PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2022–2023

Schedule 2022–2023

Week	Mon		Wed	
01	20 Sep	Theory 1	21 Sep	Lab 1 (1, 2)
02	26 Sep	—	28 Sep	Theory 2
03	03 Oct	Lab 2 (3, 4)	05 Oct	Theory 3
04	10 Oct	Lab 3 (5, 6)	12 Oct	—
05	17 Oct	Partial exam 1	19 Oct	Exam review
06	24 Oct	Theory 4	26 Oct	Lab 4 (7, 8)
07	31 Oct	—	02 Nov	Lab 5 (9, 10)
08	07 Nov	Theory 5	09 Nov	Lab 6 (11, 12)
09	14 Nov	Theory 6	16 Nov	Lab 7 (13, 14)
10	21 Nov	Partial exam 2	23 Nov	Exam review

25%	Partial exam 1	17 Oct
25%	Partial exam 2	21 Nov
50%	Final exam	09 Dec

100%	Recovery exam	09 Jan

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T1. Structured pseudocode

L1. Problem 1 (X66236, Gene finding)

L1. Problem 2 (X48860, Gene enumeration)

T2. Elementary algorithms and data structures

L2. Problem 3 (X95757, RNA to protein)

L2. Problem 4 (X28783, Read abundance)

T3. Analysis of iterative algorithms

L3. Problem 5 (X89781, Read mapping 1)

L3. Problem 6 (X38913, Read mapping 2)

T4. Linear recursion

L4. Problem 7 (X38951, Sliding window 1)

L4. Problem 8 (X99827, Sliding window 2)

L5. Problem 9 (X11585, Words 1)

L5. Problem 10 (X86108, Words 2)

T5. Multiple recursion

L6. Problem 11 (X85229, Subwords 1)

L6. Problem 12 (X43423, Subwords 2)

T6. Analysis of recursive algorithms

L7. Problem 13 (X48980, Permutations 1)

L7. Problem 14 (X87168, Permutations 2)

G. Valiente. *Algorithms on Trees and Graphs*. Springer, Cham, Switzerland, 2nd edition, 2021

Some of the basic data structures needed for the description of algorithms are illustrated next using a fragment of pseudocode.

Pseudocode conventions follow modern programming guidelines, such as avoiding global side effects (with the only exception of object attributes, which are hidden behind dot-notation) and the unconditional transfer of control by way of goto or gosub statements.

- Assignment of value a to variable x is denoted by $x = a$.
- Comparison of equality between the values of two variables x and y is denoted by $x = y$, comparison of strict inequality is denoted by either $x \neq y$, $x < y$, or $x > y$, and comparison of non-strict inequality is denoted by either $x \leqslant y$ or $x \geqslant y$.
- Logical true and false are denoted by **true** and **false**, respectively.
- Logical negation, conjunction, and disjunction are denoted by **not**, **and**, and **or**, respectively.
- Non-existence is denoted by **nil**.

- Mathematical notation is preferred over programming notation. For example, the cardinality of a set S is denoted by $|S|$, membership of an element x in a set S is denoted by $x \in S$, insertion of an element x into a set S is denoted by $S = S \cup \{x\}$, and deletion of an element x from a set S is denoted by $S = S \setminus \{x\}$.
- Control structures use the following reserved words: **all**, **break**, **do**, **else**, **for**, **if**, **repeat**, **return**, **then**, **to**, **until**, **while**.
- Blocks of statements are shown by means of indention.

```
if ... then
    ...
else
    ...
```

```
for all ... do
    ...
    ...
return ...
```

```
while ... do
    ...
    if ... then
        ...
```

```
repeat
    ...
    ...
until ...
```

The collection of abstract operations on arrays, matrices, lists, stacks, queues, priority queues, sets, and dictionaries are presented next by way of examples.

An array is a one-dimensional array indexed by non-negative integers. The $[i]$ operation returns the i-th element of the array, assuming there is such an element.

```
let A[1..n] be a new array
for i = 1 to n do
    A[i] = false
```

A matrix is a two-dimensional array indexed by non-negative integers. The $[i, j]$ operation returns the element in the i-th row and the j-th column of the matrix, assuming there is such an element.

```
let M[1..m][1..n] be a new matrix
for i = 1 to m do
    for j = 1 to n do
        A[i, j] = 0
```

A list is just a sequence of elements. The **front** operation returns the first element and the **back** operation returns the last element in the list, assuming the list is not empty. The **prev** operation returns the element before a given element in the list, assuming the given element is not at the front of the list. The **next** operation returns the element after a given element in the list, assuming the given element is not at the back of the list. The **append** operation inserts an element at the rear of the list. The **concatenate** operation deletes the elements of another list and inserts them at the rear of the list.

let L be an empty list
append x to L
let L' be an empty list
append x to L'
concatenate L' to L

let x be the element at the front of L
while $x \neq nil$ **do**
    output x
    $x = L.next(x)$
let x be the element at the back of L
while $x \neq nil$ **do**
    output x
    $x = L.prev(x)$

A stack is a sequence of elements which are inserted and deleted at the same end (the top) of the sequence. The **top** operation returns the top element in the stack, assuming the stack is not empty. The **pop** operation deletes and returns the top element in the stack, also assuming the stack is not empty. The **push** operation inserts an element at the top of the stack.

```
let S be an empty stack
push x onto S
let x be the element at the top of S
output x
while S is not empty do
    pop from S the top element x
    output x
```

A queue is a sequence of elements which are inserted at one end (the rear) and deleted at the other end (the front) of the sequence. The **front** operation returns the front element in the queue, assuming the queue is not empty. The **dequeue** operation deletes and returns the front element in the queue, assuming the queue is not empty. The **enqueue** operation inserts an element at the rear end of the queue.

let Q be an empty queue
enqueue x into Q
let x be the element at the front of Q
output x
repeat
    dequeue from Q the front element x
    output x
until Q is empty

A priority queue is a queue of elements with both an information and a priority associated with each element, where there is a linear order defined on the priorities. The **front** operation returns an element with the minimum priority, assuming the priority queue is not empty. The **dequeue** operation deletes and returns an element with the minimum priority in the queue, assuming the priority queue is not empty. The **enqueue** operation inserts an element with a given priority in the priority queue.

let Q be an empty priority queue
enqueue (x, y) into Q
let (x, y) be an element x
    with the minimum priority y in Q
output (x, y)
repeat
    dequeue from Q an element x
        with the minimum priority y
    output (x, y)
until Q is empty

A set is just a set of elements. The **insert** operation inserts an element in the set. The **delete** operation deletes an element from the set. The **member** operation returns true if an element belongs to the set and false otherwise.

let S be an empty set
$S = S \cup \{x\}$
for all $x \in S$ **do**
    output x
    delete x from S

A dictionary is an associative container, consisting of a set of elements with both an information and a unique key associated with each element, where there is a linear order defined on the keys and the information associated with an element is retrieved on the basis of its key. The **member** operation returns true if there is an element with a given key in the dictionary, and false otherwise. The **lookup** operation returns the element with a given key in the dictionary, or *nil* if there is no such element. The **insert** operation inserts and returns an element with a given key and a given information in the dictionary, replacing the element (if any) with the given key. The **delete** operation deletes the element with a given key from the dictionary, if there is such an element.

let D be an empty dictionary
$D[x] = y$
for all $x \in D$ **do**
    $y = D[x]$
    output (x, y)
    delete x from D

T1. Structured pseudocode

L1. Problem 1 (X66236, Gene finding)

L1. Problem 2 (X48860, Gene enumeration)

T2. Elementary algorithms and data structures

L2. Problem 3 (X95757, RNA to protein)

L2. Problem 4 (X28783, Read abundance)

T3. Analysis of iterative algorithms

L3. Problem 5 (X89781, Read mapping 1)

L3. Problem 6 (X38913, Read mapping 2)

T4. Linear recursion

L4. Problem 7 (X38951, Sliding window 1)

L4. Problem 8 (X99827, Sliding window 2)

L5. Problem 9 (X11585, Words 1)

L5. Problem 10 (X86108, Words 2)

T5. Multiple recursion

L6. Problem 11 (X85229, Subwords 1)

L6. Problem 12 (X43423, Subwords 2)

T6. Analysis of recursive algorithms

L7. Problem 13 (X48980, Permutations 1)

L7. Problem 14 (X87168, Permutations 2)

Gene finding X66236_en

Recall that in prokaryotic genomes, the sequence coding for a protein occurs as one contiguous open reading frame, and that an open reading frame begins with the start codon ATG (methionine) in most species and ends with a stop codon TAA, TAG, or TGA.

For example, the DNA sequence of Bacteriophage ϕ-X174, which was the first genome to be sequenced, has 117 open reading frames (11 of which are protein coding genes) within a circular single strand of 5,386 nucleotides.

Write code for the gene finding problem. The program must implement and use the GENEFINDING function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is a minimal substring of s (an open reading frame) from a start codon to a stop codon.

Sample input

```
GGTTTCTATGTGGCTAAATACGTTAACAAAAAGTCAGATATGGACCTTGCTGCTAAAGGTCTAGGAGCTAAAGAATGGAA
```

Sample output

```
ATGTGGCTAAATACGTTAACAAAAAGTCAGATATGGACCTTGCTGCTAAAGGTCTAG
```

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:27:10

```
function GENE-FINDING(s)
    n = |s|
    i = 1                                                    start codon
    while i + 2 ⩽ n do
        if s[i, ..., i + 2] = ATG then
            j = i + 3                                        stop codon
            while j + 2 ⩽ n do
                if s[j : j + 2] ∈ {TAA, TAG, TGA} then
                    return s[i : j + 2]                      open reading frame
                j = j + 3
        i = i + 1
    return an empty string
```

```
import sys

def gene_finding(s):
  n = len(s)
  i = 0
  while i + 2 <= n:
    if s[i : i + 3] == 'ATG':
      j = i + 3
      while j + 2 <= n:
        if s[j : j + 3] in {'TAA', 'TAG', 'TGA'}:
          return s[i : j + 3]
        j = j + 3
    i = i + 1
  return ''

s = sys.stdin.readline().strip()
print(gene_finding(s))
```

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

string gene_finding(string s);

int main() {
  string s;
  cin >> s;
  cout << gene_finding(s) << endl;
}
```

```
string gene_finding(string s) {
  set<string> stop = {"TAA", "TAG", "TGA"};
  int n = s.length();
  int i = 0;
  while (i + 2 <= n) {
    if (s.substr(i, 3) == "ATG") {
      int j = i + 3;
      while (j + 2 <= n) {
        if (stop.find(s.substr(j, 3)) != stop.end())
          return s.substr(i, j + 2 - i + 1);
        j = j + 3;
    } }
    i = i + 1;
  }
  return "";
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Gene enumeration — X48860_en

Recall that in prokaryotic genomes, the sequence coding for a protein occurs as one contiguous open reading frame, and that an open reading frame begins with the start codon ATG (methionine) in most species and ends with a stop codon TAA, TAG, or TGA.

For example, the DNA sequence of Bacteriophage ϕ-X174, which was the first genome to be sequenced, has 117 open reading frames (11 of which are protein coding genes) within a circular single strand of 5,386 nucleotides.

Write code for the gene enumeration problem. The program must implement and use the GENE-ENUMERATION function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is all minimal substrings of s (all open reading frames) from a start codon to a stop codon.

