```

```
 Range  (min … max):  4.884 ns …  4.262 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time   (median):     5.293 ns              ┊ GC (median):    0.00%
 Time   (mean ± σ):   8.030 ns ± 54.682 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  4.88 ns       Histogram: log(frequency) by time      30.9 ns <

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

KR4: Replicated the macro implementation of the Horner's method of the same polynomial.

In [15]:

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

Out[15]:

```
@horner (macro with 1 method)
```

In [16]:

```
f_horner_macro(x) = @horner(x, 1,2,3,4,5,6,7,8,9)
```

Out[16]:

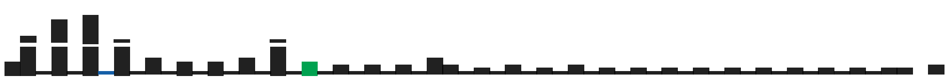
```
f_horner_macro (generic function with 1 method)
```

In [17]:

```
bench_macro = @benchmark f_horner_macro($x)
```

Out[17]:

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range  (min … max):  0.046 ns … 26.623 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time   (median):     0.049 ns              ┊ GC (median):    0.00%
 Time   (mean ± σ):   0.056 ns ±  0.282 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



```
  0.046 ns        Histogram: frequency by time       0.076 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.

In [18]:

```
method_names = ["Naive", "Horner", "Macro"];
speedup = [median(i.times) for i in [bench_naive, bench_horner, bench_macro]] / median(ber
minutes = speedup * 1440;;
```

In [19]:

```
using DataFrames
table = DataFrame("Method"=>method_names,"Speedup" => speedup, "Runtime" => minutes);

print(table)
```

```
3×3 DataFrame
 Row │ Method  Speedup      Runtime
     │ String  Float64      Float64
─────┼─────────────────────────────────
   1 │ Naive   1.0          1440.0
   2 │ Horner  0.0273834      39.4322
   3 │ Macro   0.000253503     0.365044
```

The speedup using the Horner method is very impressive, from a 1 day runtime to just 40 minutes and when using a macro it actually just takes a few seconds. I actually didn't expect that much of a speedup especially for the macro and I'll always consider this if ever I run into future bottlenecks.

In [ ]:

```

```