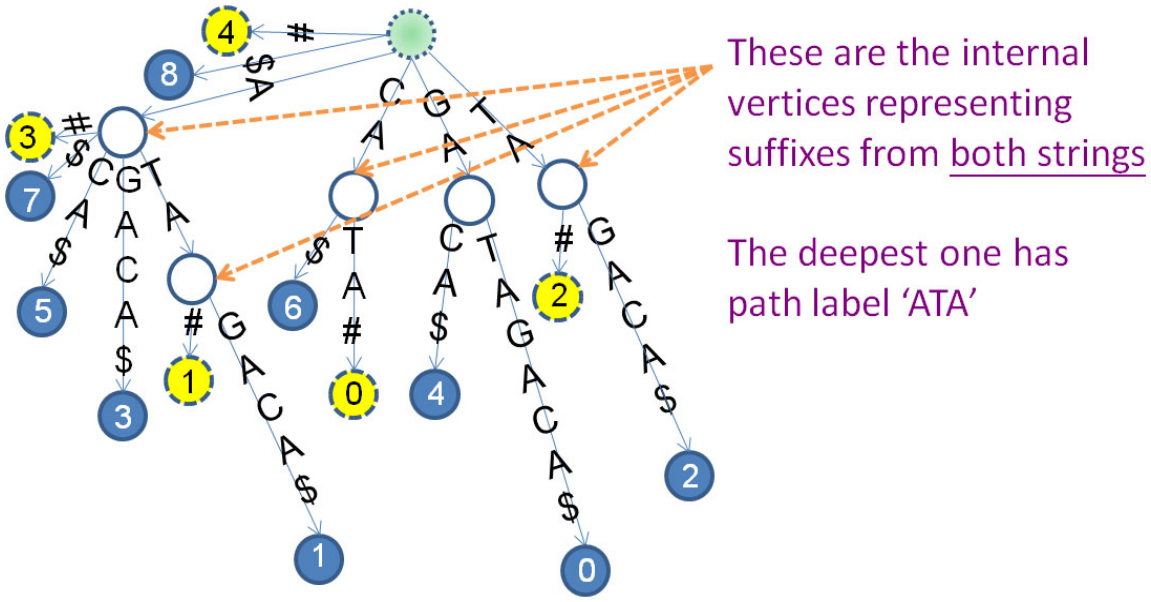


Figure 6.6: Generalized ST of T_1 = 'GATAGACA$' and T_2 = 'CATA#' and their LCS

Exercise 6.6.3.1: Given the same Suffix Tree in Figure 6.4, find P = 'CA' and P = 'CAT'!

Exercise 6.6.3.2: Find the LRS in T = 'CGACATTACATTA$'! Build the Suffix Tree first.

Exercise 6.6.3.3*: Instead of finding the LRS, we now want to find the repeated substring *that occurs the most*. Among several possible candidates, pick the longest one. For example, if T = 'DEFG1ABC2DEFG3ABC4ABC$', the answer is 'ABC' of length 3 that occurs three times (not 'BC' of length 2 or 'C' of length 1 which also occur three times) instead of 'DEFG' of length 4 that occurs only two times. Outline the strategy to find the solution!

Exercise 6.6.3.4: Find the LCS of T_1 = 'STEVEN$' and T_2 = 'SEVEN#'!

Exercise 6.6.3.5*: Think of how to generalize this approach to find the LCS of *more than two strings*. For example, given three strings T_1 = 'STEVEN$', T_2 = 'SEVEN#', and T_3 = 'EVE@', how to determine that their LCS is 'EVE'?

Exercise 6.6.3.6*: Customize the solution further so that we find the LCS of k *out of* n *strings*, where $k \leq n$. For example, given the same three strings T_1, T_2, and T_3 as above, how to determine that the LCS of 2 out of 3 strings is 'EVEN'?

6.6.4 Suffix Array

In the previous subsection, we have shown several string processing problems that can be solved *if the Suffix Tree is already built*. However, the efficient implementation of linear time Suffix Tree construction (see [65]) is complex and thus risky under programming contest setting. Fortunately, the next data structure that we are going to describe—the **Suffix Array** invented by Udi Manber and Gene Myers [43]—has similar functionalities as Suffix Tree but (much) simpler to construct and use, especially in programming contest setting. Thus, we will skip the discussion on $O(n)$ Suffix Tree construction [65] and instead focus on the $O(n \log n)$ Suffix Array construction [68] which is easier to use. Then, in the next subsection, we will show that we can apply Suffix Array to solve problems that have been shown to be solvable with Suffix Tree.

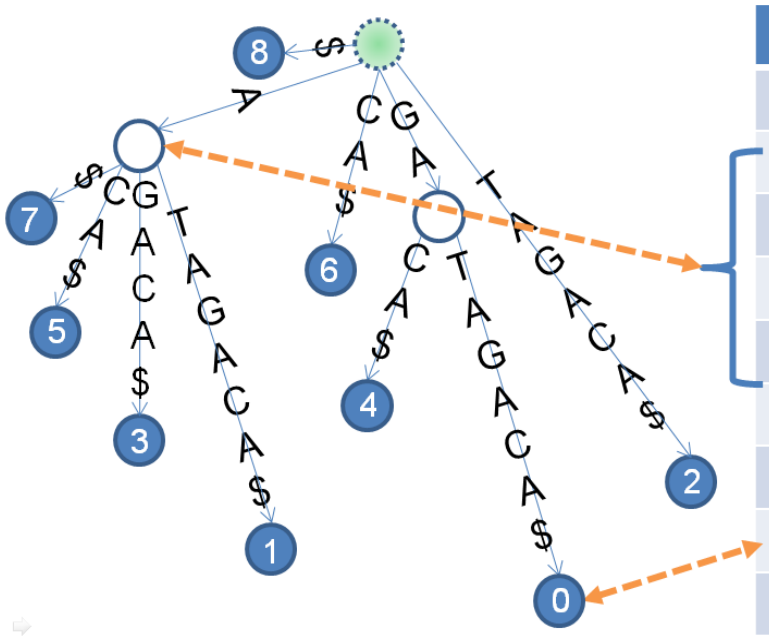
i	Suffix
0	GATAGACA$
1	ATAGACA$
2	TAGACA$
3	AGACA$
4	GACA$
5	ACA$
6	CA$
7	A$
8	$

Sort →

i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Figure 6.7: Sorting the Suffixes of `T = 'GATAGACA$'`

Basically, Suffix Array is an integer array that stores a permutation of n indices of *sorted* suffixes. For example, consider the same `T = 'GATAGACA$'` with $n = 9$. The Suffix Array of `T` is a permutation of integers `[0..n-1] = {8, 7, 5, 3, 1, 6, 4, 0, 2}` as shown in Figure 6.7. That is, the suffixes in sorted order are suffix `SA[0]` = suffix `8` = `'$'`, suffix `SA[1]` = suffix `7` = `'A$'`, suffix `SA[2]` = suffix `5` = `'ACA$'`, ..., and finally suffix `SA[8]` = suffix `2` = `'TAGACA$'`.



