An Abstract Machine

Old times

In old times, computing time was computable

- register access: negligible
- integer: $\tau_{\text{int}}$, approx. 1 unit
- reals: $\tau_{\text{real}}$, approx. 10 units
- transcendental/non-polynomial: $\tau_{\text{trans}}$, approx. 100 units

Total time

\[ T = n_{\text{int}} \times \tau_{\text{int}} + n_{\text{real}} \times \tau_{\text{real}} + n_{\text{trans}} \times \tau_{\text{trans}} \]

\( \propto n_{\text{real}} + 10 \times n_{\text{trans}} \)

"complexity"

An Abstract Machine

Computing Background

An Abstract Machine

Old times have passed.

Today, time "constants"

- vary by configuration (check yours!)
- do not differ by magnitude

Total time is not additive (operations may overlap)

Still "complexity" may be a useful notion, in particular if related to parameters of the computation.

Optimisation has two possibilities:

- reduce complexity, e.g. by clever algorithms
- make efficient use of resources, e.g. by adapted implementation

The algorithm will be applied to data. Data management is a second field for optimisation.

Complexity

For your problem:

What is an appropriate measure for system size $n$?
What is the complexity, as a function of system size $n$?

Use a pragmatic definition, for your purposes only . . .
Abstract...ignoring all implementation details

Computers just cannot understand.

For an abstract view, think of computing in terms of four states:

\[
\text{Source} \rightarrow \text{intermediate code} \rightarrow \text{evaluation} \rightarrow \text{result}
\]

Abstract...ignoring all implementation details

What is missing in this picture is data. It was a major step towards programmable computing to understand that in principle data are not different from instructions.

The intended purpose makes the difference.

For our purpose, it may be helpful to think of data as a separate entity, at least available at the evaluation step.

\[
\text{Source} \rightarrow \text{data structure} \rightarrow \text{evaluation} \rightarrow \text{result}
\]

R Specifics

An R function

\[
\text{area} <- \text{function}(r) \pi \times r^2
\]

If we enter the name, we get the definition

\[
\text{area}(r) \pi \times r^2
\]

We can look at the internal structure

\[
\text{as.list(as.list(area))(2)}
\]

R Specifics

Internal structure

\[
\text{as.list(as.list(area))(2)}
\]

Note: operators are polymorphic even for pen and paper.

ToDo 1: Blackboard graphics: attributed variables and operators

ToDo 2: Blackboard graphics: attributed variables and operators

Pen and paper arithmetics

Source: the program code we enter

This may be a valid program or it may contain errors, it may be incomplete.

Intermediate code: a graph is a convenient representation

Each node represents an operation to be performed. The terminating nodes are special: their action is to fetch some data, resulting in a value. Each node can try to evaluate. If it succeeds, the result is again a value; if it does not succeed, we have an exception.

ToDo 1: Blackboard graphics: attributed variables and operators
“Intermediate code” is the level to think of a program. In general, the most appropriate representation of a program is a graph that has to be worked through. Code optimisation amounts to replacing the graph by an equivalent graph, with a more efficient execution.

R specifics

Be aware: there is overhead by implicit type conversion or vectorization.

Vectorize, where sensible.

But: there are pitfalls ahead.
**Performance Measurement: Time**

*First clause of control theory*

*Without measurement, there is no control…*

---

**Function Arguments and Environments**

Environments are special objects in R. By default, arguments are passed to functions by value in R, that is a copy of generated. As an exception, environments are not copied. They are passed by reference. Wrapping information in an environment is an important possibility to optimize access to large data sets.

---

**Function Arguments and Promises**

R uses a lazy evaluation for arguments passed to functions. When a function is called, the arguments are passed as parsed expression, together with a reference to the environment to be used for their evaluation. Evaluation only takes place when the value of the argument is actually needed.

See *R Optimisation: Function Arguments*. 

