Introductory Computer Programming Assignment 1

Soumya Mukherjee

31 October 2019

1. Recursive Version of Insertion Sort :

- The basic problem:
  - Input: A sequence of numbers $(a_1, a_2, ..., a_n)$
  - Desired output: A permutation of the input, $(b_1, b_2, ..., b_n)$ such that $b_1 \leq b_2 \leq ... \leq b_n$
- The a_i-s are known as *keys*.
- For sorting, we usually need a simple data structure known as an *array*:
  - An array $A[1, ..., n]$ of length n is a sequence of length n.
  - The i-th element of an array A is denoted by $A[i]$
  - Each $A[i]$ acts as a *variable*, that is, we can assign values to it, and query its current value
  - The sub-array with indices i to j (inclusive) is often indicated by $A[i, ..., j]$

Insertion sort:

- *Insertion sort* is a simple and intuitive sorting algorithm
- Basic idea of Insertion sort(Iterative version):
  1. If the array is of length 1 it is trivially sorted.
  2. Else choose the first element of the input array.
  3. Compare the second element of the array with the first. If the first element is greater than the second, then swap them else leave their positions in the array unchanged.
  4. Having sorted the first two elements, insert the third element into its correct sorted position considering only the first three elements of the array now.
  5. Continue in this manner until the last element of the array is inserted into its correct sorted position in the array.
- Basic idea of Insertion sort (Recursive version):
  1. If the array is of length 1 it is trivially sorted.
  2. Else divide the array into two parts - the first part containing the first n-1 elements of the array in the given order and the second part consisting of the $n - th$ element of the array, n being the length of the array.
  3. If the first part is of length 1 then it is trivially sorted.Then compare the second part of the array i.e. the last element of the array, with it and swap them if they are not in nondecreasing order.
  4. If the first part has length greater than 1, apply this algorithm on these n-1 elements to sort them i.e. use this algorithm recursively. After they have been sorted, insert the last element of the original n-element input array into its correct sorted position.

Pseudo Code for recursive version of insertion sort:

Here the input is an already-constructed array `A` along with the no. of elements of the array starting from the first that we want to sort, given by 'n'. If we want to sort the whole array then 'n' is the length of the array.

Pseudo Code for the algorithm:

```
recur.ins.sort(A, n)
```

```
if ( n < 2 ) return (A)
else {
   A = recur.ins.sort(A, n-1)
   key = A[n]
   j = n - 1
   while (j > 0 and A[j] > key)
   {
      A[j+1] = A[j]
      j = j - 1
   }
   A[j+1] = key
   return (A)
}
```

2. Correctness of the algorithm:

We shall prove the correctness of the algorithm by recursion invariant.

Statement

> At the beginning of each recursion to sort the fitst n-j+1 elements of the array (for any particular value of j between 1 and n-1), The first $n-j$ elements in $A[1, ..., n-j]$ are the same as the first $n-j$ elements originally in the array, but they are now in sorted order, where n is the length of the input array.

Initialisation

- For $j = n-1$, the Array $A[1, 2, ..., n-j]$ has exactly one element which is trivially sorted.
- Hence the above statement holds for $j = n-1$

Maintenace

- Suppose the recursion invariant is true for index j.
- At the beginning of the recursion to sort the first n-j+1 elements of the array (for any particular value of j), $A[1, 2, ..., n-j]$ is sorted.
- While loop within each recursion works by
  - comparing $key = A[n-j+1]$ with $A[n-j], A[n-j-1], ..., A[1]$ (following this order)
  - moving them one position to the right , until the correct position of key is found
- Evidently, this while loop must terminate in atmost $n-j$ steps.
- At the end $A[1, 2, ..., n-j+1]$ is a sorted version of original array $A[1, 2, .., n-j+1]$
- Thus, the recursion invariant is true for index $j-1$.

Termination

- Each call of the function decreases the length of sub-array $A[1, ..., n-j]$ by 1 ,
- The function stops recursive calls when the length of sub-array $A[1, ..., n-j] = 1$ i.e. when $j = 1$.

- Therefore the recursion terminates.
  - Hence the algorithm is correct.