i	SA[i]	Suffix
0	8	$
1	7	A$
2	5	ACA$
3	3	AGACA$
4	1	ATAGACA$
5	6	CA$
6	4	GACA$
7	0	GATAGACA$
8	2	TAGACA$

Figure 6.8: Suffix Tree and Suffix Array of `T = 'GATAGACA$'`

Suffix Tree and Suffix Array are closely related. As we can see in Figure 6.8, the tree traversal of the Suffix Tree visits the terminating vertices (the leaves) in Suffix Array order. An **internal vertex** in Suffix Tree corresponds to a **range** in Suffix Array (a collection of sorted suffixes that share a common prefix). A **terminating vertex** (always at leaf due to the usage of a terminating character) in Suffix Tree corresponds to an **individual index** in Suffix Array (a single suffix). Keep these similarities in mind. They will be useful in the next subsection when we discuss applications of Suffix Array.

Suffix Array is good enough for many challenging string problems involving *long strings* in programming contests. Here, we present two ways to construct a Suffix Array given a string `T[0..n-1]`. The first one is very simple, as shown below:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

#define MAX_N 1010                              // first approach: O(n^2 log n)
char T[MAX_N];    // this naive SA construction cannot go beyond 1000 chars
int SA[MAX_N], i, n;                       // in programming contest settings

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }          // O(n)

int main() {
  n = (int)strlen(gets(T)); // read line and immediately compute its length
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp);      // sort: O(n log n) * cmp: O(n) = O(n^2 log n)
  for (i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

When applied to string `T = 'GATAGACA$'`, the simple code above that sorts all suffixes with built-in sorting and string comparison *library* produces the correct Suffix Array = {`8, 7, 5, 3, 1, 6, 4, 0, 2`}. However, this is barely useful except for contest problems with $n \leq 1000$. The overall runtime of this algorithm is $O(n^2 \log n)$ because the `strcmp` operation that is used to determine the order of two (possibly long) suffixes is too costly, up to $O(n)$ per one pair of suffix comparison.

A *better way* to construct Suffix Array is to sort the *ranking pairs* (small integers) of suffixes in $O(\log_2 n)$ iterations from $k = 1, 2, 4, \ldots$, the last **power of 2** that is less than n. At each iteration, this construction algorithm sorts the suffixes based on the ranking pair `(RA[SA[i]], RA[SA[i]+k])` of suffix `SA[i]`. This algorithm is based on the discussion in [68]. An example execution is shown below for `T = 'GATAGACA$'` and $n = 9$.

- First, `SA[i] = i` and `RA[i]` = ASCII value of `T[i]` $\forall$`i` $\in$ `[0..n-1]` (Table 6.1—left). At iteration $k = 1$, the ranking pair of suffix `SA[i]` is `(RA[SA[i]], RA[SA[i]+1])`.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA$	71 (G)	65 (A)
1	1	ATAGACA$	65 (A)	84 (T)
2	2	TAGACA$	84 (T)	65 (A)
3	3	AGACA$	65 (A)	71 (G)
4	4	GACA$	71 (G)	65 (A)
5	5	ACA$	65 (A)	67 (C)
6	6	CA$	67 (C)	65 (A)
7	7	A$	65 (A)	36 ($)
8	8	$	36 ($)	00 (-)

Initial ranks RA[i] = ASCII value of T[i]
$ = 36, A = 65, C = 67, G = 71, T = 84

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	8	$	36 ($)	00 (-)
1	7	A$	65 (A)	36 ($)
2	5	ACA$	65 (A)	67 (C)
3	3	AGACA$	65 (A)	71 (G)
4	1	ATAGACA$	65 (A)	84 (T)
5	6	CA$	67 (C)	65 (A)
6	0	GATAGACA$	71 (G)	65 (A)
7	4	GACA$	71 (G)	65 (A)
8	2	TAGACA$	84 (T)	65 (A)

If SA[i] + k >= n (beyond the length of string T),
we give a default rank 0 with label -

Table 6.1: L/R: Before/After Sorting; k = 1; the initial sorted order appears

Example 1: The rank of suffix 5 'ACA$' is ('A', 'C') = (65, 67).

Example 2: The rank of suffix 3 'AGACA$' is ('A', 'G') = (65, 71).

After we sort these ranking pairs, the order of suffixes is now like Table 6.1—right, where suffix 5 'ACA$' comes before suffix 3 'AGACA$', etc.

- At iteration $k = 2$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+2]). This ranking pair is now obtained by looking at the first pair and the second pair of characters only. To get the new ranking pairs, we do not have to recompute many things. We set the first one, i.e. Suffix 8 '$' to have new rank $r = 0$. Then, we iterate from i = [1..n-1]. If the ranking pair of suffix SA[i] is different from the ranking pair of the previous suffix SA[i-1] in sorted order, we increase the rank $r = r + 1$. Otherwise, the rank stays at r (see Table 6.2—left).

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	0	GATAGACA$	6 (GA)	7 (TA)
7	4	GACA$	6 (GA)	5 (CA)
8	2	TAGACA$	7 (TA)	6 (GA)

$- (first item) is given rank 0, then for i = 1 to n-1, compare rank pair of this row with previous row

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	4	GACA$	6 (GA)	5 (CA)
7	0	GATAGACA$	6 (GA)	7 (TA)
8	2	TAGACA$	7 (TA)	6 (GA)

If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label -

Table 6.2: L/R: Before/After Sorting; k = 2; 'GATAGACA' and 'GACA' are swapped

Example 1: In Table 6.1—right, the ranking pair of suffix 7 'A$' is (65, 36) which is different with the ranking pair of previous suffix 8 '$-' which is (36, 0). Therefore in Table 6.2—left, suffix 7 has a new rank 1.

Example 2: In Table 6.1—right, the ranking pair of suffix 4 'GACA$' is (71, 65) which is similar with the ranking pair of previous suffix 0 'GATAGACA$' which is also (71, 65). Therefore in Table 6.2—left, since suffix 0 is given a new rank 6, then suffix 4 is also given the same new rank 6.

Once we have updated RA[SA[i]] $\forall$i $\in$ [0..n-1], the value of RA[SA[i]+k] can be easily determined too. In our explanation, if SA[i]+k $\geq$ n, we give a default rank 0. See **Exercise 6.6.4.2*** for more details on the implementation aspect of this step.

