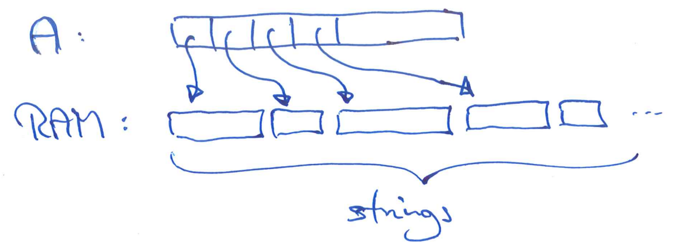
Department of Mathematics and Computer Science
University of Southern Denmark

January 19, 2017
RF

RadixQuicksort

RadixQuicksort is an algorithm for sorting strings, proposed by Bentley and Sedgewick in 1997 (based in earlier ideas by others). It can be said to be a mix of Quicksort and MSD Radixsort, hence the naming here. The authors used the name Multikey Quicksort, which fits with their focus on equal length strings (they are isomorphic to tuples), but not so well with the setting of general strings which we consider here.

The algorithms makes no assumptions on the size of the alphabet (i.e., it works for unbounded/comparison-based alphabets). The input to the algorithm is an array A, whose elements are representatives of n strings to be sorted. The strings themselves reside in RAM (each string consecutively laid out), and each string representative is a pair (p, l), where p is a pointer to the position in RAM of the first character of the string and l is the length of the string.

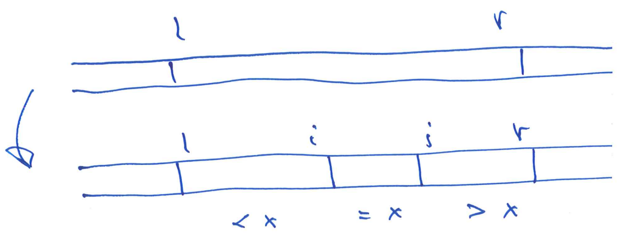


The output is the array A with the string representatives appearing in the sorted order given by the strings.

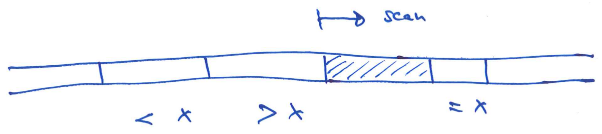
Recall Quicksort:

```
Quicksort(A)            Qsort(l,r,A)
  n = |A|                 if l < r
  Qsort(0,n-1,A)            x = ChoosePivot(l,r,A)
                            (i,j) = Partition(l,r,x,A)
                            QSort(l,i,A)
                            // no call QSort(i+1,j-1,A) needed
                            QSort(j,r,A)
```

Here, `Partition(l,r,x,A)` makes a three-way partitioning of $A[l..r]$ into segments of the elements less than, equal to, and larger than x:



It uses $O(r-l)$ time and returns the indices of the borders created around the middle segment. The method works similarly to the two-way `Partition` method you have learnt in a previous course, by maintaining the invariant illustrated below during its execution. At the end, the elements equal to x are swapped to the right of the elements larger than x.



The call `x = ChoosePivot(l,r,A)` is some method of choosing a pivot element from $A[l..r]$ to partition on. It may for instance be the element $A[r]$, the median of $A[l]$, $A[r]$, and $A[\lfloor (l+r)/2 \rfloor]$, a random element in $A[l..r]$, or the actual median of $A[l..r]$. The first options take $O(1)$ time, the last option can be be performed in $O(r-l)$ time (as you have learnt in a previous course). The last option is not competitive in practice, but gives a good worst case analysis, and we will assume that option here.

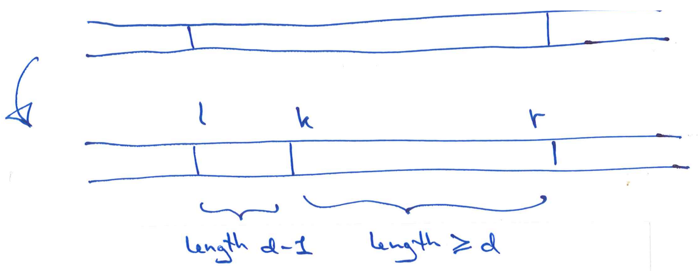
The algorithm RadixQuicksort is quite similar to Quicksort, except that the comparisons are between characters of some depth d in the strings. Each such character can be found from the first entry of the string representative and the value of d.

```
RadixQuicksort(A)        RQsort(l,r,d,A)
  n = |A|                  if l < r
  RQsort(0,n-1,1,A)          k = ShiftShortToFront(l,r,d-1,A)
                             x = ChoosePivot(k,r,d,A)
                             (i,j) = Partition(k,r,x,d,A)
                             RQSort(k,i,d,A)
                             RQSort(i+1,j-1,d+1,A)
                             RQSort(j,r,d,A)
```

The core idea of the algorithm is captured by the following invariant:

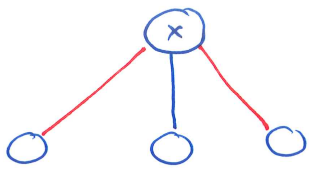
> At a call `RQsort(l,r,d,A)`, the strings with representatives in $A[l..r]$ all have length at least $d-1$, and their first $d-1$ characters are the same.

