Overview

This is an O(n log n) algorithm for suffix array construction (or rather, it would be, if instead of `::sort` a 2-pass bucket sort had been used).

It works by first sorting the 2-grams(*), then the 4-grams, then the 8-grams, and so forth, of the original string `S`, so in the i-th iteration, we sort the 2^i-grams. There can obviously be no more than $\log_2(n)$ such iterations, and the trick is that sorting the 2^i-grams in the i-th step is facilitated by making sure that each comparison of two 2^i-grams is done in O(1) time (rather than $O(2^i)$ time).

How does it do this? Well, **in the first iteration** it sorts the 2-grams (aka bigrams), and then performs what is called *lexicographic renaming*. This means it creates a new array (of length `n`) that stores, for each bigram, its *rank* in the bigram sorting.

Example for lexicographic renaming: Say we have a **sorted** list of some bigrams `{'ab','ab','ca','cd','cd','ea'}`. We then assign *ranks* (i.e. lexicographic names) by going from left to right, starting with rank 0 and incrementing the rank whenever we encounter a *new* bigram changes. So the ranks we assign are as follows:

```
ab : 0
ab : 0   [no change to previous]
ca : 1   [increment because different from previous]
cd : 2   [increment because different from previous]
cd : 2   [no change to previous]
ea : 3   [increment because different from previous]
```

These ranks are known as *lexicographic names*.

Now, **in the next iteration**, we sort 4-grams. This involves a lot of comparisons between different 4-grams. How do we compare two 4-grams? Well, we could compare them character by character. That would be up to 4 operations per comparison. But instead, we compare them by *looking up* the ranks of the two bigrams contained in them, using the rank table generated in the previous steps. That rank represents the lexicographic rank from the previous 2-gram sort, so if for any given 4-gram, its first 2-gram has a higher rank than the first 2-gram of another 4-gram, then it must be lexicographically greater *somewhere in the first two characters*. Hence, if for two 4-grams the rank of the first 2-gram is identical, they must be identical in the *first two characters*. In other words, *two look-ups* in the rank table are sufficient to compare all 4 characters of the two 4-grams.

After sorting, we create new lexicographic names again, this time for the 4-grams.

In the third iteration, we need to sort by 8-grams. Again, two look-ups in the lexicographic rank table from the previous step are sufficient to compare all 8 characters of two given 8-grams.

And so forth. Each iteration `i` has two steps:

1. Sorting by 2^i-grams, using the lexicographic names from the previous iteration to enable comparisons in 2 steps (i.e. O(1) time) each

2. Creating new lexicographic names

We repeat this until all 2^i-grams are different. If that happens, we are done. How do we know if all are different? Well, the lexicographic names are an increasing sequence of integers, starting with 0. So if the highest lexicographic name generated in an iteration is the same as `n-1`, then each 2^i-gram must have been given its own, distinct lexicographic name.

Implementation

Now let's look at the code to confirm all of this. The variables used are as follows: `sa[]` is the suffix array we are building. `pos[]` is the rank lookup-table (i.e. it contains the lexicographic names), specifically, `pos[k]` contains the lexicographic name of the `k`-th m-gram of the previous step. `tmp[]` is an auxiliary array used to help create `pos[]`.

I'll give further explanations between the code lines:

```
void buildSA()
{
    N = strlen(S);

    /* This is a loop that initializes sa[] and pos[].
       For sa[] we assume the order the suffixes have
       in the given string. For pos[] we set the lexicographic
       rank of each 1-gram using the characters themselves.
       That makes sense, right? */
    REP(i, N) sa[i] = i, pos[i] = S[i];

    /* Gap is the length of the m-gram in each step, divided by 2.
       We start with 2-grams, so gap is 1 initially. It then increases
       to 2, 4, 8 and so on. */
    for (gap = 1;; gap *= 2)
    {
        /* We sort by (gap*2)-grams: */
        sort(sa, sa + N, sufCmp);

        /* We compute the lexicographic rank of each m-gram
           that we have sorted above. Notice how the rank is computed
           by comparing each n-gram at position i with its
           neighbor at i+1. If they are identical, the comparison
           yields 0, so the rank does not increase. Otherwise the
           comparison yields 1, so the rank increases by 1. */
        REP(i, N - 1) tmp[i + 1] = tmp[i] + sufCmp(sa[i], sa[i + 1]);

        /* tmp contains the rank by position. Now we map this
           into pos, so that in the next step we can look it
           up per m-gram, rather than by position. */
        REP(i, N) pos[sa[i]] = tmp[i];

        /* If the largest lexicographic name generated is
           n-1, we are finished, because this means all
           m-grams must have been different. */
        if (tmp[N - 1] == N - 1) break;
    }
}
```

About the comparison function

The function `sufCmp` is used to compare two (2*gap)-grams lexicographically. So in the first iteration it compares bigrams, in the second iteration 4-grams, then 8-grams and so on. This is controlled by `gap`, which is a global variable.