At this stage, the ranking pair of suffix 0 'GATAGACA$' is (6, 7) and suffix 4 'GACA$' is (6, 5). These two suffixes are still not in sorted order whereas all the other suffixes are already in their correct order. After another round of sorting, the order of suffixes is now like Table 6.2—right.

- At iteration $k = 4$, the ranking pair of suffix SA[i] is (RA[SA[i]], RA[SA[i]+4]). This ranking pair is now obtained by looking at the first quadruple and the second quadruple of characters only. Now, notice that the previous ranking pairs of Suffix 4 (6, 5) and Suffix 0 (6, 7) in Table 6.2—right are now different. Therefore, after re-ranking, all n suffixes in Table 6.3 now have different ranking. This can be easily verified by checking if RA[SA[n-1]] == n-1. When this happens, we have successfully obtained the Suffix Array. Notice that the major sorting work is done in the first few iterations only and we usually do not need many iterations.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	8	$	0 ($---)	0 (----)
1	7	A$	1 (A$--)	0 (----)
2	5	ACA$	2 (ACA$)	0 (----)
3	3	AGACA$	3 (AGAC)	1 (A$--)
4	1	ATAGACA$	4 (ATAG)	2 (ACA$)
5	6	CA$	5 (CA$-)	0 (----)
6	4	GACA$	6 (GACA)	0 ($---)
7	0	GATAGACA$	7 (GATA)	6 (GACA)
8	2	TAGACA$	8 (TAGA)	5 (CA$-)

Now all suffixes have different ranking
We are done

Table 6.3: Before/After sorting; k = 4; no change

This Suffix Array construction algorithm can be new for most readers of this book. Therefore in the third edition of this book, we have added a Suffix Array visualization tool to show the steps of of any (but relatively short) input string `T` specified by the reader themselves. Several Suffix Array applications shown in the next Section 6.6.5 are also included.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/suffixarray.html`

We can implement the sorting of ranking pairs above using (built-in) $O(n \log n)$ sorting library. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n \log n) = O(n \log^2 n)$. With this time complexity, we can now work with strings of length up to $\approx 10K$. However, since the sorting process only sort *pair of small integers*, we can use a *linear time* two-pass Radix Sort (that internally calls Counting Sort—see more details in Section 9.32) to reduce the sorting time to $O(n)$. As we repeat the sorting process up to $\log n$ times, the overall time complexity is $O(\log n \times n) = O(n \log n)$. Now, we can work with strings of length up to $\approx 100K$—typical programming contest range.

We provide our $O(n \log n)$ implementation below. Please scrutinize the code to understand how it works. For ICPC contestants only: As you can bring hard copy materials to the contest, it is a good idea to put this code in your team's library.

```
#define MAX_N 100010                              // second approach: O(n log n)
char T[MAX_N];                       // the input string, up to 100K characters
int n;                                            // the length of input string
int RA[MAX_N], tempRA[MAX_N];         // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];     // suffix array and temporary suffix array
int c[MAX_N];                                        // for counting/radix sort

void countingSort(int k) {                                                // O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                            // clear frequency table
  for (i = 0; i < n; i++)       // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t; }
  for (i = 0; i < n; i++)           // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                       // update the suffix array SA
    SA[i] = tempSA[i];
}
```

```
void constructSA() {         // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                 // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {       // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);          // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                     // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                      // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }

int main() {
  n = (int)strlen(gets(T));       // input T as per normal, without the '$'
  T[n++] = '$';                                 // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Exercise 6.6.4.1*: Show the steps to compute the Suffix Array of `T = 'COMPETITIVE$'` with $n = 12$! How many sorting iterations that you need to get the Suffix Array?
Hint: Use the Suffix Array visualization tool shown above.

Exercise 6.6.4.2*: In the suffix array code shown above, will the following line:

```
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
```

causes index out of bound in some cases?
That is, will `SA[i]+k` or `SA[i-1]+k` ever be $\geq$ `n` and crash the program? Explain!

Exercise 6.6.4.3*: Will the suffix array code shown above works if the input string `T` contains a space (ASCII value = 32) inside? Hint: The default terminating character used—i.e. `'$'`—has ASCII value = 36.

6.6.5 Applications of Suffix Array

We have mentioned earlier that Suffix Array is closely related to Suffix Tree. In this subsection, we show that with Suffix Array (which is easier to construct), we can solve the string processing problems shown in Section 6.6.3 that are solvable using Suffix Tree.

String Matching in $O(m \log n)$

After we obtain the Suffix Array of `T`, we can search for a pattern string `P` (of length m) in `T` (of length n) in $O(m \log n)$. This is a factor of $\log n$ times slower than the Suffix Tree version but in practice is quite acceptable. The $O(m \log n)$ complexity comes from the fact that we can do two $O(\log n)$ binary searches on sorted suffixes and do up to $O(m)$ suffix

comparisons[16]. The first/second binary search is to find the lower/upper bound respectively. This lower/upper bound is the the smallest/largest `i` such that the prefix of suffix `SA[i]` matches the pattern string `P`, respectively. All the suffixes between the lower and upper bound are the occurrences of pattern string `P` in `T`. Our implementation is shown below:

```
ii stringMatching() {                          // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;                  // valid matching = [0..n-1]
  while (lo < hi) {                                          // find lower bound
    mid = (lo + hi) / 2;                                  // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;         // prune upper half (notice the >= sign)
    else          lo = mid + 1;           // prune lower half including mid
  }                                       // observe '=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {            // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                                  // prune upper half
    else         lo = mid + 1;            // prune lower half including mid
  }                            // (notice the selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;                 // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair, respectively

int main() {
  n = (int)strlen(gets(T));       // input T as per normal, without the '$'
  T[n++] = '$';                                  // add terminating character
  constructSA();
  for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);

