Advanced Algorithm
Solutions Homework 2

Eric Nordenstam, Erwan Lemonnier

December 3, 2000

1 Suffix array implementation

1.1 Sorting the suffix array

We tried 3 algorithms to sort the suffix array:

- Quicksort
- 3 implementations of counting sort
- the Manber-Myers algorithm

Below is a short description of our implementations of each of these algorithm, as well as a comparison of their efficiency. All the algorithms below were implemented in C, and compiled using the -O3 option, which for some algorithms divided the running time by up to 2.

1.1.1 Quicksort

A naiv implementation of quicksort. Quicksort itself runs in time $O(n\log n)$, but since it compares strings of up to n characters, the comparisons will have to test in average $n/2$ characters. Hence, to sort suffix arrays, quicksort runs in $O(n^2\log n)$

1.1.2 Counting sort

Each character in a string is an integer of range between 0 and 255, which makes it reasonable to try a counting sort. Our first implementation runs recursively, and begins by sorting on the first character of each suffix string, then on the 2nd character of each group of suffix strings having the same 1st character, and so on.

We wrote a second optimized implementation of the above algorithm, using a loop instead of recursivity, and 4 arrays of the size of the suffix array, allocated from the beginning to avoid dynamic memory allocation. We gained about 10% of time, but we were still on average 5 times slower than the target speed.

We did a third implementation in which the counting sort is made on 2 characters at a time, but it appeared to be much slower, for 2 reasons: the resulting array is very large (65536 indexes) and if the text only contains alphabetical characters (= it's not a binary file) only a short range of it is used, and time is

wasted to go across the array. Moreover, some computation is required to invert the order of the 2 characters (on x86 processors at least).

This recursive counting sort runs in $O(nl)$, where l is the size of longest repeated substring in the text. Thus, for well-behaved texts (with few repetitions), it can sort the suffix array reasonably fast, but in the worst case ($l = n$, ie a text made of the same repeated letter), it runs in time $O(n^2)$.

1.1.3 Manber-Myers algorithm

This algorithm was described in the article "Suffix arrays: A new method for on-line string searches" by Udi Manber and Gene Myers. We describe the general idea of the algorithm first, before describing it in detail. Take a string of length n. In the first stage of the algorithm, we bucket on the first character in every suffix. After this is done, we do no longer need to look at the character string, all the information we need is in the placement of suffixes in buckets. For simplicity, we number the stages in the algorithm 1, 2, 4, 8, etc. After stage number h, the suffix array will be sorted after the h first characters in each suffix. This is done in the following manner.

We have an array where the suffixes are bucketed according to the first h characters. We want to make a new array with the suffixes bucketed on the first $2h$ characters. Initialize a stack for every h bucket, the size of which is the same as the number of elements in that bucket. Thus all these stacks will fit into an array of length n. Take the first element in the first bucket. Let the index of that suffix be k. If $k - h > 0$, then lookup in what bucket the suffix $k - h$ is located and move it to the top of the stack for that bucket. Take the second element in the first bucket, and place it on the top of its stack. Go on till you have exhausted the first bucket. Do the same for the rest of the buckets. After this, the array with all the stacks will contain all the suffixes sorted on their $2h$ first characters.

Every stage takes time $O(n)$ and there are $\log n$ stages, which gives a running time of $O(n \log n)$.

We need to do a lot of accounting. For this we need three integer arrays and two boolean arrays, all with length n. The first of the integer arrays we will call `arSuffix` and will contain the suffix array. The second we will call `arPrm` and it will be such that `arPrm`[`arSuffix`[i]] $= i$. The third integer array `arCount` will keep the height of the stacks that we push elements to. The two boolean arrays will be called `arBucketStart` and `arBucket2Start`. `arBucketStart`[i] will be true if a h bucket starts at i and false otherwise. `arBucket2Start`[i] will be true if a $2h$ bucket starts at i and false otherwise. The stage h can now be described in the following way.

1. Initialize the `arCount` array to zero. This requires time $O(n)$

2. Set `arPrm`[i] to the position of the leftmost element in the h-bucket containing the ith suffix rather than the exact placement of the ith suffix. This can certainly be done in $O(n)$ time.