3. Average case Runtime:

- When inserting $A[n]$ in its correct position , it assumes that $A[1, 2, .., n-1]$ is already sorted for any value of n.
- So, in the best case scenario the while loop will have to compare $A[j] < key = A[n]$ only once, whereas in the worst case scenario the while will loop have to do the same $n-1$ times.
- So for randomised input, on an average the while loop will have to do the comparison $\frac{n}{2}$ times (As, the number of comparisons has distribution $Uniform\{1,2,\ldots,n\text{-}1\}$)
- Hence the recursive relation for Average case runtime $T(n)$ is given by:

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(n-1) + O(\frac{n}{2}) & n > 1 \end{cases}$$

- Therefore Average case runtime is given by:

$$\begin{aligned} T(n) &= T(n-1) + O(\frac{n}{2}) \\ &= O(\frac{n}{2}) + O(\frac{n-1}{2}) + \cdots + O(\frac{2}{2}) + T(1) \\ &= O(\frac{n(n+1)}{4}) \\ &= O(n^2) \end{aligned} \tag{1}$$

4. Implementations of the algorithm in R and Rcpp:

Insertion Sort in R:

1. Iterative Version of Insertion Sort:

- The iterative insertion sort is implemented in R as follows:

```
iter.ins.sort = function(A){
  if (length(A) < 2) return(A)
  for (j in 2:length(A)) {
    key = A[j]
    i = j-1
    while (i > 0 && A[i] > key) {
      A[i+1] = A[i]
      i = i-1
    }
    A[i+1] = key
  }
  return(A)
}
```

2. Recursive Version of Insertion Sort:

- Recursive version of insertion sort is implemented in R as follows :

```
recur.ins.sort = function (A, n = length(A) ){
  if ( n < 2 ) return(A)
  else{
```

```
    A = recur.ins.sort(A, n-1)
    key = A[n]
    j = n-1
    while (j > 0 && A[j] > key ) {
      A[j+1] = A[j]
      j = j-1
    }
    A[j+1] = key
    return(A)
  }
}
```

Insertion Sort using Rcpp:

1. Iterative Version of Insertion Sort using Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector iter_ins_sort_rcpp(NumericVector A)
{
    int i, n = A.size();
    double key;
  for (int j = 1; j < n; j++)
  {
      key = A[j];
      i = j - 1;
      while (i > -1 && A[i] > key)
      {
          A[i+1] = A[i];
          i = i - 1;
          }
      A[i+1] = key;
    }
    return A;
}
```

2. Recursive Version of Insertion Sort using Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector recur_ins_sort_Rcpp(NumericVector A, int n )
{
    int j;
    double key;
  if (n < 1)
  {
    return (A);
  }
  else
  {
```

```
    A = recur_ins_sort_Rcpp( A, n-1 );
    key = A[n-1];
    j = n-2;
    while(j > -1 && A[j] > key)
    {
      A[j+1] = A[j];
      j = j-1;
    }
    A[j+1] = key;
  }
  return (A);
}
```