  while (m = (int)strlen(gets(P)), m) {     // stop if P is an empty string
    ii pos = stringMatching();
    if (pos.first != -1 && pos.second != -1) {
      printf("%s found, SA [%d..%d] of %s\n", P, pos.first, pos.second, T);
      printf("They are:\n");
      for (int i = pos.first; i <= pos.second; i++)
        printf("  %s\n", T + SA[i]);
    } else printf("%s is not found in %s\n", P, T);
} } // return 0;
```

A sample execution of this string matching algorithm on the Suffix Array of `T = 'GATAGACA$'` with `P = 'GA'` is shown in Table 6.4 below.

We start by finding the lower bound. The current range is `i = [0..8]` and thus the middle one is `i = 4`. We compare the first two characters of suffix `SA[4]`, which is '<u>AT</u>AGACA$', with `P = 'GA'`. As `P = 'GA'` is larger, we continue exploring `i = [5..8]`. Next, we compare the first two characters of suffix `SA[6]`, which is '<u>GA</u>CA$', with `P = 'GA'`. It is a match.

[16]This is achievable by using the `strncmp` function to compare only the first m characters of both suffixes.

As we are currently looking for the *lower* bound, we do not stop here but continue exploring `i = [5..6]`. `P = 'GA'` is larger than suffix `SA[5]`, which is '`CA$`'. We stop here. Index `i = 6` is the lower bound, i.e. suffix `SA[6]`, which is '`GACA$`', is the *first* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.

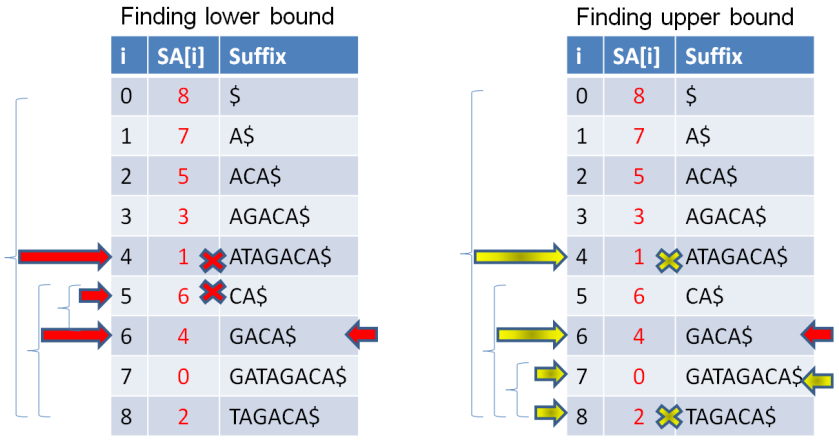


Table 6.4: String Matching using Suffix Array

Next, we search for the upper bound. The first step is the same as above. But at the second step, we have a match between suffix `SA[6]`, which is '`GACA$`', with `P = 'GA'`. Since now we are looking for the *upper* bound, we continue exploring `i = [7..8]`. We find another match when comparing suffix `SA[7]`, which is '`GATAGACA$`', with `P = 'GA'`. We stop here. This `i = 7` is the upper bound in this example, i.e. suffix `SA[7]`, which is '`GATAGACA$`', is the *last* time pattern `P = 'GA'` appears as a prefix of a suffix in the list of sorted suffixes.

Finding the Longest Common Prefix in $O(n)$

Given the Suffix Array of `T`, we can compute the Longest Common Prefix (LCP) between *consecutive* suffixes in Suffix Array order. By definition, `LCP[0] = 0` as suffix `SA[0]` is the first suffix in Suffix Array order without any other suffix preceding it. For `i > 0, LCP[i]` = the length of common prefix between suffix `SA[i]` and suffix `SA[i-1]`. See Table 6.5—left. We can compute LCP directly by definition by using the code below. However, this approach is slow as it can increase the value of L up to $O(n^2)$ times. This defeats the purpose of building Suffix Array in $O(n \log n)$ time as shown in Section 6.8.

```
void computeLCP_slow() {
  LCP[0] = 0;                                          // default value
  for (int i = 1; i < n; i++) {                 // compute LCP by definition
    int L = 0;                                         // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;        // same L-th char, L++
    LCP[i] = L;
} }
```

A better solution using the Permuted Longest-Common-Prefix (PLCP) theorem [37] is described below. The idea is simple: It is *easier* to compute the LCP in the original position order of the suffixes instead of the lexicographic order of the suffixes. In Table 6.5—right, we have the original position order of the suffixes of `T = 'GATAGACA$'`. Observe that column `PLCP[i]` forms a pattern: Decrease-by-1 block ($2 \rightarrow 1 \rightarrow 0$); increase to 1; decrease-by-1 block again ($1 \rightarrow 0$); increase to 1 again; decrease-by-1 block again ($1 \rightarrow 0$), etc.

i	SA[i]	LCP[i]	Suffix	i	Phi[i]	PLCP[i]	Suffix
0	8	0	$	0	4	2	GATAGACA$
1	7	0	A$	1	3	1	ATAGACA$
2	5	1	ACA$	2	0	0	TAGACA$
3	3	1	AGACA$	3	5	1	AGACA$
4	1	1	ATAGACA$	4	6	0	GACA$
5	6	0	CA$	5	7	1	ACA$
6	4	0	GACA$	6	1	0	CA$
7	0	2	GATAGACA$	7	8	0	A$
8	2	0	TAGACA$	8	-1	0	$

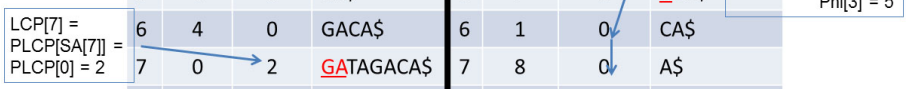


Table 6.5: Computing the LCP given the SA of `T = 'GATAGACA$'`

The PLCP theorem says that the total number of increase (and decrease) operations is at most $O(n)$. This pattern and this $O(n)$ guarantee are exploited in the code below.

First, we compute `Phi[i]`, that stores the suffix index of the previous suffix of suffix `SA[i]` in Suffix Array order. By definition, `Phi[SA[0]] = -1`, i.e. there is no previous suffix that precede suffix `SA[0]`. Take some time to verify the correctness of column `Phi[i]` in Table 6.5—right. For example, `Phi[SA[3]] = SA[3-1]`, so `Phi[3] = SA[2] = 5`.

Now, with `Phi[i]`, we can compute the permuted LCP. The first few steps of this algorithm is elaborated below. When `i = 0`, we have `Phi[0] = 4`. This means suffix 0 '`GATAGACA$`' has suffix 4 '`GACA$`' before it in Suffix Array order. The first two characters (`L = 2`) of these two suffixes match, so `PLCP[0] = 2`.

When `i = 1`, we know that *at least* `L-1 = 1` characters can match as the next suffix in position order will have one less starting character than the current suffix. We have `Phi[1] = 3`. This means suffix 1 '`ATAGACA$`' has suffix 3 '`AGACA$`' before it in Suffix Array order. Observe that these two suffixes indeed have at least 1 character match (that is, we do not start from `L = 0` as in `computeLCP_slow()` function shown earlier and therefore this is more efficient). As we cannot extend this further, we have `PLCP[1] = 1`.

We continue this process until `i = n-1`, bypassing the case when `Phi[i] = -1`. As the PLCP theorem says that L will be increased/decreased at most n times, this part runs in amortized $O(n)$. Finally, once we have the PLCP array, we can put the permuted LCP back to the correct position. The code is relatively short, as shown below.