---

**Performance Measurement: Time**

**system.time**

**CPU Time Used**

*Description*

Return CPU (and other) times that expr used.

*Usage*

```
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
```

*Arguments*

- `expr` Valid R expression to be timed.
- `gcFirst` Logical - should a garbage collection be performed immediately before the timing? Default is `TRUE`. 

*Details*

`system.time` calls the function `proc.time`, evaluates `expr`, and then calls `proc.time` once more, returning the difference between the two `proc.time` calls. `unix.time` is an alias of `system.time`, for compatibility with S.

Timings of evaluations of the same expression can vary considerably depending on whether the evaluation triggers a garbage collection. When `gcFirst` is `TRUE` a garbage collection (`gc`) will be performed immediately before the evaluation of `expr`. This will usually produce more consistent timings.
Timing Functions

Performance Measurement: Time

```r
system.time(expr, gcFirst = TRUE)
unix.time(expr, gcFirst = TRUE)
proc.time()
Sys.sleep(time)
gc.time(on = TRUE)
```

Internal steps:
- `vec <- c(vec,i)`

Case Study: Sequential Numbers

Timing: Sequential Numbers

Simple example

```r
> system.time({vec <- numeric(n); for (i in 1:n) vec[i] <- i})
```

Example with pre-allocation

```r
> system.time({vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)})
```

Timing: Sequential Numbers

Example:

```r
> benchmark(1:10^4, log(1:10^4), sin(1:10^4),
            columnnames=c('test', 'elapsed', 'replications', 'relative'))
```

Timing Functions

Performance Measurement: Time

Timer Resolution

The timing resolution is limited.

Working range on all nowadays system should be 10ms or better.

To find e.g. the number of iterations to be above the resolution
threshold, use

```r
library(iterators)
timeout(iterable, time)
```

For example: See how high we can count in a tenth of a second

```r
length(as.list(timeout(icount(), 0.1)))
```

For example: See how high we can count in a tenth of a second

Performance Measurement: Time

Performance Measurement: Time

Timing: Sequential Numbers

Simple example

```r
> n<- 100
> system.time({vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)})
```

Example with pre-allocation

```r
> system.time({vec <- numeric(n); for (i in 1:n) vec[i] <- i})
```

Vectorized

```r
> system.time({vec <- numeric(n); for (i in 1:n) vec[i] <- i})
```

Performance Measurement: Time

Performance Measurement: Time

Case Study: Sequential Numbers

Vectorization

```r
vec <- (vec,i)
```

Internal steps:
- copy argument vec,i
- check type. Adjust type if necessary
- calculate requested length length(vec)+length(i)
- allocate space of requested length
- copy vec to result space
- append i to result space
- return result as new value for vec

Note: the storage space previously allocated for vec is now free.

R Built-in Optimisations

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time

Computing Background

Performance Measurement: Time

Performance Measurement: Time
Case Study: Sequential Numbers

vec <- c(vec, i)

- Note: repeated allocation and copy steps are time consuming.
- Note: the storage space previously allocated for vec is now free.
- Bad news: this slot size is just two items less than the slot needed in the next step.
- It cannot be re-used now.
- Garbage collection is necessary eventually.

Avoid Growing Variables

If you must use growing variables: remember, R stores variables in 'column first' order. Use memory preallocation, if possible. If you must use growing variables: remember, R stores variables in 'column first' order. Use memory preallocation, if possible. If you must use growing variables: remember, R stores variables in 'column first' order. Use memory preallocation, if possible. If you must use growing variables: remember, R stores variables in 'column first' order. Use memory preallocation, if possible. If a variable is not used, consider marking it for release using gc() before critical program segments.

Avoid Storage Type Conversion

Avoid explicit or implicit type conversions. Keep it as simple as possible, but not simpler. Prefer using a vector over using a list. Prefer using a matrix over using a data frame.