Sample input

```
GAGTTTTATCGCTTCCATGACGCAGAAGTTAACACTTTCGGATATTTCTGATGAGTCGAAAAATTATCTTGATAAAGCAGGAATTACTACTGCTTG
```

Sample output

```
ATGACGCAGAAGTTAACACTTTCGGATATTTCTGATGAGTCGAAAAATTATCTTGATAAAGCAGGAATTACTACTGCTTGTTTACGAATTAAATCG
ATGAGTCGAAAAATTATCTTGATAAAGCAGGAATTACTACTGCTTGTTTACGAATTAAATCGAAGTGGACTGCTGGCGGAAAATGAGAAAATTCGA
ATGAGAAAATTCGACCTATCCTTGCGCAGCTCGAGAAGCTCTTACTTTGCGACCTTTCGCCATCAACTAACGATTCTGTCAAAAACTGACGCGTTG
ATGAGGAGAAGTGGCTTAATATGCTTGGCACGTTCGTCAAGGACTGGTTTAGATATGAGTCACATTTTGTTCATGGTAGAGATTCTCTTGTTGACA
ATGCTTGGCACGTTCGTCAAGGACTGGTTTAGATATGAGTCACATTTTGTTCATGGTAGAGATTCTCTTGTTGACATTTTAAAAGAGCGTGGATTA
ATGAGTCACATTTTGTTCATGGTAGAGATTCTCTTGTTGACATTTTAA
ATGGTAGAGATTCTCTTGTTGACATTTTAA
ATGCTGTTCAACCACTAA
```

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:26:44

```
procedure GENE-ENUMERATION(s)
    n = |s|
    i = 1                                                    start codon
    while i + 2 ⩽ n do
        if s[i : i + 2] = ATG then
            j = i + 3                                        stop codon
            while j + 2 ⩽ n do
                if s[j : j + 2] ∈ {TAA, TAG, TGA} then
                    output i, j + 2, s[i : j + 2]            open reading frame
                    break                                    minimal substring
                j = j + 3
        i = i + 1
```

```
import sys

def gene_enumeration(s):
  n = len(s)
  i = 0
  while i + 2 <= n:
    if s[i : i + 3] == 'ATG':
      j = i + 3
      while j + 2 <= n:
        if s[j : j + 3] in {'TAA', 'TAG', 'TGA'}:
          print(i + 1, j + 2 + 1, s[i : j + 3])
          break
        j = j + 3
    i = i + 1

s = sys.stdin.readline().strip()
gene_enumeration(s)
```

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

void gene_enumeration(string s);

int main() {
  string s;
  cin >> s;
  gene_enumeration(s);
}
```

```
void gene_enumeration(string s) {
  set<string> stop = {"TAA", "TAG", "TGA"};
  int n = s.length();
  int i = 0;
  while (i + 2 <= n) {
    if (s.substr(i, 3) == "ATG") {
      int j = i + 3;
      while (j + 2 <= n) {
        if (stop.find(s.substr(j, 3)) != stop.end()) {
          cout << i + 1 << "␣" << j + 2 + 1 << "␣"
            << s.substr(i, j + 2 - i + 1) << endl;
          break;
        }
        j = j + 3;
    } }
    i = i + 1;
} }
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

The searching problem consists of, given a sequence A of n numbers and a value x, output an index i such that x equals $A[i]$ or the special value **nil** if x does not appear in A.

The linear search algorithm scans through the array from beginning to end, looking for x.

The procedure LINEAR-SEARCH takes an array $A[1 : n]$ and a value x.

```
function LINEAR-SEARCH(A, x)
    i = 1
    while i ⩽ n and x ≠ A[i] do
        i = i + 1
    if i > n then
        return nil
    return i
```

At the start of each iteration of the **while** loop, the value x does not appear in the subarray $A[1 : i - 1]$.

Observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against x and eliminate half of the subarray from further consideration.

The binary search algorithm which repeats this procedure, halving the size of the remaining portion of the subarray each time.

The procedure BINARY-SEARCH takes a sorted array $A[1 : n]$ and a value x.

function BINARY-SEARCH(A, x)
    return BINARY-SEARCH($A, x, 1, n$)

The procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function BINARY-SEARCH(A, x, low, high)
    while low ⩽ high do
        mid = ⌊(low + high)/2⌋
        if x = A[mid] then
            return mid
        else if x > A[mid] then
            low = mid + 1
        else
            high = mid − 1
    return nil
```

The procedure compares x to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration.

It returns either an index i such that $A[i] = x$, or **nil** if no entry of $A[low : high]$ contains the value x.

Bubble sort is a popular, but inefficient, sorting algorithm.

It works by repeatedly swapping adjacent elements that are out of order.

The procedure BUBBLE-SORT sorts array $A[1 : n]$.

```
procedure BUBBLE-SORT(A)
    for i = 1 to n − 1 do
        for j = n downto i + 1 do
            if A[j] < A[j − 1] then
                exchange A[j] with A[j − 1]
```

At the start of each iteration of the inner **for** loop, $A[j]$ is the smallest value in the subarray $A[j : n]$, and $A[j : n]$ is a permutation of the values that were in $A[j : n]$ at the time that the loop started.

At the start of each iteration of the outer **for** loop, the subarray $A[1 : i - 1]$ consists of the $i - 1$ smallest values originally in $A[1 : n]$, in sorted order, and $A[i : n]$ consists of the $n - i + 1$ remaining values originally in $A[1 : n]$.

Insertion sort is an efficient algorithm for sorting a small number of elements. It works the way you might sort a hand of playing cards. The procedure INSERTION-SORT sorts array $A[1 : n]$.

```
procedure INSERTION-SORT(A)
    for i = 2 to n do
        key = A[i]
                                insert A[i] into the sorted subarray A[1 : i − 1]
        j = i − 1
        while j > 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j − 1
        A[j + 1] = key
```

At the start of each iteration of the **for** loop, the subarray $A[1 : i - 1]$ consists of the elements originally in $A[1 : i - 1]$, but in sorted order.

Selection sort is another algorithm for sorting a small number of elements.

It works by first finding the smallest element of $A[1 : n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2 : n]$ and exchange it with $A[2]$. Then find the smallest element of $A[3 : n]$ and exchange it with $A[3]$. Continue in this manner for the first $n - 1$ elements of A.

The procedure SELECTION-SORT sorts array $A[1 : n]$.

```
procedure SELECTION-SORT(A)
    for i = 1 to n − 1 do
        smallest = i
        for j = i + 1 to n do
            if A[j] < A[smallest] then
                smallest = i
        exchange A[i] with A[smallest]
```

At the start of each iteration of the outer **for** loop, the subarray $A[1 : i - 1]$ consists of the $i - 1$ smallest elements in $A[1 : n]$, and this subarray is in sorted order.

Array	`list` `tuple`	`vector`
List	`list` `tuple`	`list`
Stack	`from collections` `import dequeue`	`stack`
Queue	`from collections` `import dequeue`	`queue`
Priority queue	`import heaqp`	`priority_queue`
Set	`set`	`set`
Dictionary	`dict`	`map`

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Sequence containers

Headers		<array>	<vector>	<deque>	<forward_list>	<list>
Members		array	vector	deque	forward_list	list
	constructor	*implicit*	vector	deque	forward_list	list
	destructor	*implicit*	~vector	~deque	~forward_list	~list
	operator=	*implicit*	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			
element access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		

modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap
list operations	splice				splice_after	splice
	remove				remove	remove
	remove_if				remove_if	remove_if
	unique				unique	unique
	merge				merge	merge
	sort				sort	sort
	reverse				reverse	reverse
observers	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator
	data	data	data			

Associative containers

Headers		<set>		<map>		<unordered_set>		<unordered_map>	
Members		**set**	**multiset**	**map**	**multimap**	**unordered_set**	**unordered_multiset**	**unordered_map**	**unordered_multimap**
	constructor	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap
	destructor	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap
	assignment	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin
	end	end	end	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin	rbegin				
	rend	rend	rend	rend	rend				
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin
	cend	cend	cend	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin	crbegin				
	crend	crend	crend	crend	crend				
capacity	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty	empty	empty	empty
	reserve					reserve	reserve	reserve	reserve
element access	at			at				at	
	operator[]			operator[]				operator[]	
modifiers	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	insert	insert	insert	insert	insert	insert	insert	insert	insert
	erase	erase	erase	erase	erase	erase	erase	erase	erase
	clear	clear	clear	clear	clear	clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap	swap	swap	swap
operations	count	count	count	count	count	count	count	count	count
	find	find	find	find	find	find	find	find	find
	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range
	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound				
	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound				
observers	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
	key_comp	key_comp	key_comp	key_comp	key_comp				
	value_comp	value_comp	value_comp	value_comp	value_comp				
	key_eq					key_eq	key_eq	key_eq	key_eq
	hash_function					hash_function	hash_function	hash_function	hash_function
buckets	bucket					bucket	bucket	bucket	bucket
	bucket_count					bucket_count	bucket_count	bucket_count	bucket_count
	bucket_size					bucket_size	bucket_size	bucket_size	bucket_size
	max_bucket_count					max_bucket_count	max_bucket_count	max_bucket_count	max_bucket_count
hash policy	rehash					rehash	rehash	rehash	rehash
	load_factor					load_factor	load_factor	load_factor	load_factor
	max_load_factor					max_load_factor	max_load_factor	max_load_factor	max_load_factor

procedure UNIQUE(S)
    let X be an empty set
    for all $s \in S$ **do**
        if $s \in X$ **then**
            return false
        else
            $X = X \cup \{s\}$
    return true

```
def unique(S):
  X = set()
  for s in S:
    if s in X:
      return False
    else:
      X.add(s)
  return True
```

```
bool unique(const
    vector<string>& S) {
  set<string> X;
  for (auto s : S) {
    if (X.find(s) !=
        X.end())
      return false;
    else
      X.insert(s);
  }
  return true;
}
```

```
import sys

def unique(S):
  X = set()
  for s in S:
    if s in X:
      return False
    else:
      X.add(s)
  return True

L = list()
for line in sys.stdin.readlines():
  s = line.strip()
  L.append(s)
print(unique(L))
```

```
#include <set>
#include <vector>
#include <iostream>
using namespace std;

bool unique(const vector<string>& S) {
  set<string> X;
  for (auto s : S) {
    if (X.find(s) != X.end()) return false;
    else X.insert(s);
  }
  return true;
}

int main() {
  vector<string> L;
  string s;
  while (cin >> s) L.push_back(s);
  bool found = unique(L);
  cout << (found ? "True" : "False") << endl;
}
```

procedure FREQUENCY-DISTRIBUTION(S)
    let D be an empty dictionary
    for all $s \in S$ **do**
        if $s \in D$ **then**
            $D[s] = D[s] + 1$
        else
            $D[s] = 1$
    return D

```
def freq_distribution(S):
  D = dict()
  for s in S:
    D[s] = D.get(s, 0) + 1
  return D
```

```
map<string, int>
freq_distribution(const
    vector<string>& S) {
  map<string, int> D;
  for (auto s : S) {
    if (D.find(s) ==
        D.end())
      D[s] = 1;
    else
      D[s] += 1;
  }
  return D;
}
```

```
import sys

def frequency_distribution(S):
  D = dict()
  for s in S:
    D[s] = D.get(s, 0) + 1
  return D

L = list()
for line in sys.stdin.readlines():
  s = line.strip()
  L.append(s)
D = frequency_distribution(L)
for s in sorted(D):
  print(s, D[s])
```

```
#include <map>
#include <vector>
#include <iostream>
using namespace std;

map<string, int>
frequency_distribution(const vector<string>& S) {
  map<string, int> D;
  for (auto s : S) {
    if (D.find(s) == D.end()) D[s] = 1;
    else D[s] += 1;
  }
  return D;
}

int main() {
  vector<string> L;
  string s;
  while (cin >> s) L.push_back(s);
  map<string, int> D = frequency_distribution(L);
  for (auto it = D.begin(); it != D.end(); ++it) {
    cout << it->first << "␣" << it->second << endl;
} }
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

RNA to protein — X95757_en

Recall that the primary structure of a protein can be represented as a sequence over the alphabet of amino acids A (alanine, Ala), R (arginine, Arg), N (asparagine, Asn), D (aspartate, Asp), C (cysteine, Cys), E (glutamate, Glu), Q (glutamine, Gln), G (glycine, Gly), H (histidine, His), I (isoleucine, Ile), L (leucine, Leu), K (lysine, Lys), M (methionine, Met), F (phenylalanine, Phe), P (proline, Pro), S (serine, Ser), T (threonine, Thr), W (tryptophan, Trp), Y (tyrosine, Tyr), and V (valine, Val).