Some sample implementations of these programs and the inbuilt sort function in R:

```
A = sample(1:1000, 50, replace = TRUE)
A
```

```
##  [1]  74  33 219 938 323  17 589 482 324 878 393 770 189  93 597 541 813
## [18] 216 559  50 414  59 344 542 973  22  80 998 432 729 533 483 723 336
## [35] 959 486 130  22 884 237 941 969 293 254 683 819 824 795 988 354
```

```
sort(A)
```

```
##  [1]  17  22  22  33  50  59  74  80  93 130 189 216 219 237 254 293 323
## [18] 324 336 344 354 393 414 432 482 483 486 533 541 542 559 589 597 683
## [35] 723 729 770 795 813 819 824 878 884 938 941 959 969 973 988 998
```

```
A = sample(1:1000, 50, replace = TRUE)
A
```

```
##  [1] 508 632 735 242 347 969 696 561 255 598 215 934 161 623 266 782 118
## [18]  13 508 969 520  19 179 758 442 275 125 682 260 676 787 320 127 337
## [35] 633  25 934 516 913 480 631 567 194 787 365 872 717  80 360 236
```

```
iter.ins.sort(A)
```

```
##  [1]  13  19  25  80 118 125 127 161 179 194 215 236 242 255 260 266 275
## [18] 320 337 347 360 365 442 480 508 508 516 520 561 567 598 623 631 632
## [35] 633 676 682 696 717 735 758 782 787 787 872 913 934 934 969 969
```

```
A = sample(1:1000, 50, replace = TRUE)
A
```

```
##  [1] 989  24 895 148 509 927 572 600 218 912 680  57 388 376 657 575 436
## [18]  17 247 968 466 507 685  54 576 400 148 991 481 995 707 913 507 797
## [35] 672 161 321 246 908 833 331 846 747 448 182 515 612  83 812 240
```

```
recur.ins.sort(A, length(A))
```

```
##  [1]  17  24  54  57  83 148 148 161 182 218 240 246 247 321 331 376 388
## [18] 400 436 448 466 481 507 507 509 515 572 575 576 600 612 657 672 680
## [35] 685 707 747 797 812 833 846 895 908 912 913 927 968 989 991 995
```

```
A <- sample(1:1000, 50, replace = TRUE)
A
```

```
##  [1] 942   1 283 265 411 326 329 778 630 502  43 916 836 949 378  21 804
## [18] 274  85 272 873 953 804 231 262 416 870 862  20 177 404 105 726 917
```

```
## [35] 652  74 832 674 788 465 524 240 351 417 979 456 914 491 845  65
```

```
iter_ins_sort_rcpp(A)
```

```
##  [1]   1  20  21  43  65  74  85 105 177 231 240 262 265 272 274 283 326
## [18] 329 351 378 404 411 416 417 456 465 491 502 524 630 652 674 726 778
## [35] 788 804 804 832 836 845 862 870 873 914 916 917 942 949 953 979
```

```
A <- sample(1:1000, 50, replace = TRUE)
A
```

```
##  [1] 181 793 523 177 975 881 647   2 475 925 679 619 658 367 697 876 114
## [18] 721 971 640 758 470 591 874 992 190 545 641  24 981  86 220  13 601
## [35] 276 221 320 675  48  82 942 230 417 925  90 397 328 513 750 495
```

```
recur_ins_sort_Rcpp(A, length(A))
```

```
##  [1]   2  13  24  48  82  86  90 114 177 181 190 220 221 230 276 320 328
## [18] 367 397 417 470 475 495 513 523 545 591 601 619 640 641 647 658 675
## [35] 679 697 721 750 758 793 874 876 881 925 925 942 971 975 981 992
```

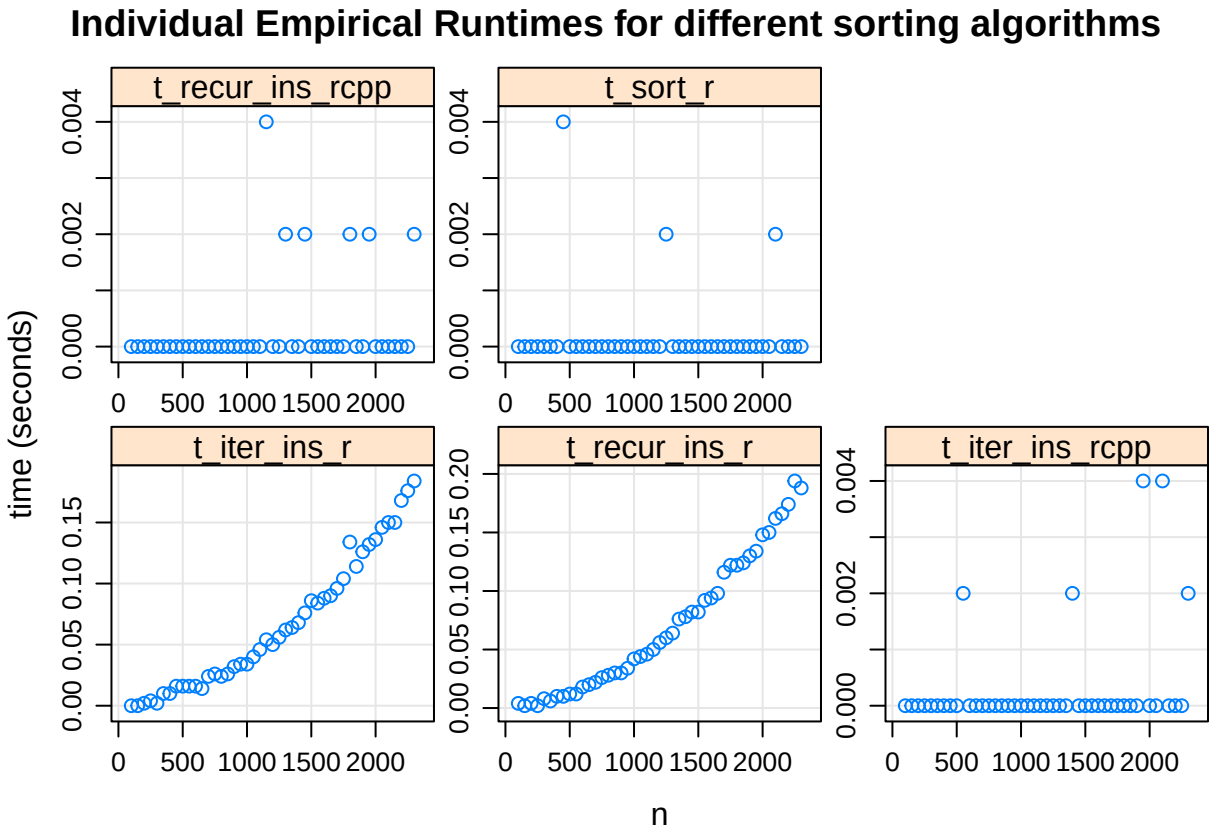
5. Empirical Runtime Comparison:

- We define a function `timeSort` which will calculate the average runtime required to sort an array.
- For the recursive version of the programs we define two other functions namely `sin.recur.ins.r` and `sin.recur.ins.rcpp` for R and Rcpp programs respectively , so that these functions would require only asingle argument instead of two arguments and consequently we will be able to run `timeSort` on them, otherwise obtsining and compaing empirical runtimes takes a very long time using for loops or some similar variants.
- With the large length of input array (above 2300) the recursive version implementations in both R and Rcpp , get teminated abruptly due to “Stack error” or “deep nested infinite recursions” .

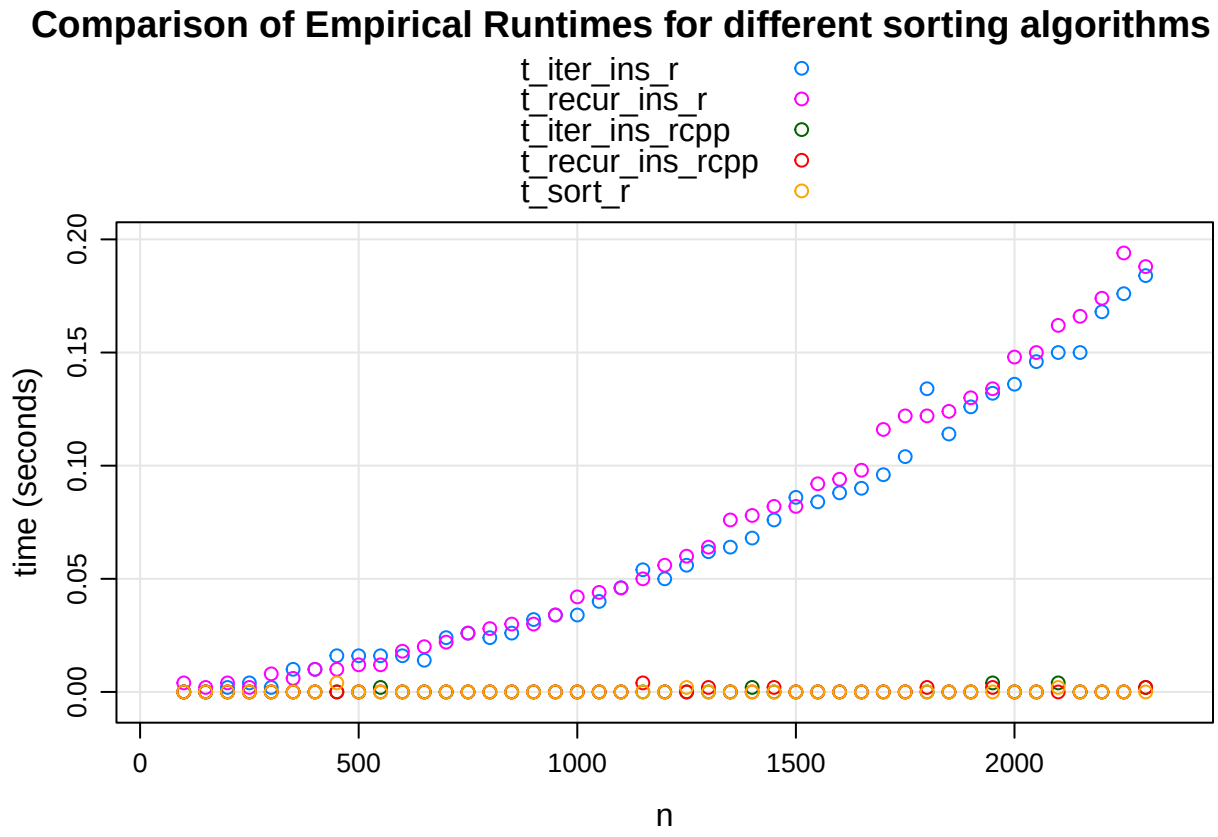
```
timeSort <- function(size, nrep , sort.fun )
{
    x <- replicate(nrep, runif(size), simplify = FALSE)
    system.time(lapply(x, sort.fun))["elapsed"] / nrep
}
n <- seq(100, 2300, by = 50)
sin.recur.ins.r <- function(A){
  n = length(A)
  recur.ins.sort(A,n)
}

sin.recur.ins.rcpp <- function(A){
  n = length(A)
  recur_ins_sort_Rcpp(A,n)
}
set.seed(0)
t_iter_ins_r <- sapply(n, timeSort, nrep = 5,
                                    sort.fun = iter.ins.sort)
t_recur_ins_r <- sapply(n, timeSort, nrep = 5 ,
                                    sort.fun = sin.recur.ins.r)
t_iter_ins_rcpp <- sapply(n, timeSort, nrep = 5,
                                       sort.fun = iter_ins_sort_rcpp)
t_recur_ins_rcpp <- sapply(n, timeSort, nrep = 5 ,
                                       sort.fun = sin.recur.ins.rcpp)
t_sort_r <- sapply(n, timeSort, nrep = 5, sort.fun = sort)
```