But of course if a list or a data frame is needed, don’t use a simpler type.
Performance Measurement: Memory

**Memory Usage**

Memory usage is dynamic. Memory may be allocated or released during evaluation, for example to store intermediate results.

In R: Information about memory usage is provided as a side effect of garbage calculation.

Part of the memory management is garbage collection, and the time used for garbage collection may be a major concern.

Garbage collection is a second process which is influenced, but not controlled by your program process.

system.time() does not include specific information about the time for garbage collection.

---

**Exercise**

Use gc.time() to extend system.time() to provide the time spent on garbage collection during the process.

Provide a modified function as system.gc.time().

Repeat the previous exercise.

---

**Garbage collection levels**

There are three levels of collections.

- Level 0 collects only the youngest generation.
- Level 1 collects the two youngest generations.
- Level 2 collects all generations.

After 20 level-0 collections the next collection is at level 1, and after 5 level-1 collections at level 2.

Further, if a level-n collection fails to provide 20% free space (for each of nodes and the vector heap), the next collection will be at level n + 1.

(The R-level function gc() performs a level-2 collection.)

---

**Memory Usage: gc()**

- **Usage**
  
  gc(override = gcinfo(TRUE), reset=FALSE)
  gcinfo(override)
  system.time()

- **Arguments**
  
  - verbose: logical; if TRUE, the garbage collection prints statistics about cons cells and the space allocated for vectors.
  - reset: logical; if TRUE the values for maximum space used are reset to the current values.

- **Details**
  
  A call of gc causes a garbage collection to take place. gcinfo sets a flag so that automatic collection is either silent (verbose=FALSE) or prints memory usage statistics (verbose=TRUE).

A call of gc() causes a garbage collection to take place. This will also take place automatically without user intervention, and the primary purpose of calling gc is for the report on memory usage.

However, it can be useful to call gc after a large object has been removed, as this may prompt R to return memory to the operating system.

R allocates space for vectors in multiples of 8 bytes: hence the report of "Vcalls", a relict of an earlier allocator (that used a vector heap).

---

When gcinfo(TRUE) is in force, messages are sent to the message connection at each garbage collection of the form:

Garbage collection 12 = 10+0+2 (level 0) ...
6.4 Byths of cons cells used (58%)
2.0 Byths of vectors used (32%)

Here the last two lines give the current memory usage rounded up to the next 0.1Mb and as a percentage of the current trigger value. The first line gives a breakdown of the number of garbage collections at various levels (for an explanation see the 'R Internals' manual).
Special Libraries, and Other Ways to Avoid the Problems

THINK!
Consider avoiding the problem.
- Think! If you can come up with a closed solution, you may avoid all (or most) computation.
- Consider changing the hardware. If there is old cpd around with only a part of the computing power of an up-to-date cpu, but unused most of the time: running your computation here may save a lot.
- Consider local cooperation. You can share computing power. For example, XGrid is a solution that is readily available. http://developer.apple.com/hardware/hpc/xgrid_intro.html.
- Make use of regional resources. For example, BW Grid provides access to more than 1000 CPUs. http://www.bw-grid.de/allgemeine-informationen/hardware/

Instrumenting the code
```
debug(fun, text="", condition=NULL)
```
A call to trace allows you to insert debugging code (e.g., a call to browser or recover) at chosen places in any function.
Do not forget the global possibility: `options(error=browser)`

Vectorization

**Vectorized Functions**
The basic functions in R are vectorized.
In some instances, vectorization is required, for example for functions passed as argument to integrators and optimisers.
A function can be vectorized using `Vectorize()`. `Vectorize()` wraps a call using `mapply()`.
Vectorization by itself does not bring any performance advantage. In particular, `Vectorize()` only puts a loop around the code of a function. Formal vectorization must be accompanied by an access optimisation.