A codon of three nucleotides is translated into a single amino acid within a protein, with translation beginning with a start codon (AUG) and ending with a stop codon (UAA, UAG, or UGA). The $4^3 = 64$ different nucleotide triplets code for 20 amino acids, one translation start signal (methionine, one of these amino acids) and three translation stop signals, with some redundancies. The genetic code defines a mapping between codons and amino acids, and despite variations in the genetic code across species, there is a standard genetic code common to most species.

AAA	K	AAC	N	AAG	K	AAU	N	ACA	T	ACC	T	ACG	T	ACU	T
AGA	R	AGC	S	AGG	R	AGU	S	AUA	I	AUC	I	AUG	M	AUU	I
CAA	Q	CAC	H	CAG	Q	CAU	H	CCA	P	CCC	P	CCG	P	CCU	P
CGA	R	CGC	R	CGG	R	CGU	R	CUA	L	CUC	L	CUG	L	CUU	L
GAA	E	GAC	D	GAG	E	GAU	D	GCA	A	GCC	A	GCG	A	GCU	A
GGA	G	GGC	G	GGG	G	GGU	G	GUA	V	GUC	V	GUG	V	GUU	V
UAA	-	UAC	Y	UAG	-	UAU	Y	UCA	S	UCC	S	UCG	S	UCU	S
UGA	-	UGC	C	UGG	W	UGU	C	UUA	L	UUC	F	UUG	L	UUU	F

Write code for the protein translation problem. The program must implement and use the RNA-TO-PROTEIN function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\{A, C, G, U\}$.

Output

The output is the translation of a minimal substring of s from a start codon to a stop codon to a string (proteomic sequence) over the alphabet $\{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$.

Sample input

```
GUCGCCAUGAUGGUGGUUAUUAUACCGUCAAGGACUGUGUGACUA
```

Sample output

```
MVVIIPSRTV
```

```
function RNA-TO-PROTEIN(s)
    let D be an empty dictionary (of strings to characters)
    D[AAA] = K
    ...
    D[UUU] = F
    let t be an empty string
    n = |s|
    i = 1
    while i + 2 ⩽ n and s[i : i + 2] ≠ AUG do
        i = i + 1
    if i + 2 ⩽ n then
        i = i + 3                                   skip the start codon
        while i + 2 ⩽ n and s[i : i + 2] ∉ {UAA, UAG, UGA} do
            append D[s[i : i + 2]] to t
            i = i + 3
        if i + 2 ⩽ n then
            return t
        else
            return an empty string          start codon but no stop codon
    else
        return an empty string                          no start codon
```

```
import sys

D = dict()
D['AAA'] = 'K'
...
D['UUU'] = 'F'

def rna_to_protein(s):
  ...

s = sys.stdin.readline().strip()
t = rna_to_protein(s)
if t != '':
  print(t)
```

```
def rna_to_protein(s):
  t = ''
  n = len(s)
  i = 1
  while i + 2 <= n and s[i : i + 3] != 'AUG':
    i = i + 1
  if i + 2 <= n:
    i = i + 3
    while i + 2 <= n and
        not s[i : i + 3] in {'UAA', 'UAG', 'UGA'}:
      t += D[s[i : i + 3]]
      i = i + 3
    if i + 2 <= n:
      return t
    else:
      return ''
  else:
    return ''
```

```
#include <map>
#include <set>
#include <string>
#include <iostream>
using namespace std;

string rna_to_protein(string s);

int main() {
  string s;
  cin >> s;
  string t = rna_to_protein(s);
  if (t != "") cout << t << endl;
}
```

```
string rna_to_protein(string s) {
  map<string, char> D;
  D["AAA"] = 'K'; ... D["UUU"] = 'F';
  string t;
  int n = s.length();
  int i = 1;
  while (i + 2 <= n and s.substr(i, 3) != "AUG")
    i = i + 1;
  if (i + 2 <= n) {
    i = i + 3;
    set<string> stop = {"UAA", "UAG", "UGA"};
    while (i + 2 <= n and
        stop.find(s.substr(i, 3)) == stop.end()) {
      if (D.find(s.substr(i, 3)) != D.end())
        t.push_back(D[s.substr(i, 3)]);
      else t.push_back('*');
      i = i + 3;
    }
    if (i + 2 <= n) return t;
    else return "";
  }
  else return "";
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Read abundance X28783_en

The abundance (number of occurrences) of a read in a read set is an indicator value for read confidence in high-throughput sequencing studies.

Write pseudocode, Python code, and C++ code for the read abundance problem. Make two submissions, including the pseudocode as a comment to both the Python and the C++ code.

Input

The input is a collection of n strings $S = \{s_1, s_2, \ldots, s_n\}$ (genomic sequence reads, possibly reverse complemented) over the alphabet $\{A, C, G, T\}$.

Output

The output is the sorted frequency distribution of S.

Sample input

```
TCATC
TTGAT
TCATC
TGAAA
GATGA
TTTCA
ATCAA
TTGAT
TTTCA
```

Sample output

```
ATCAA 3
GATGA 3
TGAAA 3
```

Problem information

Author : Gabriel Valiente
Generation : 2021-10-05 14:50:03

```
function REVERSE-COMPLEMENT(s)
    let C be an empty dictionary (of characters to characters)
    C[A] = T
    C[C] = G
    C[G] = C
    C[T] = A
    let t be a copy of s
    for i = 1 to |s| do
        t[i] = C[s[n − (i − 1)]]
    return t
```

```
procedure READ-ABUNDANCE(S)
    let D be an empty dictionary (of strings to integers)
    for all s ∈ S do
        r = REVERSE-COMPLEMENT(s)
        if r < s then
            s = r
        if s ∈ D then
            D[s] = D[s] + 1
        else
            D[s] = 1

    sort D
    return D
```

```
import sys

def reverse_complement(s):
  C = {'A' : 'T', 'C' : 'G', 'G' : 'C', 'T' : 'A'}
  s = ''.join([C[x] for x in s[::-1]])
  return s

def read_abundance(S):
  D = dict()
  for s in S:
    r = reverse_complement(s)
    if r < s:
      s = r
    D[s] = D.get(s, 0) + 1
  return D

S = list()
for line in sys.stdin.readlines():
  s = line.strip()
  S.append(s)
D = read_abundance(S)
for s in sorted(D):
  print(s, D[s])
```

```
#include <map>
#include <vector>
#include <iostream>
using namespace std;

string reverse_complement(string s);

map<string, int>
read_abundance(const vector<string>& S);

int main() {
  vector<string> S;
  string s;
  while (cin >> s) {
    string t = reverse_complement(s);
    if (t < s) s = t;
    S.push_back(s);
  }
  map<string, int> D = read_abundance(S);
  for (auto item : D)
    cout << item.first << "␣" << item.second << endl;
}
```

```
string reverse_complement(string s) {
  string t(s);
  map<char, char>
  C = {{'A', 'T'}, {'C', 'G'}, {'G', 'C'}, {'T', 'A'}};
  for (int i = 0; i < int(s.length()); ++i)
    t[i] = C[s[s.length() - (i + 1)]];
  return t;
}

map<string, int>
read_abundance(const vector<string>& S) {
  map<string, int> D;
  for (auto s : S) {
    if (D.find(s) != D.end()) D[s] += 1;
    else D[s] = 1;
  }
  return D;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

You might consider resources such as memory, communication bandwidth, or energy consumption.

Most often, however, you'll want to measure computational time.

If you analyze several candidate algorithms for a problem, you can identify the most efficient one.

There might be more than just one viable candidate, but you can often rule out several inferior algorithms in the process.

Before you can analyze an algorithm, you need a model of the technology it runs on, including the resources of that technology and a way to express their costs.

We assume a generic one-processor, random-access machine (RAM) model of computation as the implementation technology, with the understanding that algorithms are implemented as computer programs.

In the RAM model, instructions execute one after another, with no concurrent operations.

The RAM model assumes that each instruction takes the same amount of time as any other instruction and that each data access (using the value of a variable or storing into a variable) takes the same amount of time as any other data access.

In other words, in the RAM model each instruction or data access takes a constant amount of time (even indexing into an array, assuming that each element of a given array occupies the same number of bytes and that the elements of a given array are stored in contiguous memory locations).

The RAM model contains instructions commonly found in real computers. arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in the RAM model are integer, floating point (for storing real-number approximations), and character.

We assume that each word of data has a limit on the number of bits (similar to the number of bits in a computer word).

For example, when working with inputs of size n, we typically assume that integers are respresented by $c \log_2 n$ bits for some constant $c \geqslant 1$.

We require $c \geqslant 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily.

The RAM model does not account for the memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. However, RAM-model analyses are usually excellent predictors of performance on actual machines.

Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as the ability to identify the most significant terms in a formula.

Because an algorithm might behave differently for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Instead of timing a run, or even several runs, of an implementation of a given algorithm in a real programming language, we can determine how long it takes by analyzing the algorithm itself.

We'll examine how many times it executes each line of pseudocode and how long each line of pseudocode takes to run.

We'll first come up with a precise but complicated formula for the running time.

Then, we'll distill the important part of the formula using a convenient notation that can help us compare the running times of different algorithms for the same problem.

How do we analyze a given algorithm?

First, let's acknowledge that the running time depends on the input.

Even though the running time can depend on many features of the input, we'll focus on the one that has been shown to have the greatest effect, namely the size of the input, and describe the running time of a program as a function of the size of its input.

The best notion for input size depends of the problem being studied.

For many problems, the most natural measure is the number of items in the input.

For many other problems, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.

The running time of an algorithm on a particular input is the number of instructions and data accesses executed.

How we account for these costs should be independent of any particular computer, but within the framework of the RAM model.

A constant amount of time is required to execute each line of our pseudocode.

One line might take more or less time than another line, but we'll assume that each execution of the kth line takes c_k time, where c_k is a constant.

For each $i = 2, 3, \ldots, n$, let t_i denote the number of times the while loop test is executed for that value of i.

Recall that the procedure INSERTION-SORT sorts array $A[1 : n]$.

```
procedure INSERTION-SORT(A)
    for i = 2 to n do                                          cost c1 times n
        key = A[i]                                             cost c2 times n − 1
        insert A[i] into the sorted subarray A[1 : i − 1]      cost 0 times n−1
        j = i − 1                                              cost c4 times n − 1
        while j > 0 and A[j] > key do                          cost c5 times ∑_{i=2}^{n} t_i
            A[j + 1] = A[j]                                    cost c6 times ∑_{i=2}^{n} (t_i − 1)
            j = j − 1                                          cost c7 times ∑_{i=2}^{n} (t_i − 1)
        A[j + 1] = key                                         cost c8 times n − 1
```

The running time of the algorithm is the sum of the running times for each statement executed.

A statement that takes c_k steps to execute and executes m times contributes $c_k m$ to the total running time.

We usually denote the running time of an algorithm on an input of size n by $T(n)$.

To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of cost and times, obtaining

$$\begin{aligned} T(n) = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\ & + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \end{aligned}$$

Even for inputs of a given size, the algorithm's running time may depend on which input of that size is given.

For INSERTION-SORT, the best case occurs when the array is already sorted, and the worst case arises when the array is in reverse sorted order.

In the best case, each time the **while** loop executes, the value originally in $A[i]$ is already greater than or equal to all values in $A[1 : i-1]$, so that the **while** loop always exits upon the first test.

Therefore, we have that $t_i = 1$ for $i = 2, 3, \ldots, n$, and the best-case running time is given by

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

We can express this best-case running time as $an + b$ for constants a and b that depend on the statement costs c_k.

The running time is thus a linear function of n.

In the worst case, the procedure must compare each element $A[i]$ with each element in the entire sorted subarray $A[1 : i-1]$, and so $t_i = i$ for $i = 2, 3, \ldots, n$. Noting that

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

we find that the worst-case running time is given by

$$\begin{aligned} T(n) =& c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &+ c_6\left(\frac{n(n-1)}{2} - 1\right) + c_7\left(\frac{n(n-1)}{2} - 1\right) + c_8(n-1) \\ =& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &- (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b and c that depend on the statement costs c_k.