3. Scan the `arSuffix` array in increasing order, one bucket at a time. Let l and r ($l \leq r$) represent the left and right boundry of the bucket currently scaned. Let T_i be `arSuffix`[i] $- h$. For every $l \leq i \leq r$ we increment `arCount`[`arPrm`[T_i]], set `arSuffix`[T_i] = `arSuffix`[T_i]+`arCount`[`arPrm`[T_i]]−

1 and set `arStart2Bucket`$[i]$ = `true`. Here we use the `arStart2Bucket` array to mark the element that have moved to the top of their stacks.

4. Before we go on to the next h-bucket, make an other pass through this one to find all moved suffixes and set `arStart2Bucket` to false for all but the leftmost one in each $2h$-bucket. This will make the `arStart2Bucket` correctly mark the start of each $2h$-bucket.

5. Scan all buckets in this way.

6. Set `arBucketStart` := `arBucket2Start` and set $h := 2 \cdot h$. If $h > n$ then we are done. Otherwise go on to the next stage.

1.1.4 Comparison

We ran tests on a 466 mhz intel celeron computer, with 128 MB of RAM and a Linux operating system, and here is the time required by the algorithms to build suffix arrays:

size of text	93,7 Kb	626 Kb	1,1 Mb	3,3 Mb
quicksort	61"550	> 4500"		
counting sort	9"200	100"130		1150
optimized counting sort	8"	73"530		888"
manber-myers	1"		25"	125"

1.2 counting occurrences

Our algorithm works in 2 steps: first, finding an occurrence of the substring, then searching for the most distant occurrence on the left and right side of it. The difference between the position of the last right and left occurrences is the answer.

To find one occurrence, we use a binary search that divides the search interval by 2 at each iteration. A trick is to compare only the relevant characters when 2 string are being compared, since we already know that their d-th first characters are the same. It has a complexity of $O(l\log n)$, where l is the length of the substring searched. Usually, $l << n$, and we have $O(\log n)$ The search of the last right or left occurrence takes also $O(l\log n)$: it searches the first different string on one side, multiplying by 2 the distance at each step, then runs a binary search between the first occurrence of the different string and the known occurrence of the string to find the last same string.

Hence, the global complexity is $O(\log n)$.

1.3 Length of the longest repeated substring

The idea is to recursively search the suffix array for every string that is repeated at least once, and to keep track of the longest string met in the process. Below is a pseudocode for an algorithm performing this task:

```
longest(SuffixArray, left, right, digit)
1       l = 0
2       while left < right
3               p = indexOfFirstOccurence(SuffixArray, left, right, digit)
4               if right − p > 0
5                       l = MAX(1, l)
6                       l = MAX(l, 1 + longest(SuffixArray, p, right, digit + 1))
7               right = p − 1
8       returnl
```

In the algorithm above, indexOfFirstOccurence returns the index of the first (on the left) occurrence in SuffixArray of a string identical up to the digit-th character with the suffix at index 'right' in SuffixArray.

This algorithm's complexity is hard to analyse because the number of calls between two levels of recursion depends on the text analyzed, but we can give an upper bound. At each level of recursion, at most n calls to longest() can be made, for a total of less than $n\log n$ operations (indexOfFirstOccurence runs in $O(\log n)$). And the number of levels is l, the length of the longest repeated substring. Hence, the complexity is in $O(ln\log n)$. In the worst case, $l \approx n$ and complexity $= n^2\log n$, but this is a large approximation.

We can show that in the worst case (a text made of n times the same character), the complexity is $n\log n$.

1.4 Meta-character search

The algorithm we designed is based on Dynamic Programming: let's consider the string to find as a sequence of substrings separated by meta-characters '*'. We begin by finding all occurrences of the first substring, and build 2 arrays, one containing the indexes of suffixes beginning with this substring, and one containing the number of matches found for the corresponding suffix (1 if the meta-string does not begin with '*', the number of characters in the text before the suffix otherwise).

Then we search for the occurrence of the N-th substring, separated from the previous by a '*', and we update the 2 arrays. The first contains the indexes to suffixes beginning with the new substring. The second contains the number of substrings in the text that match the meta-character substring up to this substring included. This can be computed by taking the sum of the number of substrings in the text matching up to the previous substring (and we indeed have access to the previous array) for each possible position of the previous substring that is compatible with the current position of the current substring. At the end, the number of matches found is the sum of the values in the second array.

Special care is required when the meta-character string starts or end with a '*', or when it is only a '*' (in which case the number of matches is $\frac{l(l+1)}{2}$, l being the length of the text).