```
void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                        // default value
  for (i = 1; i < n; i++)                                 // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {                 // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }             // special case
    while (T[i + L] == T[Phi[i] + L]) L++;       // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                                       // L decreased max n times
  }
  for (i = 0; i < n; i++)                                 // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];   // put the permuted LCP to the correct position
}
```

Finding the Longest Repeated Substring in $O(n)$

If we have computed the Suffix Array in $O(n \log n)$ and the LCP between consecutive suffixes in Suffix Array order in $O(n)$, then we can determine the length of the Longest Repeated Substring (LRS) of `T` in $O(n)$.

The length of the longest repeated substring is just the highest number in the LCP array. In Table 6.5—left that corresponds to the Suffix Array and the LCP of `T = 'GATAGACA$'`, the highest number is 2 at index `i = 7`. The first 2 characters of the corresponding suffix `SA[7]` (suffix `0`) is `'GA'`. This is the longest repeated substring in `T`.

Finding the Longest Common Substring in $O(n)$

i	SA[i]	LCP[i]	Owner	Suffix
0	13	0	2	#
1	8	0	1	$CATA#
2	12	0	2	A#
3	7	1	1	A$CATA#
4	5	1	1	ACA$CATA#
5	3	1	1	AGACA$CATA#
6	10	1	2	ATA#
7	1	3	1	ATAGACA$CATA#
8	6	0	1	CA$CATA#
9	9	2	2	CATA#
10	4	0	1	GACA$CATA#
11	0	2	1	GATAGACA$CATA#
12	11	0	2	TA#
13	2	2	1	TAGACA$CATA#

Table 6.6: The Suffix Array, LCP, and owner of `T = 'GATAGACA$CATA#'`

Without loss of generality, let's consider the case with only *two* strings. We use the same example as in the Suffix Tree section earlier: T_1 `= 'GATAGACA$'` and T_2 `= 'CATA#'`. To solve the LCS problem using Suffix Array, first we have to concatenate both strings (note that the terminating characters of both strings *must be different*) to produce `T = 'GATAGACA$CATA#'`. Then, we compute the Suffix and LCP array of `T` as shown in Figure 6.6.

Then, we go through consecutive suffixes in $O(n)$. If two consecutive suffixes belong to different owner (can be easily checked[17], for example we can test if suffix `SA[i]` belongs to T_1 by testing if `SA[i]` $<$ the length of T_1), we look at the LCP array and see if the maximum LCP found so far can be increased. After one $O(n)$ pass, we will be able to determine the Longest Common Substring. In Figure 6.6, this happens when `i = 7`, as suffix `SA[7]` = suffix `1` = `'ATAGACA$CATA#'` (owned by T_1) and its previous suffix `SA[6]` = suffix `10` = `'ATA#'` (owned by T_2) have a common prefix of length 3 which is `'ATA'`. This is the LCS.

We close this section and this chapter by highlighting the availability of our source code. Please spend some time understanding the source code which may not be trivial for those who are new with Suffix Array.

Source code: `ch6_04_sa.cpp/java`

[17]With three or more strings, this check will have more 'if statements'.

Exercise 6.6.5.1*: Suggest some possible improvements to the `stringMatching()` function shown in this section!

Exercise 6.6.5.2*: Compare the KMP algorithm shown in Section 6.4 with the string matching feature of Suffix Array. When it is more beneficial to use Suffix Array to deal with string matching and when it is more beneficial to just use KMP or standard string libraries?

Exercise 6.6.5.3*: Solve all exercises on Suffix Tree applications, i.e. **Exercise 6.6.3.1, 2, 3*, 4, 5*, and 6*** using Suffix Array instead!

Programming Exercises related to Suffix Array[18]:

1. UVa 00719 - Glass Beads (min lexicographic rotation[19]; $O(n \log n)$ build SA)
2. **UVa 00760 - DNA Sequencing *** (Longest Common *Substring* of two strings)
3. UVa 01223 - Editor (LA 3901, Seoul07, Longest Repeated Substring (or KMP))
4. *UVa 01254 - Top 10* (LA 4657, Jakarta09, Suffix Array + Segment Tree)
5. **UVa 11107 - Life Forms *** (Longest Common Substring of $> \frac{1}{2}$ of the strings)
6. **UVa 11512 - GATTACA *** (Longest Repeated Substring)
7. SPOJ 6409 - Suffix Array (problem author: Felix Halim)
8. IOI 2008 - Type Printer (DFS traversal of Suffix Trie)

[18] You can try solving these problems with Suffix Tree, but you have to learn how to code the Suffix Tree construction algorithm by yourself. The programming problems listed here are solvable with Suffix Array. Also please take note that our sample code uses `gets` for reading the input strings. If you use `scanf(``%s'')` or `getline`, do not forget to adjust the potential DOS/UNIX 'end of line' differences.

[19] This problem can be solved by concatenating the string with itself, build the Suffix Array, then find the first suffix in Suffix Array sorted order that has length greater or equal to n.

6.7 Solution to Non-Starred Exercises

C Solutions for Section 6.2

Exercise 6.2.1:

(a) A string is stored as an array of characters terminated by null, e.g. `char str[30x10+50], line[30+50];`. It is a good practice to declare array size slightly bigger than requirement to avoid "off by one" bug.

(b) To read the input line by line, we use[20] `gets(line);` or `fgets(line, 40, stdin);` in `string.h` (or `cstring`) library. Note that `scanf(``%s'', line)` is not suitable here as it will only read the first word.

(c) We first set `strcpy(str, ``'');`, and then we combine the `line`s that we read into a longer string using `strcat(str, line);`. If the current line is not the last one, we append a space to the back of `str` using `strcat(str, `` '');` so that the last word from this line is not accidentally combined with the first word of the next line.

(d) We stop reading the input when `strncmp(line, ``.......'', 7) == 0`. Note that `strncmp` only compares the first n characters.

Exercise 6.2.2:

(a) For finding a substring in a relatively short string (the standard string matching problem), we can just use library function. We can use `p = strstr(str, substr);`
The value of `p` will be `NULL` if `substr` is not found in `str`.

(b) If there are multiple copies of `substr` in `str`, we can use `p = strstr(str + pos, substr)`. Initially `pos = 0`, i.e. we search from the first character of `str`. After finding one occurrence of `substr` in `str`, we can call `p = strstr(str + pos, substr)` again where this time `pos` is the index of the current occurrence of `substr` in `str` *plus one* so that we can get the next occurrence. We repeat this process until `p == NULL`. This C solution requires understanding of the memory address of a C array.

Exercise 6.2.3: In many string processing tasks, we are required to iterate through every characters in `str` once. If there are n characters in `str`, then such scan requires $O(n)$. In both C/C++, we can use `tolower(ch)` and `toupper(ch)` in `ctype.h` to convert a character to its lower and uppercase version. There are also `isalpha(ch)`/`isdigit(ch)` to check whether a given character is alphabet `[A-Za-z]`/digit, respectively. To test whether a character is a vowel, one method is to prepare a string `vowel = "aeiou";` and check if the given character is one of the five characters in `vowel`. To check whether a character is a consonant, simply check if it is an alphabet but not a vowel.