We plot the empirical runtimes of the different sorting algorithms against input size individually and again together on the same plot for comparing them.

```
xyplot(t_iter_ins_r + t_recur_ins_r + t_iter_ins_rcpp + t_recur_ins_rcpp
       + t_sort_r ~ n , grid = TRUE , outer = TRUE, ylab = "time (seconds)",
       scales = list(y = "free", x = "free"),
       main="Individual Empirical Runtimes for different sorting algorithms")
```



```
xyplot(t_iter_ins_r + t_recur_ins_r + t_iter_ins_rcpp + t_recur_ins_rcpp
       + t_sort_r ~ n , grid = TRUE , ylab = "time (seconds)",
       scales = list(y = "free", x = "free"), auto.key = TRUE ,
       main="Comparison of Empirical Runtimes for different sorting algorithms")
```



Note:

- Observations for each implementation individually:
  - Iterative version using R - The empirical runtime for this implementation appears to be monotonically increasing in a quadratic manner with respect to the increasing input array size , with runtime reaching a maximum of 0.20 seconds for input array size 2300. As far as we tested the program sorted correctly even upto input array sizes 5000.
  - Recursive version using R - The empirical runtime for this implementation appears to be monotonically increasing in a quadratic manner with respect to the increasing input array size , with runtime reaching a maximum of 0.25 seconds for input array size 2300. As noted due to stack error the program does not run for larger input arrays.
  - Iterative version using Rcpp - The empirical runtime for this implementation appears to be almost zero for majority of input array sizes, and for the rest the difference is negligible with respect to 0.This implementation correctly sorts arrays even upto sizes 5000 and larger.
  - Recursive version using Rcpp - The empirical runtime for this implementation appears to be almost zero for almost every input array size, and for the rest the difference is negligible with respect to 0. As noted due to deep nested recursion the program does not run for larger size input arrays than 2300.
  - Inbuilt sort function in R - The empirical runtime for this implementation appears to be almost zero for almost every input array size, and for the rest the difference is negligible with respect to 0.

This implementation runs for every input array size we could think of.

- Observations regarding comparison of the implementations:

The empirical runtimes for the iterative and recursive versions of insertion sort using Rcpp along with the inbuilt sort function are almost zero for every input array size upto 2300, after which we cannot compare since the first two implementations do not work for larger input array sizes. Within this range of input array sizes The empirical runtimes of the iterative and recursive versions of insertion sort using R are significantly higher than the other three implementations and the runtimes for them are comparable among themselves upto around input sizes 1800 to 2000, both increasing monotonically in a quadratic manner with increase in input array size and for array sizes in the range 2000 to 2300, the iterative version using R performs better than its corresponding recursive version.