A naive implementation of `sufCmp` would be this:

```
bool sufCmp(int i, int j)
{
  int pos_i = sa[i];
  int pos_j = sa[j];

  int end_i = pos_i + 2*gap;
  int end_j = pos_j + 2*gap;
  if (end_i > N)
    end_i = N;
  if (end_j > N)
    end_j = N;

  while (i < end_i && j < end_j)
  {
    if (S[pos_i] != S[pos_j])
      return S[pos_i] < S[pos_j];
    pos_i += 1;
    pos_j += 1;
  }
  return (pos_i < N && pos_j < N) ? S[pos_i] < S[pos_j] : pos_i > pos_j;
}
```

This would compare the (2*gap)-gram at the beginning of the i-th suffix `pos_i:=sa[i]` with the one found at the beginning of the j-th suffix `pos_j:=sa[j]`. And it would compare them character by character, i.e. comparing `S[pos_i]` with `S[pos_j]`, then `S[pos_i+1]` with `S[pos_j+1]` and so on. It continues as long as the characters are identical. Once they differ, it returns 1 if the character in the i-th suffix is smaller than the one in the j-th suffix, 0 otherwise. (Note that `return a<b` in a function returning `int` means you return 1 if the condition is true, and 0 if it is false.)

The complicated looking condition in the return-statement deals with the case that one of the (2*gap)-grams is located at the end of the string. In this case either `pos_i` or `pos_j` will reach `N` before all (2*gap) characters have been compared, even if all characters up to that point are identical. It will then return 1 if the i-th suffix is at the end, and 0 if the j-th suffix is at the end. This is correct because if all characters are identical, the *shorter* one is lexicographically smaller. If `pos_i` has reached the end, the i-th suffix must be shorter than the j-th suffix.

Clearly, this naive implementation is O(gap), i.e. its complexity is linear in the length of the (2*gap)-grams. The function used in your code, however, uses the lexicographic names to bring this down to O(1) (specifically, down to a maximum of two comparisons):

```
bool sufCmp(int i, int j)
{
  if (pos[i] != pos[j])
    return pos[i] < pos[j];
  i += gap;
  j += gap;
  return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}
```

As you can see, instead of looking up individual characters `S[i]` and `S[j]`, we check the lexicographic rank of the i-th and j-th suffix. Lexicographic ranks were computed in the previous iteration for gap-grams. So, if `pos[i] < pos[j]`, then the i-th suffix `sa[i]` must start with a gap-gram that is lexicographically smaller than the gap-gram at the beginning of `sa[j]`. In other words, simply by looking up `pos[i]` and `pos[j]` and comparing them, we have compared the first *gap*characters of the two suffixes.

If the ranks are identical, we continue by comparing `pos[i+gap]` with `pos[j+gap]`. This is the same as comparing the next *gap* characters of the (2*gap)-grams, i.e. the *second half*. If the ranks are indentical again, the two (2*gap)-grams are indentical, so we return 0. Otherwise we return 1 if the i-th suffix is smaller than the j-th suffix, 0 otherwise.

Example

The following example illustrates how the algorithm operates, and demonstrates in particular the role of the lexicographic names in the sorting algorithm.

The string we want to sort is `abcxabcd`. It takes three iterations to generate the suffix array for this. In each iteration, I'll show `S` (the string), `sa` (the current state of the suffix array) and `tmp` and `pos`, which represent the lexicographic names.

First, we initialize:

```
S   abcxabcd
sa  01234567
pos abcxabcd
```

Note how the lexicographic names, which initially represent the lexicographic rank of unigrams, are simply identical to the characters (i.e. the unigrams) themselves.

First iteration:

Sorting `sa`, using bigrams as sorting criterion:

```
sa  04156273
```

The first two suffixes are 0 and 4 because those are the positions of bigram 'ab'. Then 1 and 5 (positions of bigram 'bc'), then 6 (bigram 'cd'), then 2 (bigram 'cx'). then 7 (incomplete bigram 'd'), then 3 (bigram 'xa'). Clearly, the positions correspond to the order, based solely on character bigrams.