R Built-in Optimisations

**Optimisation**

**Vectorization**

The Rcpp package provides a C++ library which facilitates the integration of R and C++

Rcpp: R-Forge Development Page
https://r-forge.r-project.org/projects/rcpp/

References

CRAN task view
http://cran.r-project.org/web/views/HighPerformanceComputing.html

Bigmemory: R-Forge Development Page
https://r-forge.r-project.org/R/?group_id=556

Large objects for R: R-Forge Development Page
https://r-forge.r-project.org/R/?group_id=483

See also
http://developer.r-project.org/Sparse.html
Classical Optimisation

Control-flow analysis

Basis: Control-flow graph
Basic block: must enter at beginning, exit only at end (no branches)

Local Optimisation - Classical Techniques

- Constant Folding: Evaluation of Constants at "Compile Time"
- Constant Propagation: Replace Variables by Constants if Value Does Not Change
- Algebraic Simplification And Re-association
- Operator Strength Reduction
- Copy Propagation
- Dead Code Elimination
- Common Subexpression Elimination
- Loop Uncoiling
- Global Optimisations and Data Flow Analysis
- Code Motion
- Machine Optimisation
- Register/Memory Allocation
- Instruction Scheduling
- Peephole Optimisation

Classical Techniques: Constant folding

Main effect: constant evaluation made at compile time, not at run time.

\[ 1 + x + 2 \to x + 3 \]

Constants may be hidden, such as \( n \equiv n x \times n y \).

Classical Techniques: Algebraic simplification and re-association

Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions. Re-association refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimisations such as constant-folding or loop-invariant code motion.

The most obvious of these are the optimisations that can remove useless instructions entirely via algebraic identities.

Operator Strength Reduction

Operator strength reduction replaces an operator by a “less expensive” one.

Copy Propagation

This optimisation is similar to constant propagation, but generalised to non-constant values. If we have an assignment \( a = b \) in our instruction stream, we can replace later occurrences of \( a \) with \( b \) (assuming there are no changes to either variable in-between).

This may be a particularly valuable optimisation since it may be able to eliminate a large number of instructions that only serve to copy values from one variable to another.
Dead Code Elimination

If an instruction’s result is never used, the instruction is considered “dead” and can be removed from the instruction stream.

Dead code frequently results from previous editing operations.

Loop Uncoiling

Loop uncoiling, or loop unrollment replaces two or more loop iterations by explicit statements.

Code Motion

Code motion unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation. The most common form of code motion is loop-invariant code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop.

Register and Memory Allocation

Registers are the fastest kind of memory available, but as a resource, they can be scarce. The problem is how to minimise traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches.

Global Optimisations, Data-Flow Analysis

The additional analysis the optimiser must do to perform optimisations across basic blocks is called data-flow analysis.

Machine Optimisations

In this pass, specific machines features (specialised instructions, hardware pipeline abilities, register details) are taken into account to produce code optimised for this particular architecture.

Instruction Scheduling

Another extremely important optimisation of the final code generator is instruction scheduling. Because many machines have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions.
Local Optimisation

Peephole Optimisations

Peephole optimisation is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent code improvements. For example, peephole optimisations might include elimination of multiplication by 1, elimination of load of load of a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect.

Local Optimisation

Optimisation in R

Support tools for most standard techniques are available in library(compiler).

Some are supported in the byte code compiler (under development).

Function Arguments

Function Arguments in R

- As a rule, parameters are passed by value. (i.e. copied. Beware! This means memory allocation and copying.)
- The main exception are environments. These are not copied, but passed as a reference. You can make use of this and pass use information efficiently by just passing its name and the environment to look it up.
- R uses lazy evaluation: if a value needs to be calculated, the evaluated expression needed to calculate it is passed, together with the environment where the evaluation should take place. This is called a promise.
- R can pass a value by reference, if it can prove that it is unchanged during evaluation. This is used extensively for .Internal functions.

Function Arguments

Internal Functions

If an .Internal function can be used, prefer it over other solutions. It may be cheaper.