Exercise 6.2.4: Combined C and C++ solutions:

(a) One of the easiest ways to tokenize a string is to use `strtok(str, delimiters);` in C.

(b) These tokens can then be stored in a C++ `vector<string> tokens`.

(c) We can use C++ STL `algorithm::sort` to sort `vector<string> tokens`. When needed, we can convert C++ `string` back to C string by using `str.c_str()`.

[20]Note: Function `gets` is actually unsafe because it does not perform bounds checking on input size.

Exercise 6.2.5: See the C++ solution.

Exercise 6.2.6: Read the input character by character and count incrementally, look for the presence of '`\n`' that signals the end of a line. Pre-allocating a fixed-sized buffer is not a good idea as the problem author can set a ridiculously long string to break your code.

C++ Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `string`.

(b) We can use `cin.getline()` in `string` library.

(c) We can use the '+' operator directly to concatenate strings.

(d) We can use the '==' operator directly to compare two strings.

Exercise 6.2.2:

(a) We can use function `find` in class `string`.

(b) Same idea as in C language. We can set the offset value in the second parameter of function `find` in class `string`.

Exercise 6.2.3-4: Same solutions as in C language.

Exercise 6.2.5: We can use C++ STL `map<string, int>` to keep track the frequency of each word. Every time we encounter a new token (which is a string), we increase the corresponding frequency of that token by one. Finally, we scan through all tokens and determine the one with the highest frequency.

Exercise 6.2.6: Same solution as in C language.

Java Solutions for Section 6.2

Exercise 6.2.1:

(a) We can use class `String`, `StringBuffer`, or `StringBuilder` (this one is faster than `StringBuffer`).

(b) We can use the `nextLine` method in Java `Scanner`. For faster I/O, we can consider using the `readLine` method in Java `BufferedReader`.

(c) We can use the `append` method in `StringBuilder`. We should not concatenate Java Strings with the '+' operator as Java String class is immutable and thus such operation is (very) costly.

(d) We can use the `equals` method in Java `String`.

Exercise 6.2.2:

(a) We can use the `indexOf` method in class `String`.

(b) Same idea as in C language. We can set the offset value in the second parameter of `indexOf` method in class `String`.

Exercise 6.2.3: Use Java `StringBuilder` and `Character` classes for these operations.

Exercise 6.2.4:

(a) We can use Java `StringTokenizer` class or the `split` method in Java `String` class.

(b) We can use Java `Vector` of `String`s.

(c) We can use Java `Collections.sort`.

Exercise 6.2.5: Same idea as in C++ language.
We can use Java `TreeMap<String, Integer>`.

Exercise 6.2.6: We need to use the `read` method in Java `BufferedReader` class.

Solutions for the Other Sections

Exercise 6.5.1.1: Different scoring scheme will yield different (global) alignment. If given a string alignment problem, read the problem statement and see what is the required cost for match, mismatch, insert, and delete. Adapt the algorithm accordingly.

Exercise 6.5.1.2: You have to save the predecessor information (the arrows) during the DP computation. Then follow the arrows using recursive backtracking. See Section 3.5.1.

Exercise 6.5.1.3: The DP solution only need to refer to previous row so it can utilize the 'space saving trick' by just using two rows, the current row and the previous row. The new space complexity is just $O(min(n, m))$, that is, put the string with the lesser length as string 2 so that each row has lesser columns (less memory). The time complexity of this solution is still $O(nm)$. The only drawback of this approach, as with any other space saving trick is that we will not be able to reconstruct the optimal solution. So if the actual optimal solution is needed, we cannot use this space saving trick. See Section 3.5.1.

Exercise 6.5.1.4: Simply concentrate along the main diagonal with width d. We can speed up Needleman-Wunsch's algorithm to $O(dn)$ by doing this.

Exercise 6.5.1.5: It involves Kadane's algorithm again (see maximum sum problem in Section 3.5.2).

Exercise 6.5.2.1: `'pple'`.

Exercise 6.5.2.2: Set score for match = 0, mismatch = 1, insert and delete = negative infinity. However, this solution is not efficient and not natural, as we can simply use an $O(min(n, m))$ algorithm to scan both string 1 and string 2 and count how many characters are different.

Exercise 6.5.2.3: Reduced to LIS, $O(n \log k)$ solution. The reduction to LIS is not shown. Draw it and see how to reduce this problem into LIS.

Exercise 6.6.3.1: 'CA' is found, 'CAT' is not.

Exercise 6.6.3.2: 'ACATTA'.

Exercise 6.6.3.4: 'EVEN'.

6.8 Chapter Notes

The material about String Alignment (Edit Distance), Longest Common Subsequence, and Suffix Trie/Tree/Array are originally from **A/P Sung Wing Kin, Ken** [62], School of Computing, National University of Singapore. The material have since evolved from a more theoretical style into the current competitive programming style.

The section about basic string processing skills (Section 6.2) and the Ad Hoc string processing problems were born from our experience with string-related problems and techniques. The number of programming exercises mentioned there is about three quarters of all other string processing problems discussed in this chapter. We are aware that these are not the typical ICPC problems/IOI tasks, but they are still good programming exercises to improve your programming skills.

In Section 6.4, we discuss the library solutions and one fast algorithm (Knuth-Morris-Pratt/KMP algorithm) for the String Matching problem. The KMP implementation will be useful if you have to modify basic string matching requirement yet you still need fast performance. We believe KMP is fast enough for finding pattern string in a long string for typical contest problems. Through experimentation, we conclude that the KMP implementation shown in this book is slightly faster than the built-in `C strstr`, `C++ string.find` and `Java String.indexOf`. If an even faster string matching algorithm is needed during contest time for one longer string and much more queries, we suggest using Suffix Array discussed in Section 6.8. There are several other string matching algorithms that are not discussed yet like **Boyer-Moore's**, **Rabin-Karp's**, **Aho-Corasick's**, **Finite State Automata**, etc. Interested readers are welcome to explore them.

We have expanded the discussion of *non classical* DP problems involving string in Section 6.5. We feel that the classical ones will be rarely asked in modern programming contests.

The practical implementation of Suffix Array (Section 6.6) is inspired mainly from the article "Suffix arrays - a programming contest approach" by [68]. We have integrated and synchronized many examples given there with our way of writing Suffix Array implementation. In the third edition of this book, we have (re-)introduced the concept of terminating character in Suffix Tree and Suffix Array as it simplifies the discussion. It is a good idea to solve *all* the programming exercises listed in Section 6.6 although they are not that many *yet*. This is an important data structure that will be more popular in the near future.