The call `k = ShiftShortToFront(l,r,d-1,A)` is a linear time scan of $A[l..r]$ which partitions it such that all strings of length $d-1$ appears first (leftmost) and returns the left border k of the segment of A with the remaining longer strings:



The calls `x = ChoosePivot(k,r,d,A)` and `(i,j) = Partition(k,r,x,d,A)` are similar to those of Quicksort, except that the comparisons are between characters of depth d in the strings.

Correctness of the algorithm follows directly from observing that it maintains the invariant. For analysis of the time complexity, we consider the recursion tree of the algorithm. For each node in the recursion tree, we color blue the edges of recursive calls which increase the value of d and we color red the edges of recursive calls which do not increase the value of d:



Each string leaves the recursion either in an internal node due to having no more characters (and being shifted to front by `ShiftEndedToFront`), or by being the single string left (in the case $l = r$). The local work in each node of the recursion tree is linear in the input size $r - l$. Hence, all work is paid for if every string s pays $O(1)$ for each node on the path in the recursion tree from the root to the node where the string leaves the recursion. On such a path, there can be at most $|s|$ blue edges, since the string depth d of the recursion increase by one for each such string. Also, there can be at most $\log n$ red edges, since for each traversal of a red edge, the size $r - l$ of the array segment decreases by at least a factor of two by our choice of the median as pivot element. In conclusion, the time complexity is

$$O(n \log n + S),$$

where S is the total length of the strings. A bit stronger, S can be taken to be the sum of the distinguishing prefixes of the string, where the distinguishing prefix of a string is one more than its longest common prefix with any of the other strings. An easy adversary argument shows that this is best possible for comparison based alphabets (a lower bound of $n \log n$ follows from sorting strings of length one, and a lower bound of S follows from the need to examine all characters of all distinguishing prefixes unless an adversary can prove the sorting wrong).

Note that the algorithm can be seen as an elegant and coherent way to implement MSD Radixsort using Quicksort for sorting buckets.

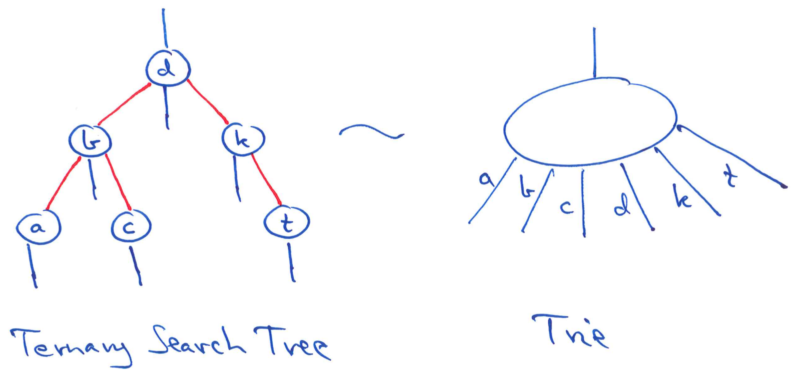
Finally, note that if the recursion tree is physically created and stored during the recursion, it can afterwards be used as a search tree over the strings. For a query string q, we consider characters of q at increasing depths while searching the tree from the root. If the current character c matches the character x of the current node (which is the pivot element chosen at the node during the execution of RadixQuicksort), we follow the blue edge from the node, and advances to the next character of q. Else, we follow the left or right red edge, based on whether $c < x$ or $c > x$.

The correctness of the search algorithm is clear from the working of the algorithm. Its running time is $O(\log n + |q|)$, by essentially the same argument as before: each time a blue edge is followed, we advance in q, and each time a red edge is followed, the number of strings stored below the current node is at least halved.

The recursion tree used in that manner is termed a *ternary search tree* by the authors.

We note that when implementing a trie for a set of strings, each trie node will always be implemented by a dictionary of some form (linked list, array, hash table, binary search tree). The data of each dictionary are trie edges of the corresponding trie node, and the search keys of the dictionary are the characters of these edges.

Ternary search trees can be seen as an elegant and coherent way to construct tries in which each trie node is implemented as a binary search tree. Blue edges of the ternary search tree correspond to trie edges (data in the dictionary, stored in the nodes of the binary tree along with the character search key of the node). Red edges of the ternary search tree correspond to binary tree edges in the dictionaries, each binary tree/trie node being a maximal subset of ternary search tree nodes connected by red edges. This correspondence is illustrated below for one such connected set (i.e., for one binary tree/trie node).


d
b
k
a
c
t
~
a
b
c
d
k
t
Ternary Search Tree
Trie