Do not spend time optimising something .Internal. (And do not spend time optimising it. This is something already in the optimised libraries.)

If only information from a variable descriptor is used by a function, assume the function is implemented as .Internal.

For example, length() is .Internal (and cheap to call).

Function Arguments

Parameter Passing Exceptions

By default, function arguments are passed by value.

There are few exceptions defined by in the language:

- environments
- promises
- ...
Function Arguments

Example: Cache for Fibonacci

Creating a persistent local environment for a function to implement a persistent memory:

```r
fibonacci <- local({
  memo <- c(1, 1, rep(NA, 100))
  f <- function(x) {
    if(x <= 0) return(0)
    if(x < 0) return(memo[x])
    if(length(memo) > x) stop("x too big for implementation")
    memo[x] <<- f(x-2) + f(x-1)
    ans <- f(x-2) + f(x-1)
    memo[x] <<- ans
    ans
  }
})
```

From P. Burns, R Inferno, Circle 6.1

Compilation

Byte Code

The byte code compiler needs hints about the function interface used for external access.

These hints are part of the declaration of a name space that must be included with the package. (See Writing R extensions)

Byte Code

To generate a byte-compiled version of an individual function, use `library(compiler)` and function `cmpfun()`.

Example

As a case study, we use an algorithm to calculate the shorth functional, a non-parametric functional for the analysis of one-dimensional data. The complexity of the algorithm keeps time consumption annoying, yet still in a range allowing for a scalable variation of sample sizes.

Shorth length \( S_\alpha(x) \) at \( x \) covering \( \alpha \): minimum length of an interval containing \( x \) and covering a proportion \( \alpha \) of the data.

\[
S_\alpha(x) = \min \{ |I| : x \in I, P(I) \geq \alpha \}
\]

Sh orth plot: Plot \( x \mapsto S_\alpha(x) \) for a collection of coverages \( \alpha \).

For the implementation, see <http://lshorth.r-forge.r-project.org/>.

Case Study: Byte Code

No byte-code

<table>
<thead>
<tr>
<th>n</th>
<th>user time</th>
<th>system time</th>
<th>elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.022</td>
<td>0.002</td>
<td>0.027</td>
</tr>
<tr>
<td>100</td>
<td>0.124</td>
<td>0.008</td>
<td>0.134</td>
</tr>
<tr>
<td>1000</td>
<td>2.013</td>
<td>0.028</td>
<td>2.237</td>
</tr>
</tbody>
</table>

Moderate until up to 50% gain in total job turnaround time. Note: base libraries have already been byte-compiled.

Byte-code

<table>
<thead>
<tr>
<th>n</th>
<th>user time</th>
<th>system time</th>
<th>elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.017</td>
<td>0.002</td>
<td>0.021</td>
</tr>
<tr>
<td>100</td>
<td>0.114</td>
<td>0.015</td>
<td>0.131</td>
</tr>
<tr>
<td>1000</td>
<td>1.335</td>
<td>0.030</td>
<td>1.367</td>
</tr>
</tbody>
</table>

Good news: pre-compilation comes for free for packages (if implemented on your implementation).
A Case Study: Melbourne Data

Melbourne Temperature Data

This example will be used for the remainder of this section.

```r
library(lshorth)
melbourne3 <- data.frame(read.csv("/data/melbourne/temp v pressure 3 hourly intervals.csv"))
dt<-as.POSIXlt(melbourne3[,1])
melbourne3 <- data.frame(read.csv("/data/melbourne/temp v pressure 3 hourly intervals.csv"))
```

```r
melbourne <- melbourne3[dt$hour==15,] # 15h data
```

```r
dt<-as.POSIXlt(melbourne3[,1])#lenght 9??
```

```r
melbourne3 <- data.frame(read.csv("/data/melbourne/temp v pressure 3 hourly intervals.csv"))
```

```r
library(lshorth)
```