The running time is thus a quadratic function of n.

Our analysis of insertion sort looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted.

We'll usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n, for three reasons.

First, the worst-case running time of an algorithm gives an upper bound on the running time for any input.

Second, for some algorithms, the worst case occurs fairly often.

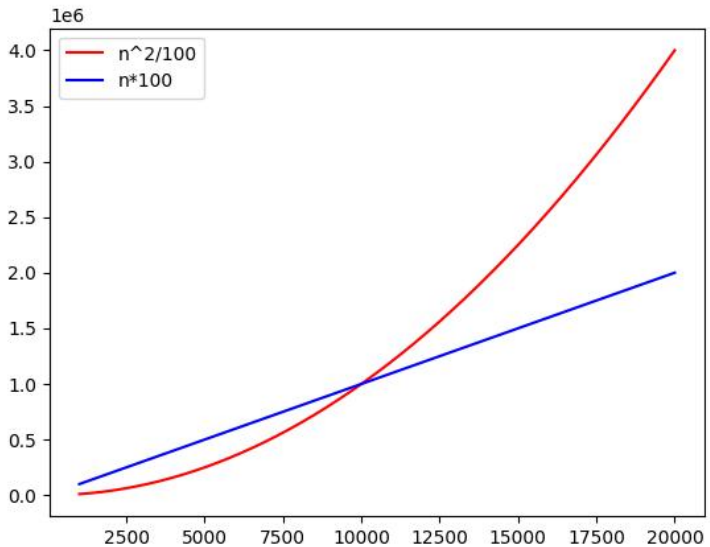
Third, the "average case" is often roughly as bad as the worst case.

Now, we expressed the best-case running time as $an + b$, and the worst-case running time as $an^2 + bn + c$, for constants a, b and c that depend on the statement costs.

Another simplifying assumption is the rate of growth, or order of growth, of the running time, where we consider only the leading term of a formula, since the lower-order terms are relatively insignificant for large values of n, and we also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

For insertion sort's worst-case running time, when we ignore the lower-order terms and the leading term's constant coefficient, only the factor of n^2 from the leading term remains, and that factor is by far the most important part of the running time.

For example, suppose that an algorithm implemented on a particular maching takes $n^2/100 + 100n + 17$ microseconds on an input of size n. Although the coefficients of $1/100$ for the n^2 term and 100 for the n term differ by four orders of magnitude, the $n^2/100$ term dominates the $100n$ term once n exceeds 10,000.



To highlight the order of growth of the running time, we have a special notation that uses the Greek letter Θ (theta).

We say that insertion sort has a worst-case running time of $\Theta(n^2)$, which means "roughly proportional to n^2 when n is large."

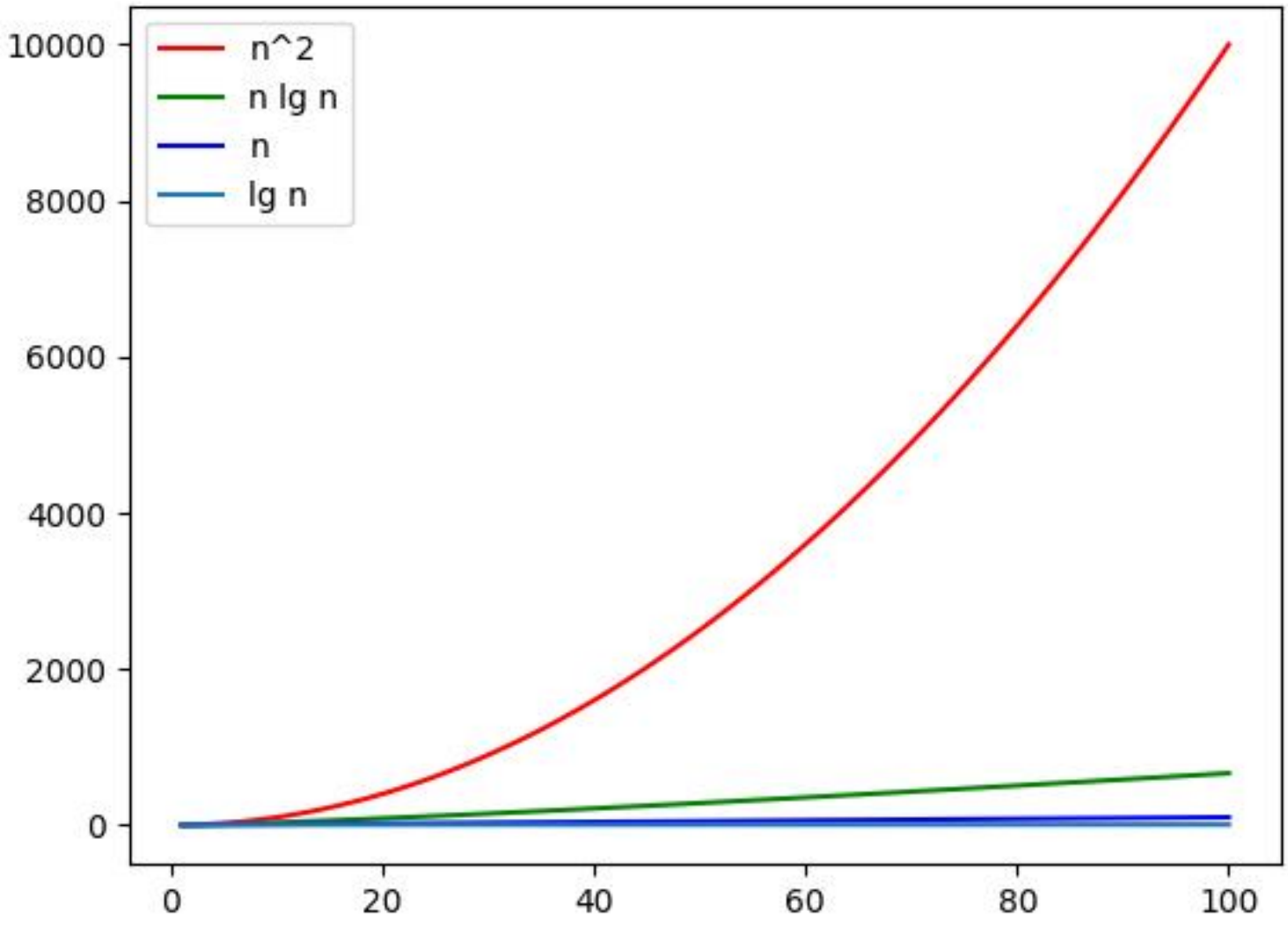
We also say that insertion sort has a best-case running time of $\Theta(n)$, which means "roughly proportional to n when n is large."

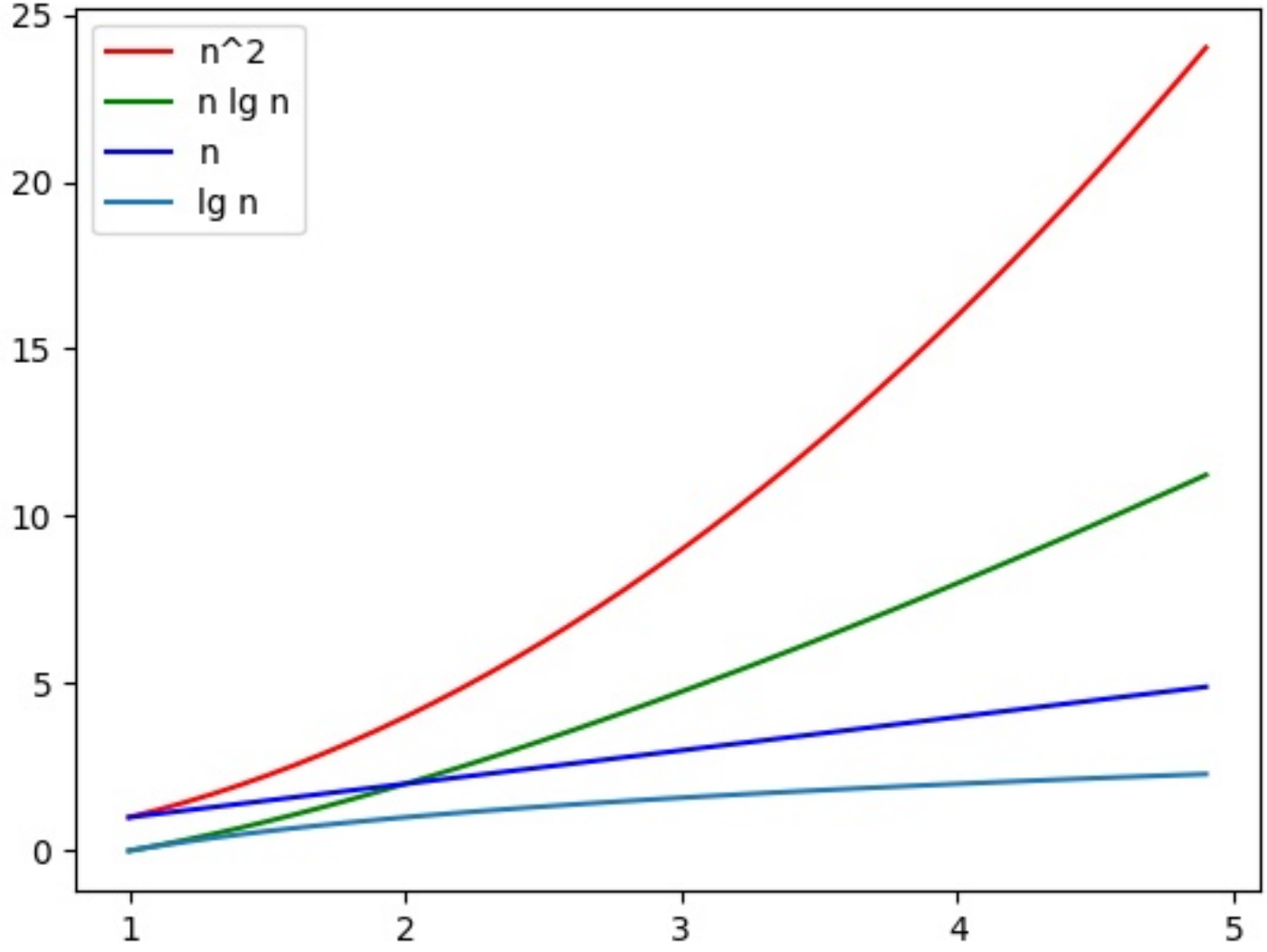
We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