Compared to the first two editions of this book, this chapter has grown even more—similar case as with Chapter 5. However, there are several other string processing problems that we have not touched yet: **Hashing Techniques** for solving some string processing problems, the **Shortest Common Superstring** problem, **Burrows-Wheeler transformation** algorithm, **Suffix Automaton**, **Radix Tree**, etc.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	10	24 (+140%)	35 (+46%)
Written Exercises	4	24 (+500%)	17+16* = 33 (+38%)
Programming Exercises	54	129 (+138%)	164 (+27%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
6.3	**Ad Hoc Strings Problems**	126	77%	8%
6.4	String Matching	13	8%	1%
6.5	**String Processing with DP**	17	10%	1%
6.6	Suffix Trie/Tree/Array	8	5%	≈ 1%


COACH
L-R: Dr Bill Poucher, Steven

Chapter 7

(Computational) Geometry

Let no man ignorant of geometry enter here.
— **Plato's Academy in Athens**

7.1 Overview and Motivation

(Computational[1]) Geometry is yet another topic that frequently appears in programming contests. Almost all ICPC problem sets have *at least one* geometry problem. If you are lucky, it will ask you for some geometry solution that you have learned before. Usually you draw the geometrical object(s) and then derive the solution from some basic geometric formulas. However, many geometry problems are the *computational* ones that require some complex algorithm(s).

In IOI, the existence of geometry-specific problems depends on the tasks chosen by the Scientific Committee that year. In recent years (2009-2012), IOI tasks do not feature *pure* geometry-specific problems. However, in the earlier years [67], every IOI contain one or two geometry related problems.

We have observed that geometry-related problems are usually not attempted during the early part of the contest time for *strategic reason* because the solutions for geometry-related problems have *lower* probability of getting Accepted (AC) during contest time compared to the solutions for other problem types in the problem set, e.g. Complete Search or Dynamic Programming problems. The typical issues with geometry problems are as follow:

- Many geometry problems have one and usually several tricky 'corner test cases', e.g. What if the lines are vertical (infinite gradient)?, What if the points are collinear?, What if the polygon is concave?, What if the convex hull of a set of points is the set of points itself?, etc. Therefore, it is usually a very good idea to test your team's geometry solution with lots of corner test cases before you submit it for judging.
- There is a possibility of having floating point precision errors that cause even a 'correct' algorithm to get a Wrong Answer (WA) response.
- The solutions for geometry problems usually involve *tedious* coding.

These reasons cause many contestants to view that spending precious minutes attempting *other* problem types in the problem set more worthwhile than attempting a geometry problem that has lower probability of acceptance.

[1]We differentiate between *pure* geometry problems and the *computational* geometry ones. Pure geometry problems can normally be solved by hand (pen and paper method). Computational geometry problems typically require running an algorithm using computer to obtain the solution.

However, another not-so-good reason for the lack of attempts for geometry problems is because the contestants are not well prepared.

- The contestants forget some important basic formulas or are unable to derive the required (more complex) formulas from the basic ones.
- The contestants do not prepare well-written library functions before contest and their attempts to code such functions during stressful contest environment end up with one, but usually several[2], bug(s). In ICPC, the top teams usually fill a sizeable part of their hard copy material (which they can bring into the contest room) with lots of geometry formulas and library functions.

The main aim of this chapter is therefore to increase the number of attempts (and also AC solutions) for geometry-related problems in programming contests. Study this chapter for some ideas on tackling (computational) geometry problems in ICPCs and IOIs. There are only two sections in this chapter.

In Section 7.2, we present many (it is impossible to enumerate all) English geometric terminologies[3] and various basic formulas for 0D, 1D, 2D, and 3D **geometry objects** commonly found in programming contests. This section can be used as a quick reference when contestants are given geometry problems and are not sure of certain terminologies or forget some basic formulas.

In Section 7.3, we discuss several algorithms on 2D **polygons**. There are several nice pre-written library routines which can differentiate good from average teams (contestants) like the algorithms for deciding if a polygon is convex or concave, deciding if a point is inside or outside a polygon, cutting a polygon with a straight line, finding the convex hull of a set of points, etc.

The implementations of the formulas and computational geometry algorithms shown in this chapter use the following techniques to increase the probability of acceptance:

1. We highlight the special cases that can potentially arise and/or choose the implementation that reduces the number of such special cases.
2. We try to avoid floating point operations (i.e. division, square root, and any other operations that can produce numerical errors) and work with precise integers whenever possible (i.e. integer additions, subtractions, multiplications).
3. If we really need to work with floating point, we do floating point equality test this way: `fabs(a - b) < EPS` where `EPS` is a small number[4] like `1e-9` instead of testing if `a == b`. When we need to check if a floating point number $x \geq 0.0$, we use `x > -EPS` (similarly to test if $x \leq 0.0$, we use `x < EPS`).

[2] As a reference, the library code on points, lines, circles, triangles, and polygons shown in this chapter require several iterations of bug fixes to ensure that as many (usually subtle) bugs and special cases are handled properly.

[3] ACM ICPC and IOI contestants come from various nationalities and backgrounds. Therefore, we would like to get everyone familiarized with the English geometric terminologies.

[4] Unless otherwise stated, this `1e-9` is the default value of EPS(ilon) that we use in this chapter.

7.2 Basic Geometry Objects with Libraries

7.2.1 0D Objects: Points

1. **Point** is the basic building block of higher dimensional geometry objects. In 2D Euclidean[5] space, points are usually represented with a struct in C/C++ (or Class in Java) with two[6] members: The **x** and **y** coordinates w.r.t origin, i.e. coordinate (0, 0). If the problem description uses integer coordinates, use `int`s; otherwise, use `double`s. In order to be generic, we use the floating-point version of `struct point` in this book. A default and user-defined constructors can be used to (slightly) simplify coding later.