This example will be used for the remainder of this section.

```r
temp <- quantile(melbourne15h[,2], probs=seq(0,1,0.125)) # 1.9
```

```r
gress <- quantile(melbourne15h[,3], probs=seq(0,1,0.125))
```

```r
plotcond3 <- function(tlim,main=NULL,...){
  tlim <- seq(tlim)
  main <- paste("T: ", tlim[1], "...","C p: ", plim, "... ", "hpa", sep=""), legend=NULL)
  plotcond3(tlim, main=main, cex=1.5, ...)}
```

```r
bigplot <- function(tlim,plim=c(0,qpress[2]), legend=NULL){
  oldpar <- par(mfrow=c(3,3),
  bigplot <- function(tlim,plim=c(0,qpress[2]), legend=NULL){
  oldpar <- par(mfrow=c(3,3),
  par(oldpar)...

```r
Melbourne Temperature Data

```r
plotcond <- function(tlim,plim=c(NULL,...)){
  diffT <- incond[,4]-incond[,2]
  diffT <- diffT_FINITE
  ist <- lshorth(diffT, probs=c(0.125,0.25,0.5,0.75, 0.875),plot=FALSE)
  if (is.null(main)){main=paste("T : ", tlim,"p : ", plim, "... " , "hpa", sep=""))
  plot(ls, frame.plot=FALSE, main=main, cex=1.5, ...)
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```

```r
plott <- system.time(bigplot())
```
Profiling Time: SummaryRprof()

summaryRprof

Summarise Output of R Sampling Profiler

Description

Summarise the output of the Rprof function to show the amount of time used by different R functions.

Usage

summaryRprof(filename = "Rprof.out", chunksize = 5000, memory=c("none","both","tseries","stats"), index=2, diff=TRUE, exclude=NULL)

Arguments

filename Name of a file produced by Rprof()
chunksize Number of lines to read at a time
memory Summaries for memory information. See 'Details' below
index How to summarize the stack trace for memory information. See 'Details' below.
diff If TRUE memory summaries use change in memory rather than current memory
exclude Functions to exclude when summarizing the stack trace for memory summaries

Details

This function provides the analysis code for Rprof files used by R CMD Rprof.
As the profiling output file could be larger than available memory, it is read in blocks of chunksize lines. Increasing chunksize will make the function run faster if sufficient memory is available.

When called with memory=profiling = TRUE, the profiler writes information on three aspects of memory use: vector memory in small blocks on the R heap, vector memory in large blocks (from malloc), memory in nodes on the R heap. It also records the number of calls to the internal function duplicate() when arguments need to be copied. Note that the profiler does not track which function actually allocated the memory.

With memory = "both" the change in total memory (truncated at zero) is reported in addition to timing data.

Profiling Time: summaryRprof()

Rprof("lshorth_prof.txt", interval=0.002)
bigplot()
Rprof(NULL)
summaryRprof("lshorth_prof.txt")

Profiling Time: SummaryRprof()

summaryRprof([lshorth_prof.txt.txt])

$sampling.time

> summaryRprof("lshorth_prof.txt")

$sby.self

$by.total

Profiling Time: $by.total

Profiling Time: SummaryRprof() II

"[.data.frame" 0.004 1.57 0.004 1.57
"[.factor" 0.018 7.09 0.020 7.87
"title" 0.018 7.09 0.018 7.09
"which.min" 0.004 1.57 0.004 1.57
"plot" 0.004 1.57 0.004 1.57
"plotcond" 0.002 0.79 0.002 0.79
"lines" 0.004 1.57 0.004 1.57
"which" 0.002 0.79 0.002 0.79
"Axis" 0.002 0.79 0.002 0.79
"Axis.default" 0.012 4.60 0.012 4.60
"axis" 0.002 0.79 0.002 0.79
"plot.new" 0.004 1.57 0.004 1.57
"plotcond" 0.002 0.79 0.002 0.79
"plot.xy" 0.002 0.79 0.002 0.79
"lines" 0.004 1.57 0.004 1.57
"which" 0.002 0.79 0.002 0.79
"Axis" 0.002 0.79 0.002 0.79
"Axis.default" 0.012 4.60 0.012 4.60
"axis" 0.002 0.79 0.002 0.79
"plot.new" 0.004 1.57 0.004 1.57
"plotcond" 0.002 0.79 0.002 0.79
"plot.xy" 0.002 0.79 0.002 0.79
"lines" 0.004 1.57 0.004 1.57
"which" 0.002 0.79 0.002 0.79
"Axis" 0.002 0.79 0.002 0.79
"Axis.default" 0.012 4.60 0.012 4.60
"axis" 0.002 0.79 0.002 0.79
"plot.new" 0.004 1.57 0.004 1.57
"plotcond" 0.002 0.79 0.002 0.79
"plot.xy" 0.002 0.79 0.002 0.79
"lines" 0.004 1.57 0.004 1.57
"which" 0.002 0.79 0.002 0.79
"Axis" 0.002 0.79 0.002 0.79
"Axis.default" 0.012 4.60 0.012 4.60
"axis" 0.002 0.79 0.002 0.79
"plot.new" 0.004 1.57 0.004 1.57
"plotcond" 0.002 0.79 0.002 0.79
"plot.xy" 0.002 0.79 0.002 0.79
"lines" 0.004 1.57 0.004 1.57
"which" 0.002 0.79 0.002 0.79
...
Profiling Time

Using Rprof

The profile is easily scattered with garbage, and there is no semantic filter.

Use `Rprof()` selectively for zones of interest, do several runs using the append option.