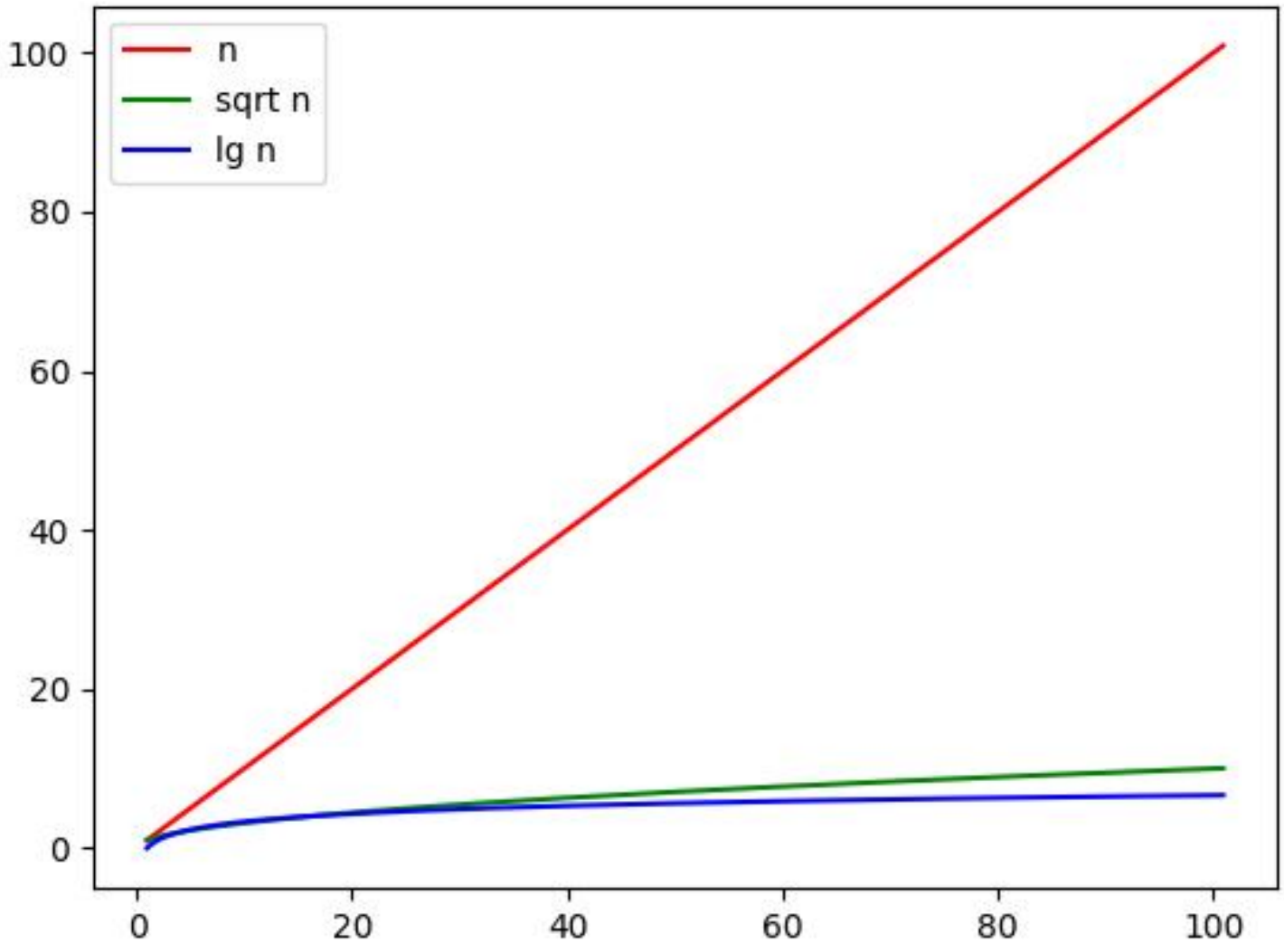
Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth.

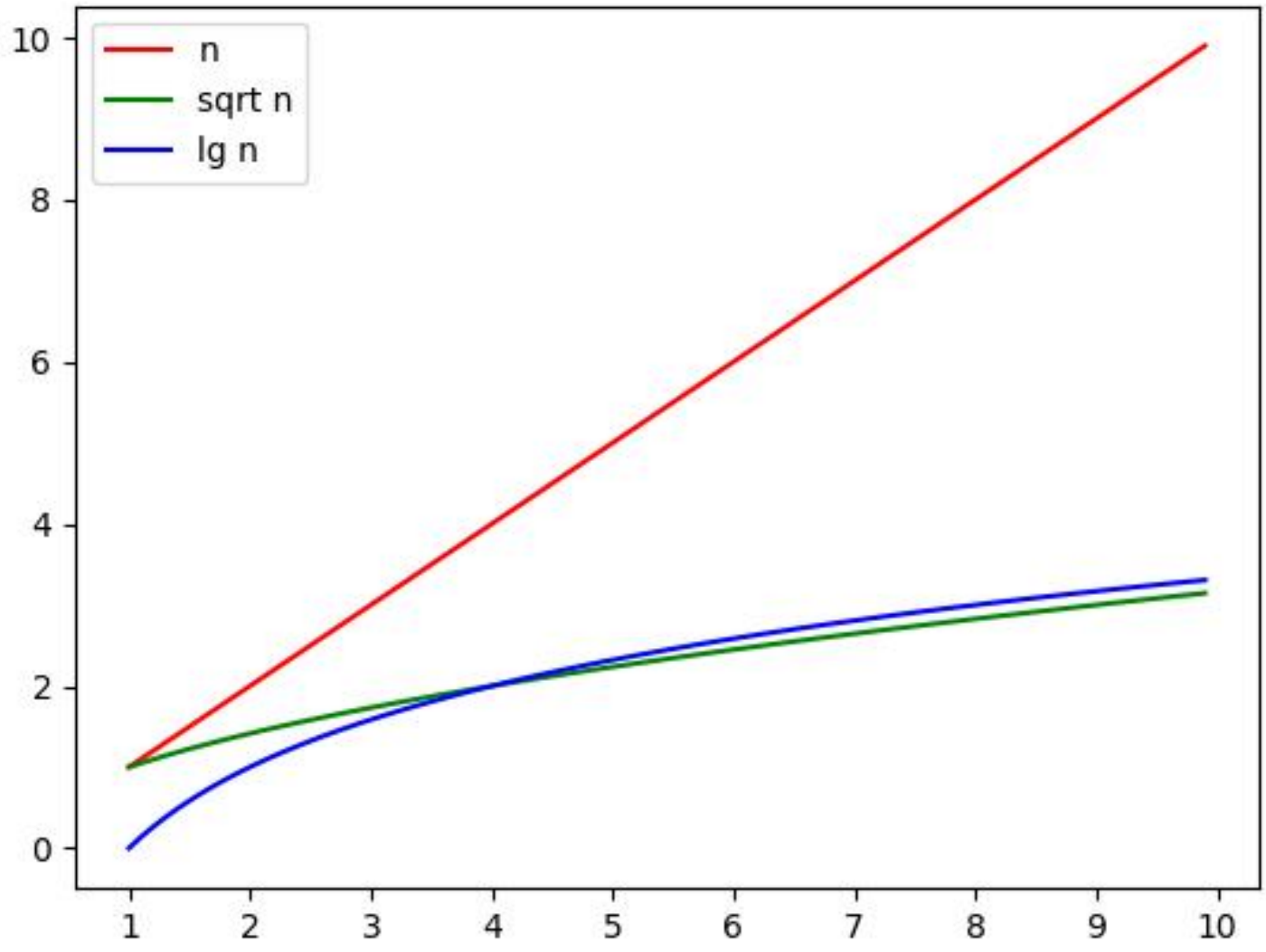
But on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, for example, takes less time in the worst case than an algorithm whose worst-case running time is $\Theta(n^3)$.

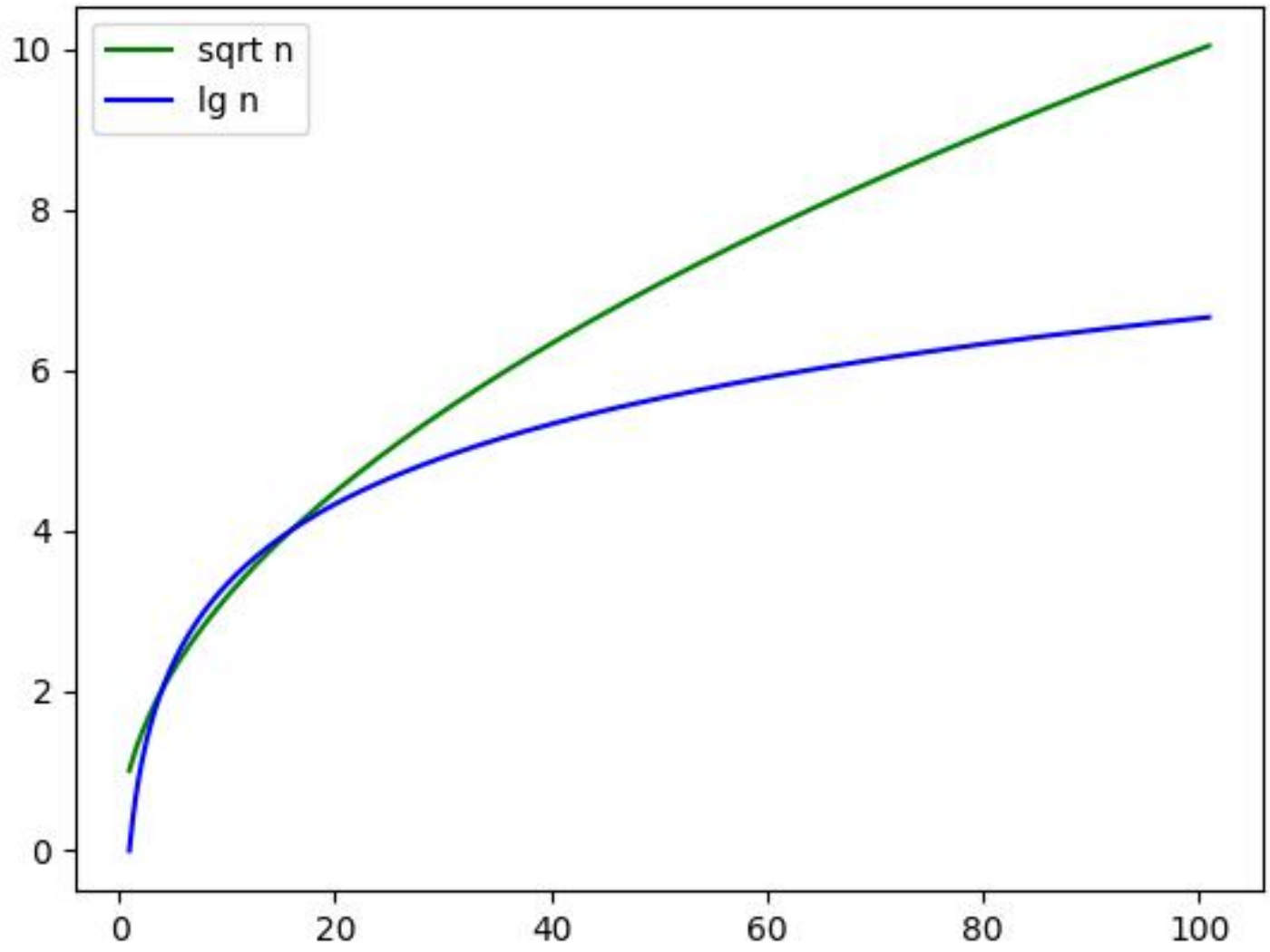
Regardless of the constants hidden by the Θ-notation, there is always some number, say n_0, such that for all input sizes $n \geqslant n_0$, the $\Theta(n^2)$ algorithm beats the $\Theta(n^3)$ algorithm in the worst case.


n^2
n lg n
n
lg n
10000
8000
6000
4000
2000
0
0
20
40
60
80
100


n^2
n lg n
n
lg n
25
20
15
10
5
0
1
2
3
4
5


n
sqrt n
lg n
100
80
60
40
20
0
0
20
40
60
80
100


n
sqrt n
lg n
10
8
6
4
2
0
1
2
3
4
5
6
7
8
9
10


sqrt n
lg n
10
8
6
4
2
0
0
20
40
60
80
100

The procedure LINEAR-SEARCH takes an array $A[1 : n]$ and a value x.

```
function LINEAR-SEARCH(A, x)
    i = 1
    while i ⩽ n and x ≠ A[i] do
        i = i + 1
    if i > n then
        return nil
    return i
```

In the best case, x appears in the first position of the array. Therefore, the running time of the algorithm is $\Theta(1)$ in the best case.

In the worst case, x appears only in the last position of the array, so that the entire array must be checked. Therefore, the running time of the algorithm is $\Theta(n)$ in the worst case.