Each time the algorithm goes through the loop of taking a new substring of length l_i into account, it requires $O(l_i\log n)$ to find all the occurrences of the substring, then $O(l_i)$ to update the 2 arrays. The final complexity is

$$\begin{aligned}\sum_i (O(l_i \log n) + O(l_i)) &= O(\sum_i l_i(1 + \log n)) \\ &= O(\log n) \text{ if } \forall\ i,\ (l_i << n)\end{aligned}$$

1.5 Our C implementation

Finally, our implementation of the above algorithms, written in C, is 1200 lines long (including comments), and can be run from a unix command line with the following arguments:

```
index [-mX] [-v] [-c ‘‘string’’] [-mc ‘‘string’’] [-l] <filename>
```

where:

- -m0, -m1, -m2, -m3 selects the sorting algorithm to use to build the suffix array. Respectively Quicksort, recursive counting sort, looping counting sort and Manber-Myers. The default value is -m3.
- -v (verbose) displays the suffix array (avoid for big texts).
- -c “string” counts the occurrences of string in the text.
- -mc “string” counts the occurrences of string, which can contain ’*’ meta-characters.
- -l returns the length of the longest substring in the text.

This program (source code and binary file) is available at f99-eno/avalg.

2 Fast sorting

First, me must confess that we are not sure to properly understand these questions: do they concern sorting algorithms in general, or speacial cases of the $O(n \log \log n)$ algorithm ? Below we will assume the first case.

2.1 Question 1

Independently of the number of elements, various conditions are required to enable the use of linear time sorting algorithms: if the elements all contain the same limited number of digits, Radix sort can be used. If the elements all belong to a restricted set of possible values, Counting Sort can be used. If the elements have their values limited to an interval in which they are uniformly distributed, Bucket Sort may be used. Counting Sort, Bucket Sort and Radix Sort all run in linear time.

A condition on the number of elements is not enough to place us in one of the 3 above cases. If we assume that the value of the elements has an upper bound, then we could probably use a linear algorithm.

2.2 Question 2

For simplicity, let's suppose that all elements are positive integers. The condition is that they are no larger than some small constant N. Then it is possible to sort them in linear time using Counting Sort. The pseudocode for Counting Sort is given below:

```
    CountingSort(A, B, k)
1       for i ← 1 to k
2           C[i] ← 0
3       for j ← 1 to length[A]
4           C[A[j]] ← C[A[j]] + 1
    //C[i] contains the number of elements equal to i
5       for i ← 2 to k
6           C[i] ← C[i] + C[i − 1]
    //C[i] now contains the number of elements less than or equal to i
7       for j ← length[A] downto 1
8           B[C[A[j]]] ← A[j]
9           C[A[j]] ← C[A[j]] − 1
```

A is an array containing the elements to sort, B will contain the sorted elements, k is the number of elements, and C is an intermediary array used to determine the position of the elements in B. An implementation of Counting Sort is used in one of our algorithm used to build the suffix array (the recursive counting sort).

2.3 Question 3

2.4 Question 4

3 Simplex method

3.1 Theory

The idea of the simplex method is simple. Take a linear function $f : \mathbb{R}^n \to \mathbb{R}$. Also take a set of linear functions $g_1, \ldots, g_m : \mathbb{R}^n \to \mathbb{R}$ and a set of real values $a_1, \ldots, a_m$. Let $\mathbb{R}^n \supset K = \{x : g_1(x) \leq a_1, g_2(x) \leq a2, dots, g_m(x) \leq a_m\}$. Since K is a closed set and f continuous, we know that $\max_{x \in K} f(x)$ exists. Our task is to find a value x_0 such that $f(x_0) = \max_{x \in K} f(x)$.

The problem can be formulated on matrix form: Find $\max c^T x$ subject to $Ax \leq b$.

It is easy to realize that K is convex. It also has vertexes, i.e. points where n of the constraints $g_1(x) \leq a_1, \ldots, g_m(x) \leq a_m$ are satisfied with equality. It is also easy to realise that the maximum of f is attained in one of these vertexes. So to find this vertex, start at any vertex, call this vertex H and do the following. (This is more or less copied from the course notes.)

- Find the equalities which are satisfied with equality at H, forming a matrix A_H.
- Invert A_H.

- Find a unit vector e_i such that $c^T A_H^{-1} e_i < 0$. If to such unit vector exists, then H is optimum. Otherwise, for the one found e_i replace H with $H - t_{max} A_H^{-1}$ for the maximum t_{max} that gives a new H that does not violate $Ax \leq b$.