```
Using Rprof
```

```
Memory is attributed to the function active at the end of the sampling interval.
This may be misleading.
```

```
Profiling Functions

Rprof(filename = "Rprof.out",
append = FALSE,
memory.profiling=FALSE)

summaryRprof(filename = "Rprof.out",
chunksize = 5000,
memory=c("none","both","tseries","stats"),
index=2, diff=TRUE, exclude=NULL)
```

Profiling Time I

```
> summaryRprof("lshorth_profm.txt", memory="both")

$by.self

self.time self.pct total.time total.pct mem.total

"axis" 0.092 39.32 0.092 39.32 7.7
"lapply" 0.28 11.97 0.044 18.80 24.1
"[.data.frame" 0.024 10.26 0.056 23.93 7.6
"title" 0.018 7.69 0.018 7.69 2.5
"[.factor" 0.018 7.69 0.018 7.69 0.1
"lshorth" 0.044 18.80 24.1 0.028 11.97
"[.data.frame" 0.024 10.26 0.056 23.93 7.6
"[.factor" 0.018 7.69 0.018 7.69 2.5
"title" 0.018 7.69 0.018 7.69 0.1
"which.min" 0.012 5.13 0.012 5.13 2.6
"<" 0.008 3.42 0.008 3.42 0.0
"[" 0.004 1.71 0.004 1.71 0.0
"<=" 0.002 0.85 0.002 0.85 1.4
"sys.call" 0.002 0.85 0.002 0.85 0.3

$by.total

```

Profiling Time II

```
total.time total.pct mem.total self.time self.pct

"bigplot" 0.234 100.00 40.9 0.000 0.00
"plotcond3" 0.234 100.00 40.9 0.000 0.00
"plotcond" 0.218 93.16 39.7 0.000 0.00
"plot.lshorth" 0.134 57.26 9.2 0.000 0.00
"plot" 0.134 57.26 9.2 0.000 0.00
"axis" 0.092 39.32 7.7 0.092 39.32
"[.data.frame" 0.056 23.93 7.6 0.024 10.26
"[.factor" 0.018 7.69 2.5 0.018 7.69
"title" 0.018 7.69 0.1 0.018 7.69
"which.min" 0.012 5.13 0.4 0.012 5.13
"[" 0.008 3.42 0.0 0.008 3.42
"<=" 0.004 1.71 1.0 0.004 1.71
"<" 0.004 1.71 6.1 0.004 1.71