The procedure BINARY-SEARCH takes a sorted array $A[1 : n]$ and a value x.

```
function BINARY-SEARCH(A, x)
    return BINARY-SEARCH(A, x, 1, n)
```

The procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function BINARY-SEARCH(A, x, low, high)
    while low ⩽ high do
        mid = ⌊(low + high)/2⌋
        if x = A[mid] then
            return mid
        else if x > A[mid] then
            low = mid + 1
        else
            high = mid − 1
    return nil
```

The running time of the algorithm is $\Theta(\lg n)$ in the worst case.

The procedure terminates the search unsuccessfully when the range is empty and terminates it successfully if the value x has been found.

Based on the comparison of x to the middle element in the searched range, the search continues with the range halved.

The recurrence for this procedure is therefore $T(n) = T(n/2) + \Theta(1)$, whose solution is $\Theta(\lg n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ T(n/2) + 1 & \text{if } n > 2 \end{cases}$$

is $T(n) = \lg n$.

The base case is when $n = 2$, and we have $\lg n = \lg 2 = 1$.

For the inductive step, our hypothesis is that $T(n/2) = \lg(n/2)$. Then

$$T(n) = T(n/2) + 1 = \lg(n/2) + 1 = \lg n - \lg 2 + 1 = \lg n - 1 + 1 = \lg n$$

which completes the inductive proof for exact powers of 2.

The procedure BUBBLE-SORT sorts array $A[1 : n]$.

```
procedure BUBBLE-SORT(A)
    for i = 1 to n − 1 do
        for j = n downto i + 1 do
            if A[j] < A[j − 1] then
                exchange A[j] with A[j − 1]
```

The running time depends on the number of iterations of the inner **for** loop.

For a given value of i, this loop makes $n - i$ iterations, and i takes on the values $1, 2, \ldots, n - 1$.

The total number of iterations, therefore, is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, the running time of the algorithm is $\Theta(n^2)$ for all cases.

The procedure SELECTION-SORT sorts array $A[1:n]$.

```
procedure SELECTION-SORT(A)
    for i = 1 to n − 1 do
        smallest = i
        for j = i + 1 to n do
            if A[j] < A[smallest] then
                smallest = i
        exchange A[i] with A[smallest]
```

The running time depends on the number of iterations of the inner **for** loop.

For a given value of i, this loop makes $n - i$ iterations, and i takes on the values $1, 2, \ldots, n - 1$.

The total number of iterations, therefore, is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, the running time of the algorithm is $\Theta(n^2)$ for all cases.

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Read mapping 1 X89781_en

Sequence reads are mapped to a reference genome using some indexing data structure, one of which is the suffix array. The suffix array of a string is a sorted array of the suffixes of the string, that is, an array of the (starting) positions of the sorted suffixes of the string.

For example, the genomic sequence fragment TATAAT has the suffixes TATAAT at position 1, ATAAT at position 2, TAAT at position 3, AAT at position 4, AT at position 5, and T at position 6. The sorted suffixes are AAT at position 4, AT at position 5, ATAAT at position 2, T at position 6, TAAT at position 3, and TATAAT at position 1. Therefore, the suffix array of TATAAT is $(4, 5, 2, 6, 3, 1)$.

Write code for the read mapping problem. The program must implement and use the SUFFIX-ARRAY function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is the suffix array of s.

Sample input	Sample output

`TATAAT`	`4`
	`5`
	`2`
	`6`
	`3`
	`1`

Hint

Use the built-in sorting algorithm of Python or C++ to sort suffixes.

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:28:10

```
procedure SUFFIX-ARRAY(s)
    n = |s|
    let A[1 : n] be a new array (of strings)
    for i = 1 to n do
        A[i] = s[i : n]
    sort A in lexicographic order
    let A′[1 : n] be a new array (of integers)
    for i = 1 to n do
        A′[i] = |s| − |A[i]| + 1
    return A′
```

```
import sys

def suffix_array(s):
  A = [s[i:] for i in range(len(s))]
  A.sort()
  return [len(s) - len(x) + 1 for x in A]

s = sys.stdin.readline().strip()
SA = suffix_array(s)
for x in SA:
  print(x)
```

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

vector<int> suffix_array(string s) {
  vector<string> A;
  for (int i = 0; i < int(s.length()); ++i)
    A.push_back(s.substr(i, string::npos));
  sort(A.begin(), A.end());
  vector<int> SA;
  for (auto x : A)
    SA.push_back(s.length() - x.length() + 1);
  return SA;
}

int main() {
  string s;
  cin >> s;
  vector<int> SA = suffix_array(s);
  for (auto x : SA) cout << x << endl;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Read mapping 2 X38913_en

Sequence reads are mapped to a reference genome using some indexing data structure, one of which is the suffix array. The suffix array of a string is a sorted array of the suffixes of the string, that is, an array of the (starting) positions of the sorted suffixes of the string.

For example, the genomic sequence fragment TATAAT has the suffixes TATAAT at position 1, ATAAT at position 2, TAAT at position 3, AAT at position 4, AT at position 5, and T at position 6. The sorted suffixes are AAT at position 4, AT at position 5, ATAAT at position 2, T at position 6, TAAT at position 3, and TATAAT at position 1. Therefore, the suffix array of TATAAT is $(4, 5, 2, 6, 3, 1)$.

Write code for the read mapping problem. The program must implement and use the SUFFIX-ARRAY function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is the suffix array of s.

Sample input	Sample output

`TATAAT`	`4`
	`5`
	`2`
	`6`
	`3`
	`1`

Hint

Implement and use the bubble-sort algorithm to sort suffixes.

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:28:26

Recall that the procedure BUBBLE-SORT sorts array $A[1 : n]$.

```
procedure BUBBLE-SORT(A)
    for i = 1 to n − 1 do
        for j = n downto i + 1 do
            if A[j] < A[j − 1] then
                exchange A[j] with A[j − 1]
```

```
def bubble_sort(L):
  n = len(L)
  for i in range(n - 1):
    for j in range(n - 1, i, -1):
      if L[j] < L[j - 1]:
        L[j], L[j - 1] = L[j - 1], L[j]
```

```
void bubble_sort(vector<pair<string, int>>& T) {
  int n = T.size();
  for (int i = 0; i < n - 1; ++i)
    for (int j = n - 1; j > i; --j)
      if (T[j] < T[j - 1]) swap(T[j], T[j - 1]);
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

Many useful algorithms are recursive in structure: to solve a given problem, they recurse (call themselves) one or more times to handle closely related subproblems.

The procedure LINEAR-SEARCH takes an array $A[1 : n]$ and a value x.

```
function LINEAR-SEARCH(A, x)
    return LINEAR-SEARCH(A, x, 1, n)
```

The procedure LINEAR-SEARCH takes an array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function LINEAR-SEARCH(A, x, low, high)
    if low > high then
        return nil
    else if x = A[low] then
        return low
    else
        return LINEAR-SEARCH(A, x, low + 1, high)
```

The procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function BINARY-SEARCH(A, x)
    return BINARY-SEARCH(A, x, 1, n)

function BINARY-SEARCH(A, x, low, high)
    if low > high then
        return nil
    m = ⌊(low + high)/2⌋
    if x = A[m] then
        return m
    else if x > A[m] then
        return BINARY-SEARCH(A, x, m + 1, high)
    else
        return BINARY-SEARCH(A, x, low, m − 1)
```

Give the pseudocode of a recursive algorithm to compute the factorial $n!$ of a natural number n.

```
function FACTORIAL(n)
    if n ⩽ 1 then
        return 1
    else
        return n · FACTORIAL(n − 1)
```

Give the pseudocode of a recursive algorithm to compute the greatest common divisor of two strictly positive natural numbers a and b using the slow version of the Euclidean algorithm.

```
function GCD(a, b)
    if a > b then
        return GCD(a − b, b)
    else if a < b then
        return GCD(a, b − a)
    else
        return a
```

Give the pseudocode of a recursive algorithm to compute the greatest common divisor of two natural numbers a and b using the fast version of the Euclidean algorithm.

```
function GCD(a, b)
    if b = 0 then
        return a
    else
        return GCD(b, a mod b)
```

A well-known mathematical property states that a natural number is a multiple of 3 if and only if the sum of its digits is also a multiple of 3.

For instance, the sum of the digits of 8472 is $8 + 4 + 7 + 2 = 21$, which is a multiple of 3. Therefore, 8472 is also a multiple of 3.

Give the pseudocode of a recursive algorithm to compute the sum of the digits of a natural number n.

```
function SUM-OF-DIGITS(n)
    if n ⩽ 9 then
        return n
    else
        return n mod 10 + SUM-OF-DIGITS(n/10)
```

Give the pseudocode of a recursive algorithm to determine if a strictly positive natural number n is a multiple of 3.

```
function IS-MULTIPLE-OF-3(n)
    if n ⩽ 9 then
        return n = 3 or n = 6 or n = 9
    else
        return IS-MULTIPLE-OF-3(SUM-OF-DIGITS(n))
```

Give the pseudocode of a recursive algorithm to read a sequence of words and print it in the same order.

```
procedure KEEP()
    read a string s from the input stream
    if s is not empty then
        output s
        KEEP()
```

Give the pseudocode of a recursive algorithm to read a sequence of words and print it in reverse order.

```
procedure REVERSE()
    read a string s from the input stream
    if s is not empty then
        REVERSE()
        output s
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Sliding window 1 X38951_en

Recall that a string (genomic sequence) can be split in words of length 3 (codons) by sliding a window of size 3 over the string, with a step size of 3. More in general, a string can be split in overlapping words of length x and overlap size $x - y$ by sliding a window of size x and step size y over the string. For example, sliding a window of size 3 and step size 2 over the string TATAAT gives the overlapping words TAT and TAA.

Write code for the sliding window problem. The program must implement and use the SLIDING-WINDOW function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$, an integer x (the window size), and an integer y (the step size).

Output

The output is all substrings of s of size x starting at positions $1, 1 + y, 1 + 2y, \ldots$

Sample input 1

```
ACGGTAGACCT
3
1
```

Sample output 1

```
ACG
CGG
GGT
GTA
TAG
AGA
GAC
ACC
CCT
```

Sample input 2

```
ACGGTAGACCT
3
3
```

Sample output 2

```
ACG
GTA
GAC
```

Sample input 3

```
ACGGTAGACCT
3
5
```

Sample output 3

```
ACG
AGA
```

Sample input 4

```
ACGGTAGACCT
5
2
```

Sample output 4

```
ACGGT
GGTAG
TAGAC
GACCT
```

```
function SLIDING-WINDOW(s, x, y)
    let L be an empty list
    for i = 1 to |s| − x − 1 step y do
        append s[i : i + x − 1] to L
    return L
```

```
function SLIDING-WINDOW(s, x, y)
    let L be an empty list
    i = 1
    while i < |s| − x do
        append s[i : i + x − 1] to L
        i = i + y
    return L
```

```
import sys

def sliding_window(s, x, y):
  L = list()
  for i in range(0, len(s) - x + 1, y):
    L += [s[i : i + x]]
  return L

s = sys.stdin.readline().strip()
x = int(sys.stdin.readline().strip())
y = int(sys.stdin.readline().strip())
for z in sliding_window(s, x, y):
  print(z)
```

```
#include <iostream>
#include <vector>
using namespace std;

vector<string>
sliding_window(string s, int x, int y) {
  vector<string> L;
  for (int i = 0; i < s.length() - x + 1; i += y)
    L.push_back(s.substr(i, x));
  return L;
}

int main() {
  string s;
  int x, y;
  cin >> s >> x >> y;
  vector<string> L = sliding_window(s, x, y);
  for (string z : L) cout << z << endl;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Sliding window 2 X99827_en

Recall that a string (genomic sequence) can be split in words of length 3 (codons) by sliding a window of size 3 over the string, with a step size of 3. More in general, a string can be split in overlapping words of length x and overlap size $x - y$ by sliding a window of size x and step size y over the string. For example, sliding a window of size 3 and step size 2 over the string TATAAT gives the overlapping words TAT and TAA.

Write code for the sliding window problem. The program must implement and use the SLIDING-WINDOW function in the pseudocode discussed in class, which is recursive and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s (a genomic sequence) over the alphabet $\Sigma = \{A, C, G, T\}$, an integer x (the window size), and an integer y (the step size).