Generating the lexicographic names:

```
tmp 00112345
```

As described, lexicographic names are assigned as increasing integers. The first two suffixes (both starting with bigram 'ab') get 0, the next two (both starting with bigram 'bc') get 1, then 2, 3, 4, 5 (each a different bigram).

Finally, we map this according to the positions in `sa`, to get `pos`:

```
sa  04156273
tmp 00112345
pos 01350124
```

(The way `pos` is generated is this: Go through `sa` from left to right, and use the entry to define the index in `pos`. Use the corresponding entry in `tmp` to define the value for that index. So `pos[0]:=0`, `pos[4]:=0`, `pos[1]:=1`, `pos[2]:=1`, and so on. The index comes from `sa`, the value from `tmp`.)

Second iteration:

We sort `sa` again, and again we look at bigrams from `pos` (which each represents a sequence of*two* bigrams of the original string).

```
sa  04516273
```

Notice how the position of 1 5 have switched compared to the previous version of `sa`. It used to be 15, now it is 51. This is because the bigram at `pos[1]` and the bigram at `pos[5]` used to be identical (both `bc`) in during the previous iteration, but now the bigram at `pos[5]` is `12`, while the bigram at `pos[1]` is `13`. So position `5` comes *before* position `1`. This is due to the fact that the lexicographic names now each represent bigrams of the original string: `pos[5]` represents `bc` and `pos[6]` represents 'cd'. So, together they represent `bcd`, while `pos[1]` represents `bc` and `pos[2]` represents `cx`, so together they represent `bcx`, which is indeed lexicographically greater than `bcd`.

Again, we generate lexicographic names by screening the current version of `sa` from left to right and comparing the corrsponding bigrams in `pos`:

```
tmp 00123456
```

The first two entries are still identical (both 0), because the corresponding bigrams in `pos` are both `01`. The rest is an strictly increasing sequence of integers, because all other bigrams in `pos` are each unique.

We perform the mapping to the new `pos` as before (taking indices from `sa` and values from `tmp`):

```
sa  04516273
tmp 00123456
pos 02460135
```

Third iteration:

We sort `sa` again, taking bigrams of `pos` (as always), which now each represents a sequence of 4 bigrams of the orginal string.

```
sa  40516273
```

You'll notice that now the first two entries have switched positions: `04` has become `40`. This is because the bigram at `pos[0]` is `02` while the one at `pos[4]` is `01`, the latter obviously being lexicographically smaller. The deep reason is that these two represent `abcx` and `abcd`, respectively.

Generating lexicographic names yields:

```
tmp 01234567
```

They are all different, i.e. the highest one is `7`, which is `n-1`. So, we are done, because are sorting is now based on m-grams that are all different. Even if we continued, the sorting order would not change.

Improvement suggestion

The algorithm used to sort the 2^i-grams in each iteration appears to be the built-in `sort` (or `std::sort`). This means it's a comparison sort, which takes O(n log n) time in the worst case, *in each iteration*. Since there are log n iterations in the worst case, this makes it a $O(n (\log n)^2)$-time algorithm. However, the sorting could by performed using two passes of bucket sort, since the keys we use for the sort comparison (i.e. the lexicographic names of the previous step), form an increasing integer sequence. So this could be improved to an actual O(n log n)-time algorithm for suffix sorting.

Remark

I believe this is the original algorithm for suffix array construction that was suggested in the 1992-paper by Manber and Myers (link on Google Scholar; it should be the first hit, and it may have a link to a PDF there). This (at the same time, but independently of a paper by Gonnet and Baeza-Yates) was what introduced suffix arrays (also known as pat arrays at the time) as a data structure interesting for further study.

Modern algorithms for suffix array construction are O(n), so the above is no longer the best algorithm available (at least not in terms of theoretical, worst-case complexity).

Footnotes

(*) By *2-gram* I mean a sequence of two *consecutive* characters of the original string. For example, when `S=abcde` is the string, then `ab`, `bc`, `cd`, `de` are the 2-grams of `S`.
Similarly, `abcd` and `bcde` are the 4-grams. Generally, an m-gram (for a positive integer m) is a

sequence of `m` consecutive characters. 1-grams are also called unigrams, 2-grams are called bigrams, 3-grams are called trigrams. Some people continue with tetragrams, pentagrams and so on.

Note that the suffix of `S` that starts and position `i`, is an (n-i)-gram of `S`. Also, every m-gram (for any m) is a prefix of one of the suffixes of `S`. Therefore, sorting m-grams (for an m as large as possible) can be the first step towards sorting suffixes.