$sample.interval

[1] 0.002

$sampling.time

[1] 0.234

Profiling Memory

Profiling Memory: `Rprofmem()`

```
Rprofmem(filename = "Rprofmem.out",
append = FALSE,
memory.profiling=FALSE)
```

Profiling Functions

Rprof(filename = "Rprof.out",
append = FALSE,
memory.profiling=FALSE)

summaryRprof(filename = "Rprof.out",
chunksize = 5000,
memory=c("none","both","tseries","stats"),
index=2, diff=TRUE, exclude=NULL)

Profiling Memory Use

Description

Enable or disable reporting of memory allocation in R.

Usage

```
Rprofmem(filename = "Rprofmem.out",
append = FALSE,
threshold = 0)
```
Profiling Memory: Rprofmem()

Usage

Rprofmem(filename = "Rprofmem.out", append = FALSE, threshold = 0)

Arguments

filename

The file to be used for recording the memory allocations. Set to NULL or "" to disable reporting.

append

logical: should the file be over-written or appended to?

threshold

numeric: allocations on R's "large vector" heap larger than this number of bytes will be reported.

Details

Enabling profiling automatically disables any existing profiling to another or the same file.

Profiling writes the call stack to the specified file every time malloc is called to allocate a large vector object or to allocate a page of memory for small objects. The size of a page of memory and the size above which malloc is used for vectors are compile-time constants, by default 2000 and 128 bytes respectively. The profiler tracks allocations, some of which will be to previously used memory and will not increase the total memory use of R.

Note: at sampling time, only the size of memory requested is known. There is no variable name associated to it.

To identify the objects, given their size, you can use a code snippet as:

```R
object.size(get(x, envir = baseenv()))
```

A summary() method for Rprofmem() is still under construction.

Profiling Memory: Rprofmem()

A memory threshold can be installed to avoid uninformative events.

Rprofmem("lshorth_profmem1024.txt", threshold=1024)
bigplot()

Rprofmem(NULL)

bigplot()

Rprofmem("lshorth_profmem.txt")

Profiling Memory: Rprofmem()

II

> noquote(readLines("lshorth_profmem1024.txt", n=30))

Duplicates

The internal function duplicate is called when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

It is a major cause of hard-to-predict memory use in R. traceemem() can be used to track the creation of duplicates.
Profiling Memory: `tracemem()`

**tracemem**  
*Trace Copying of Objects*

**Description**
This function marks an object so that a message is printed whenever the internal function `duplicate` is called. This happens when two objects share the same memory and one of them is modified. It is a major cause of hard-to-predict memory use in R.

**Usage**

```r
tracemem(x)
untracemem(x)
retracemem(x, previous = NULL)
```

**Arguments**

- `x` An R object, not a function or environment or `NULL`.
- `previous` A value as returned by `tracemem` or `retracemem`.

**Details**
This functionality is optional, determined at compilation, because it makes R run a little more slowly even when no objects are being traced. `tracemem` and `untracemem` give errors when R is not compiled with memory profiling; `retracemem` does not (so it can be left in code during development).

When an object is traced any copying of the object by the C function `duplicate` or by arithmetic or mathematical operations produces a message to standard output. The message consists of the string `tracemem`, the identifying strings for the object being copied and the new object being created, and a stack trace showing where the duplication occurred. `retracemem()` is used to indicate that a variable should be considered a copy of a previous variable (e.g. after subscripting).

It is not possible to trace functions, as this would conflict with `trace` and it is not useful to trace `NULL`, environments, promises, weak references, or external pointer objects, as these are not duplicated.

These functions are primitive.