Output

The output is all substrings of s of size x starting at positions $1, 1 + y, 1 + 2y, \ldots$

Sample input 1

```
ACGGTAGACCT
3
1
```

Sample output 1

```
ACG
CGG
GGT
GTA
TAG
AGA
GAC
ACC
CCT
```

Sample input 2

```
ACGGTAGACCT
3
3
```

Sample output 2

```
ACG
GTA
GAC
```

Sample input 3

```
ACGGTAGACCT
3
5
```

Sample output 3

```
ACG
AGA
```

Sample input 4

```
ACGGTAGACCT
5
2
```

Sample output 4

```
ACGGT
GGTAG
TAGAC
GACCT
```

```
function SLIDING-WINDOW(s, x, y)
    if |s| < x then
        return an empty list
    else
        let L be an empty list
        append s[1, ..., x] to L
        concatenate SLIDING-WINDOW(s[y + 1 : |s|], x, y) to L
        return L
```

```
import sys

def sliding_window(s, x, y):
  if len(s) < x:
    return []
  else:
    return [s[0:x]] + sliding_window(s[y:], x, y)

s = sys.stdin.readline().strip()
x = int(sys.stdin.readline().strip())
y = int(sys.stdin.readline().strip())
for z in sliding_window(s, x, y):
  print(z)
```

```
#include <iostream>
#include <vector>
using namespace std;

vector<string>
sliding_window(string s, int x, int y) {
  vector<string> L;
  if (s.length() < x) return L;
  else {
    L.push_back(s.substr(0, x));
    vector<string> LL = sliding_window(
      s.substr(y, s.length() - y), x, y);
    L.insert(L.end(), LL.begin(), LL.end());
    return L;
} }

int main() {
  string s;
  int x, y;
  cin >> s >> x >> y;
  vector<string> L = sliding_window(s, x, y);
  for (string z : L) cout << z << endl;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Words 1 X11585_en

Nucleic acid sequences are labeled over the alphabet $\{A, C, G, T\}$, and there are 4^n possible genomic sequences of length n. Amino acid sequences, on the other hand, are labeled over the alphabet $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$, and there are 20^n possible proteomic sequences of length n. An interesting problem is the generation of all the genomic sequences with n nucleotides or all the proteomic sequences with n amino acids, that is, the generation of all the words of length n over an alphabet Σ.

Write code for the words problem. The program must implement and use the WORDS function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is an integer n and an alphabet Σ.

Output

The output is a sorted list of all the words of length n over the alphabet Σ.

Sample input 1

```
1
G T A C
```

Sample output 1

```
A
C
G
T
```

Sample input 2

```
2
G T A C
```

Sample output 2

```
AA
AC
AG
AT
CA
CC
CG
CT
GA
GC
GG
GT
TA
TC
TG
TT
```

Sample input 3

```
3
G T A C
```

Sample output 3

```
AAA
AAC
AAG
AAT
```

```
function WORDS(n, Σ)
    sort Σ
    let L be an empty list (of strings)
    append an empty string to L
    for i = 1 to n do
        let L′ be an empty list (of strings)
        for all w ∈ L do
            for all x ∈ Σ do
                let w′ be the concatenation of x and w
                append w′ to L′
        L = L′
    return L
```

```
import sys

def words(n, a):
  a.sort()
  L = ['']
  for i in range(n):
    LL = []
    for w in L:
      for x in a:
        LL += [x + w]
    L = LL
  return L

n = int(sys.stdin.readline().strip())
a = sys.stdin.readline().strip().split()
for w in words(n, a):
  print(w)
```

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

vector<string> words(int n, vector<char>& A);

int main() {
  int n;
  cin >> n;
  vector<char> A;
  char x;
  while (cin >> x) A.push_back(x);
  vector<string> L;
  words(n, A, L);
  for (auto w : L) cout << w << endl;
}
```

```
vector<string> words(int n, vector<char>& A) {
  sort(A.begin(), A.end());
  vector<string> L;
  L.push_back("");
  for (int i = 0; i < n; ++i) {
    vector<string> LL;
    for (auto w : L)
      for (auto x : A)
        LL.push_back({x + w});
    L = LL;
  }
  return L;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Words 2 X86108_en

Nucleic acid sequences are labeled over the alphabet $\{A, C, G, T\}$, and there are 4^n possible genomic sequences of length n. Amino acid sequences, on the other hand, are labeled over the alphabet $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$, and there are 20^n possible proteomic sequences of length n. An interesting problem is the generation of all the genomic sequences with n nucleotides or all the proteomic sequences with n amino acids, that is, the generation of all the words of length n over an alphabet Σ.

Write code for the words problem. The program must implement and use the WORDS function in the pseudocode discussed in class, which is recursive and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is an integer n and an alphabet Σ.

Output

The output is a sorted list of all the words of length n over the alphabet Σ.

Sample input 1

```
1
G T A C
```

Sample output 1

```
A
C
G
T
```

Sample input 2

```
2
G T A C
```

Sample output 2

```
AA
AC
AG
AT
CA
CC
CG
CT
GA
GC
GG
GT
TA
TC
TG
TT
```

Sample input 3

```
3
G T A C
```

Sample output 3

```
AAA
AAC
AAG
AAT
```

```
function WORDS(n, Σ)
    sort Σ
    let L be an empty list
    if n = 0 then
        append an empty string to L
    else
        W = WORDS(n − 1, Σ)
        for all w ∈ W do
            for all x ∈ Σ do
                let w′ be the concatenation of x and w
                append w′ to L
    return L
```

```
import sys

def words(n, a):
  a.sort()
  L = []
  if n == 0:
    L = ['']
  else:
    W = words(n - 1, a)
    for w in W:
      for x in a:
        L += [x + w]
  return L

n = int(sys.stdin.readline().strip())
a = sys.stdin.readline().strip().split()
for w in words(n, a):
  print(w)
```

```
#include <iostream>
#include <algorithm>
#include <sstream>
#include <vector>
using namespace std;

vector<string> words(int n, vector<char>& A);

int main() {
  string s;
  getline(cin, s);
  int n = stoi(s);
  getline(cin, s);
  istringstream ss(s);
  vector<char> A;
  char x;
  while (ss >> x) A.push_back(x);
  vector<string> L = words(n, A);
  for (auto w : L) cout << w << endl;
}
```

```
vector<string> words(int n, vector<char>& A) {
  sort(A.begin(), A.end());
  vector<string> L;
  if (n == 0) L.push_back("");
  else {
    vector<string> W = words(n - 1, A);
    for (auto w : W)
      for (auto x : A)
        L.push_back(x + w);
  }
  return L;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

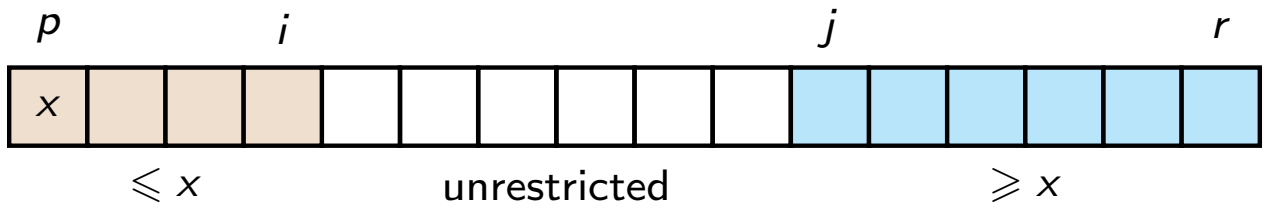
The quicksort algorithm is often the best practical choice for sorting because it is remarkably efficient on average. It also has the advantage of sorting in place, and it works well even in virtual-memory environments.

The procedure QUICKSORT sorts array $A[1 : n]$.

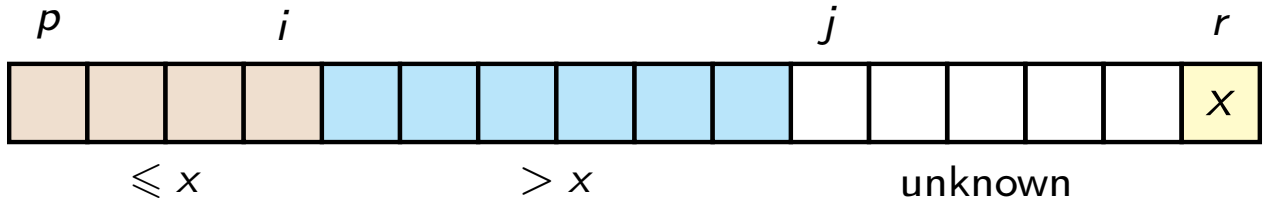
```
procedure QUICKSORT(A)
    QUICKSORT(A, 1, n)

procedure QUICKSORT(A, p, r)
    if p < r then
        partition the subarray around the pivot, which ends up in A[q]
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)          recursively sort the low side
        QUICKSORT(A, q + 1, r)          recursively sort the high side
```

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p : r]$ in place, returning the index of the dividing point between the two sides of the partition.

```
function PARTITION(A, p, r)
    x = A[p]
    i = p − 1
    j = r + 1
    while true do
        repeat
            j = j − 1
        until A[j] ⩽ x
        repeat
            i = i + 1
        until A[i] ⩾ x
        if i < j then
            exchange A[i] with A[j]
        else
            return j
```