```
// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                      // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };         // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };     // user-defined
```

2. Sometimes we need to sort the points. We can easily do that by overloading the less than operator inside `struct point` and use sorting library.

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                 // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; } };       // second criteria, by y-coordinate

// in int main(), assuming we already have a populated vector<point> P
  sort(P.begin(), P.end());    // comparison operator is defined above
```

3. Sometimes we need to test if two points are equal. We can easily do that by overloading the equal operator inside `struct point`.

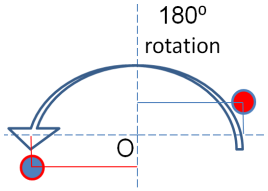
```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

// in int main()
  point P1(0, 0), P2(0, 0), P3(0, 1);
  printf("%d\n", P1 == P2);                                  // true
  printf("%d\n", P1 == P3);                                 // false
```

[5]For simplicity, the 2D and 3D Euclidean spaces are the 2D and 3D world that we encounter in real life.
[6]Add one more member, `z`, if you are working in 3D Euclidean space.

4. We can measure Euclidean distance[7] between two points by using the function below.

```
double dist(point p1, point p2) {                  // Euclidean distance
                      // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double
```

5. We can rotate a point by angle[8] θ counter clockwise around origin (0, 0) by using a rotation matrix:



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 7.1: Rotating point (10, 3) by 180 degrees counter clockwise around origin (0, 0)

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }
```

Exercise 7.2.1.1: Compute the Euclidean distance between point (2, 2) and (6, 5)!

Exercise 7.2.1.2: Rotate a point (10, 3) by 90 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (easy to compute by hand).

Exercise 7.2.1.3: Rotate the same point (10, 3) by 77 degrees counter clockwise around origin. What is the new coordinate of the rotated point? (this time you need to use calculator and the rotation matrix).

7.2.2 1D Objects: Lines

1. **Line** in 2D Euclidean space is the set of points whose coordinates satisfy a given linear equation $ax + by + c = 0$. Subsequent functions in this subsection assume that this linear equation has $b = 1$ for non vertical lines and $b = 0$ for vertical lines unless otherwise stated. Lines are usually represented with a struct in C/C++ (or Class in Java) with three members: The coefficients a, b, and c of that line equation.

```
struct line { double a, b, c; };          // a way to represent a line
```

2. We can compute the required line equation if we are given *at least* two points that pass through that line via the following function.

[7]The Euclidean distance between two points is simply the distance that can be measured with ruler. Algorithmically, it can be found with Pythagorean formula that we will see again in the subsection about triangle below. Here, we simply use a library function.

[8]Humans usually work with degrees, but many mathematical functions in most programming languages (e.g. C/C++/Java) work with radians. To convert an angle from degrees to radians, multiply the angle by $\frac{\pi}{180.0}$. To convert an angle from radians to degrees, multiply the angle with $\frac{180.0}{\pi}$.

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }
```

3. We can test whether two lines are *parallel* by checking if their coefficients a and b are the same. We can further test whether two lines are *the same* by checking if they are parallel and their coefficients c are the same (i.e. all three coefficients a, b, c are the same). Recall that in our implementation, we have fixed the value of coefficient b to 0.0 for all vertical lines and to 1.0 for all *non* vertical lines.

```
bool areParallel(line l1, line l2) {       // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

4. If two lines[9] are not parallel (and therefore also not the same), they will intersect at a point. That intersection point (x, y) can be found by solving the system of two linear algebraic equations[10] with two unknowns: $a_1x + b_1y + c_1 = 0$ and $a_2x + b_2y + c_2 = 0$.

```
// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }
```

5. **Line Segment** is a line with two end points with *finite length*.

6. **Vector**[11] is a line segment (thus it has two end points and length/magnitude) with a *direction*. Usually[12], vectors are represented with a struct in C/C++ (or Class in Java) with two members: The **x** and **y** magnitude of the vector. The magnitude of the vector can be scaled if needed.

7. We can translate (move) a point w.r.t a vector as a vector describes the displacement magnitude in x and y-axis.

[9] To avoid confusion, please differentiate between line intersection versus line *segment* intersection.
[10] See Section 9.9 for the general solution for a system of linear equations.
[11] Do not confuse this with C++ STL `vector` or Java `Vector`.
[12] Another potential design strategy is to merge `struct point` with `struct vec` as they are similar.

```
struct vec { double x, y;  // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
```

8. Given a point p and a line l (described by two points a and b), we can compute the minimum distance from p to l by first computing the location of point c in l that is closest to point p (see Figure 7.2—left) and then obtain the Euclidean distance between p and c. We can view point c as point a translated by a scaled magnitude u of vector ab, or $c = a + u \times ab$. To get u, we do scalar projection of vector ap onto vector ab by using dot product (see the dotted vector $ac = u \times ab$ in Figure 7.2—left). The short implementation of this solution is shown below.

```
double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                  // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c
```

Note that this is not the only way to get the required answer.
Solve **Exercise 7.2.2.10** for the alternative way.

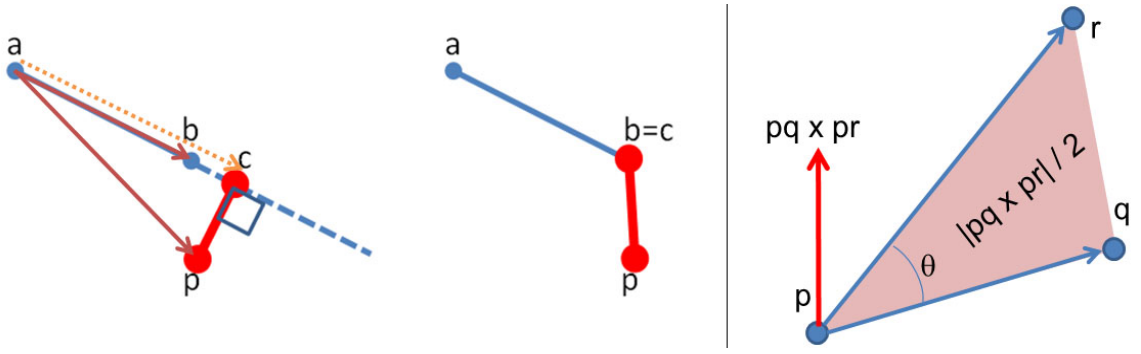


Figure 7.2: Distance to Line (left) and to Line Segment (middle); Cross Product (right)

9. If we are given a line *segment* instead (defined by two *end* points a and b), then the minimum distance from point p to line segment ab must also consider two special cases, the end points a and b of that line segment (see Figure 7.2—middle). The implementation is very similar to `distToLine` function above.

```
// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                    // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                    // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above
```

10. We can compute the angle aob given three points: a, o, and b, using dot product[13]. Since $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, we have $theta = \arccos(oa \cdot ob / (|oa| \times |ob|))$.

```
double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVector(o, a), ob = toVector(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
```

11. Given a line defined by two points p and q, we can determine whether a point r is on the left/right side of the line, or whether the three points p, q, and r are collinear. This can be determined with *cross product*. Let pq and pr be the two vectors obtained from these three points. The cross product $pq \times pr$ result in another vector that is perpendicular to both pq and pr. The magnitude of this vector is equal to the area of the *parallelogram* that the vectors span[14]. If the magnitude is positive/zero/negative, then we know that $p \rightarrow q \rightarrow r$ is a left turn/collinear/right turn, respectively (see Figure 7.2—right). The left turn test is more famously known as the **CCW (Counter Clockwise) Test**.

```
double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }
```

Source code: `ch7_01_points_lines.cpp/java`

[13] `acos` is the C/C++ function name for mathematical function `arccos`.
[14] The area of triangle pqr is therefore *half* of the area of this parallelogram.