3.2 Practice

Suppose we wish to maximize

$$Z = 10x_1 + 6x_2 + 4x_3$$

subject to the linear constraints

$$\begin{gathered} x_1 + x_2 + x_3 \leq 100 \\ 10x_1 + 4x_2 + 5x_3 \leq 600 \\ 2x_1 + 2x_2 + 6x_3 \leq 300 \\ x_1 \geq 0 \\ x_2 \geq 0 \\ x_3 \geq 0 \end{gathered}$$

We can write this on matrix form.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 10 & 4 & 5 \\ 2 & 2 & 6 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 100 \\ 600 \\ 300 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad c = \begin{bmatrix} 10 \\ 6 \\ 4 \end{bmatrix} \tag{1}$$

Maximize $c^T x$ subject to $Ax \leq b$.

The simplex method starts at a feasible vertex and moves from vertex to vertex improving the objective function. Let us start at the vertex $v_1 = (x_1, x_2, x_3) = (0, 0, 0)$. Pick out those inequalities that are satisfied with equality at this vertex. These can on matrix form be writen as $A_1 v_1 = b_1$ for

$$A_1 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Invert A_1. In this case it is easy since we see that A_1 is an orthogonal matrix so $A_1^{-1} = A_1^T$. Now check whether any of $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$ or $e_3 = (0, 0, 1)$ satisfy $c^T A_1^{-1} e_i < 0$. Here, it turns out that all of e_1, e_2 and e_3 satisfy this. We pick e_1 and replace our current vertex v_1 with $v_2 = v_1 - t_{max} A_1^{-1} e_1$ where $t_{max} \in \mathbb{R}$ is the maximum we can use without violating any constraints. $A_1^{-1} e_1 = (-1, 0, 0)$. The largest t_{max} we can choose without violating the first constraint is 100. The largest t_{max} we can choose without voilating the second constraint is 60, and for the third constraint it is 150. So we choose $t_{max} = 60$ so as not to violate any constraint.

We now have a new vertex $v_2 = (60, 0, 0)$. The second, fifth and sixth constraints are satisfied with equality, so write these on matrix form: $A_2 v_2 = b_2$ for

$$A_2 = \begin{bmatrix} 10 & 4 & 5 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 600 \\ 0 \\ 0 \end{bmatrix}$$

Invert A_2.

$$A_2^{-1} = \begin{bmatrix} \frac{1}{10} & \frac{2}{5} & \frac{1}{2} \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Calculate $c^T A_2^{-1} = [1, -2, 1]$. We see that $c^T A_2^{-1} e_2 < 0$. We want to replace v_2 with $v_3 = v_2 - t_{max} A_2^{-1} e_2$ for some new value t_{max} so that v_3 does not violate any constraints. $A_2^{-1} e_2 = (\frac{2}{5}, -1, 0)$. A t_{max} of $\frac{200}{3}$ is the largest t_{max} that does not violate the first constraint. As for the second constraint, any t_{max} will do. As for the third constraint, it will tolerate a t_{max} of 150. So we choose $t_{max} = \frac{200}{3}$. This gives $v_3 = \frac{1}{3}(100, 200, 0)$.

Again, pick out the inequalities which are satisfied with equality by this vertex. It turns out to be the first, the second and the last inequality. Write them on matrix form: $A_3 v_3 = b_3$.

$$A_3 = \begin{bmatrix} 1 & 1 & 1 \\ 10 & 4 & 5 \\ 0 & 0 & -1 \end{bmatrix}, \quad b_3 = \begin{bmatrix} 100 \\ 600 \\ 0 \end{bmatrix}$$

Invert A_3.

$$A_3^{-1} = \begin{bmatrix} -\frac{2}{3} & \frac{1}{6} & \frac{1}{6} \\ \frac{5}{3} & -\frac{1}{6} & \frac{5}{6} \\ 0 & 0 & -1 \end{bmatrix}$$

Now let us try to find an e_i such that $c^T A_3^{-1} e_i < 0$. But $c^T A_3^{-1} = \frac{1}{3}(10, 2, 8)$ so no such e_i exists. Therefore, we have found an optimum. The answer is $x_0 = \frac{1}{3}(100, 200, 0)$ which gives $Z = \frac{2200}{3}$.