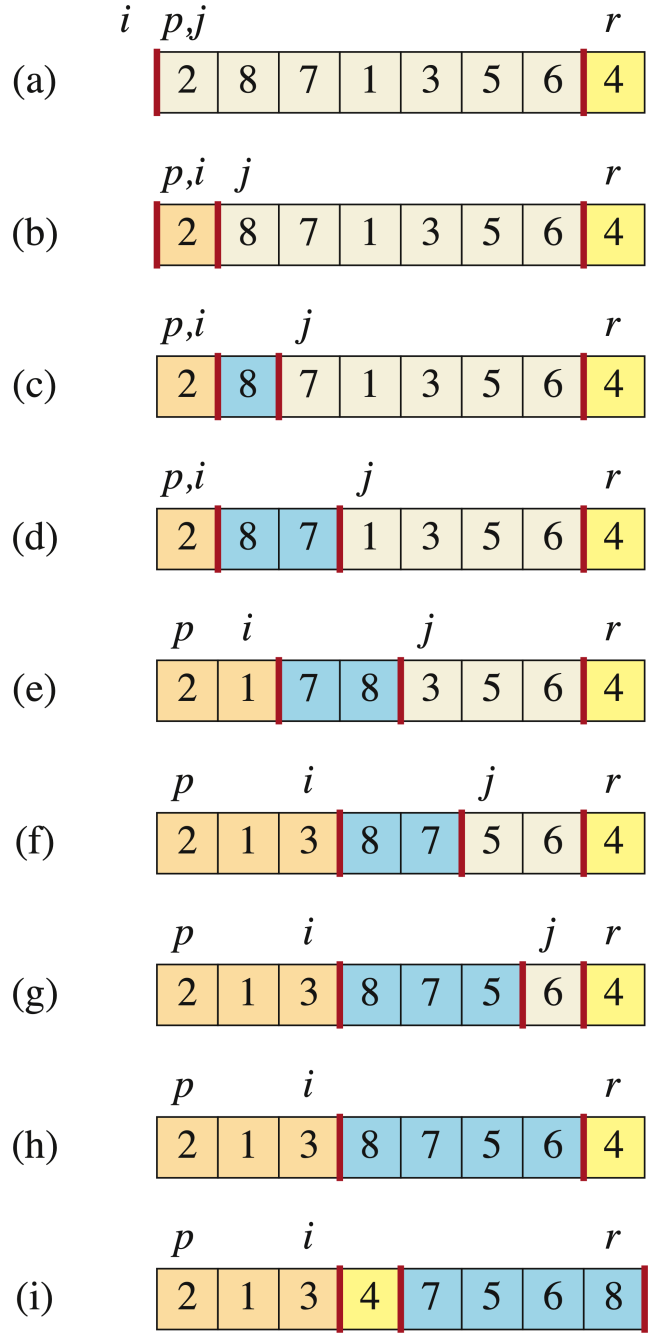


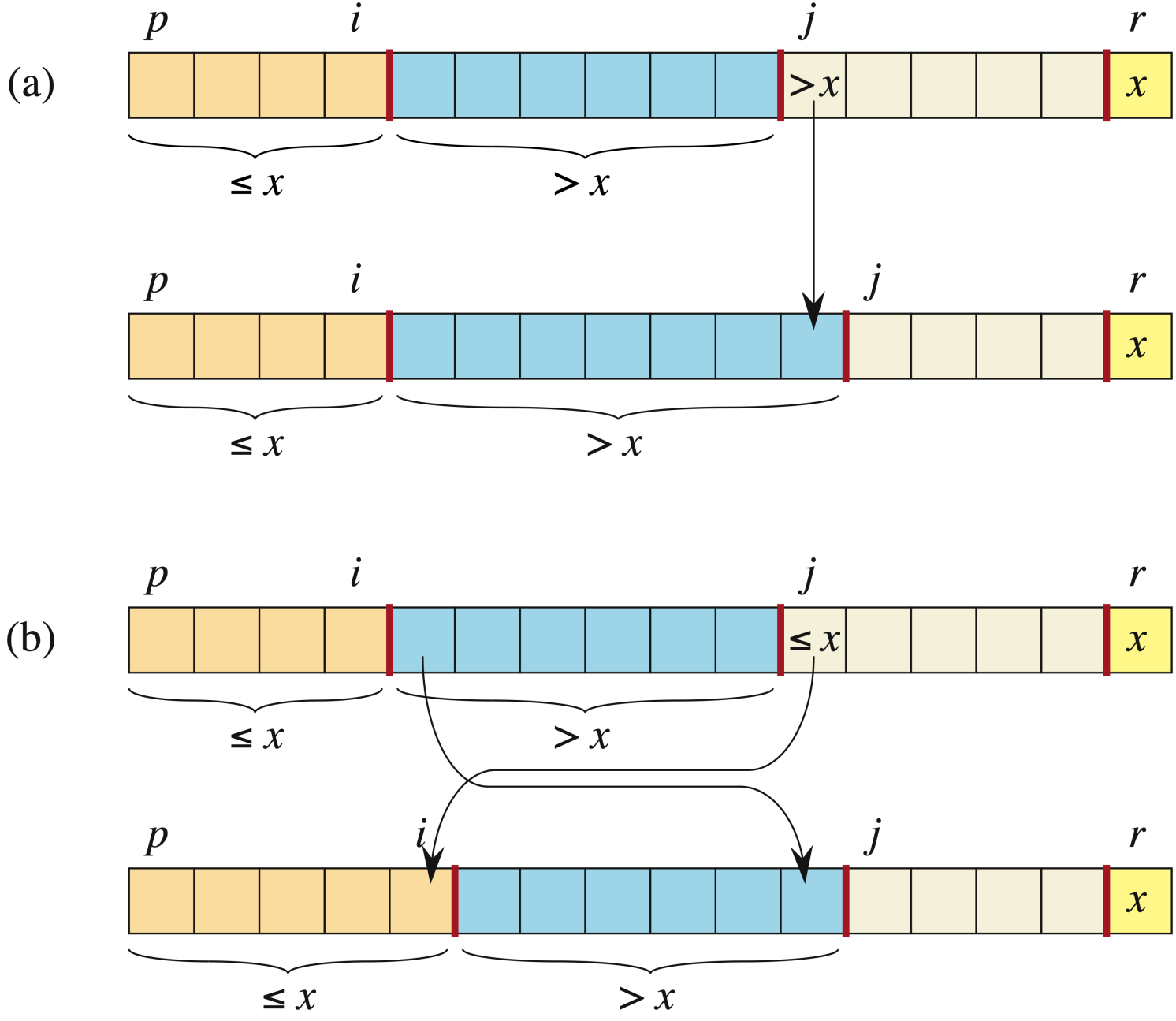
```
function PARTITION(A, p, r)
    x = A[r]                                  the pivot
    i = p − 1                                 highest index into the low side
    for j = p to r − 1 do                     process each element other than the pivot
        if A[j] ⩽ x then                      does this element belong on the low side?
            i = i + 1                         index of a new slot in the low side
            exchange A[i] with A[j]           put this element there
    exchange A[i + 1] with A[r]               pivot goes just right of the low side
    return i + 1                              new index of the pivot
```



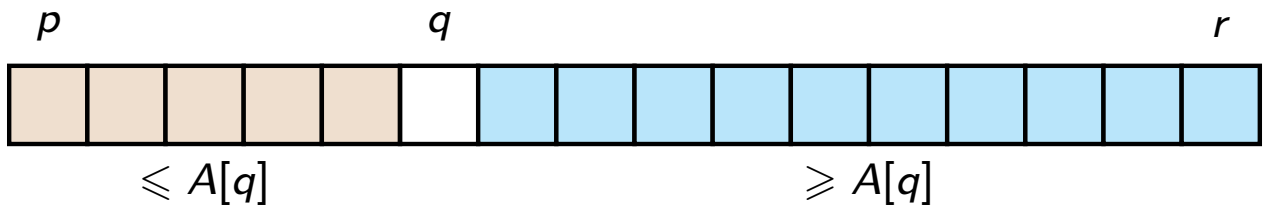
At the beginning of each iteration of the **for** loop, for any array index k, the following conditions hold:

- if $p \leqslant k \leqslant i$, then $A[k] \leqslant x$ — the tan region
- if $i + 1 \leqslant k \leqslant j - 1$, then $A[k] > x$ — the cyan region
- if $k = r$, then $A[k] = x$ — the yellow region


i p,j r
(a) 2 8 7 1 3 5 6 4
p,i j r
(b) 2 8 7 1 3 5 6 4
p,i j r
(c) 2 8 7 1 3 5 6 4
p,i j r
(d) 2 8 7 1 3 5 6 4
p i j r
(e) 2 1 7 8 3 5 6 4
p i j r
(f) 2 1 3 8 7 5 6 4
p i j r
(g) 2 1 3 8 7 5 6 4
p i r
(h) 2 1 3 8 7 5 6 4
p i r
(i) 2 1 3 4 7 5 6 8


(a)
p
i
j
r
> x
x
≤ x
> x
p
i
j
r
x
≤ x
> x
(b)
p
i
j
r
≤ x
x
≤ x
> x
p
i
j
r
x
≤ x
> x

The quicksort algorithm rearranges the array $A[p : r]$ into two (possibly empty) subarrays $A[p : q - 1]$ (the low side) and $A[q + 1 : r]$ (the high side) such that each element in the low side of the partition is less than or equal to the pivot $A[q]$, which is, in turn, less than or equal to each element in the high side.



Then, it calls quicksort recursively to sort each of the subarrays $A[p : q - 1]$ and $A[q + 1 : r]$.

All elements in $A[p : q - 1]$ are sorted and less than or equal to the pivot $A[q]$, and all elements in $A[q + 1 : r]$ are sorted and greater than or equal to the pivot $A[q]$.

Therefore, the entire array $A[p : q]$ is sorted.

The merge sort algorithm sorts, in each step, a subarray $A[p : q]$, starting with the entire array $A[1 : n]$ and recursing down to smaller subarrays.

The procedure MERGE-SORT sorts array $A[1 : n]$.

```
procedure MERGE-SORT(A)
    MERGE-SORT(A, 1, n)

procedure MERGE-SORT(A, p, r)
    if p ⩾ r then                                   zero or one element?
        return
    q = ⌊(p + r)/2⌋                                 midpoint of A[p : r]
    MERGE-SORT(A, p, q)                             recursively sort A[p : q]
    MERGE-SORT(A, q + 1, r)                         recursively sort A[q + 1 : r]
    MERGE(A, p, q, r)       merge A[p : q] and A[q + 1 : r] into A[p : r]
```

The key operation of the merge sort algorithm occurs in the merge step, which merges two adjacent, sorted subarrays.

```
procedure MERGE(A, p, q, r)
    n_L = q − p + 1                                   length of A[p : q]
    n_R = r − q                                       length of A[q + 1 : r]
    let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
    for i = 0 to n_L − 1 do                           copy A[p : q] into L[0 : n_L − 1]
        L[i] = A[p + i]
    for j = 0 to n_R − 1 do                           copy A[q + 1 : r] into R[0 : n_R − 1]
        R[j] = A[q + j + 1]
    i = 0                                             i indexes the smallest remaining element in L
    j = 0                                             j indexes the smallest remaining element in R
    k = p                                             k indexes the location in A to fill
    as long as each of the arrays L and R contains an unmerged element,
                                                      copy the smallest unmerged element back into A[p : r]
    while i < n_L and j < n_R do
        if L[i] ⩽ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        k = k + 1
```

having gone through one of L and R entirely, copy the remainder of the other to the end of $A[p : r]$

while $i < n_L$ **do**
    $A[k] = L[i]$
    $i = i + 1$
    $k = k + 1$
while $j < n_R$ **do**
    $A[k] = R[j]$
    $j = j + 1$
    $k = k + 1$

The MERGE procedure copies the two subarrays $A[p : q]$ and $A[q + 1 : r]$ into temporary arrays L and R, and then it merges the values in L and R back into $A[p : r]$.

The first lines compute the lengths n_L and n_R of the subarrays $A[p : q]$ and $A[q + 1 : r]$, and create arrays $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ with lengths n_L and n_R.

The first **for** loop copies the subarray $A[p : q]$ into L, and the second **for** loop copies the subarray $A[q + 1 : r]$ into R.

The first **while** loop repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p : r]$ and copies it back in.

Eventually, either all of L or all of R is copied back into $A[p : r]$, and this loop terminates.

If the loop terminates because all of R has been copied back, then some of L has yet to be copied back, and the second **while** loop copies these remaining values of L back into the end of $A[p : r]$.

If instead the loop terminates because all of L has been copied back, then some of R has yet to be copied back, and the third **while** loop copies these remaining values of R back into the end of $A[p : r]$.

The Fibonacci numbers F_n are defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geqslant 2 \end{cases}$$

Therefore, the first Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Give the pseudocode of a recursive algorithm to compute F_n for a natural number n.

```
function FIBONACCI(n)
    if n ⩽ 1 then
        return n
    else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

The Towers of Hanoi is a game that consists of three rods and n disks of different sizes that can slide onto any rod. The game starts with the disks stacked in order of size on the left rod, the biggest disk at the bottom. The aim of the game is to move all the disks from the left rod to the right rod, using the middle rod as auxiliary. All the moves have to follow these rules:

- In each step, only one disk can be moved.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, over the other disks that may already be present on that rod.
- No disk can be placed over a smaller disk.

Give the pseudocode of a recursive algorithm to solve the game of the Towers of Hanoi, using the minimal number of movements.

```
procedure HANOI(n, left, middle, right)
    if n > 0 then
        HANOI(n − 1, left, right, middle)
        output left → right
        HANOI(n − 1, middle, left, right)
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Subwords 1 X85229_en

A nucleic acid or amino acid sequence of length n can be seen as composed of a number of possibly overlapping k-mers or words of length k, for $1 \leq k \leq n$. An interesting problem is the generation of all the words of length k contained in a genomic sequence with n nucleotides, for all k with $1 \leq k \leq n$. That is, the generation of all the subwords of a genomic sequence of length n.

Write code for the subwords problem. The program must implement and use the SUBWORDS function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is a sorted list of all the nonempty subwords of s, without repetitions.

Sample input

```
TATAAT
```

Sample output

```
A
AA
AAT
AT
ATA
ATAA
ATAAT
T
TA
TAA
TAAT
TAT
TATA
TATAA
TATAAT
```

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:29:59

```
function SUBWORDS(s)
    let S be an empty set (of strings)
    for i = 1 to |s| do
        for j = i to |s| do
            insert s[i : j] in S
    let L be an empty list (of strings)
    for all w ∈ S do
        append w to L
    sort L
    return L
```

```
import sys

def subwords(s):
  S = set()
  for i in range(len(s)):
    for j in range(i, len(s)):
      S.add(s[i : j + 1])
  return sorted(S)

s = sys.stdin.readline().strip()
L = subwords(s)
for w in L:
  print(w)
```

```
#include <iostream>
#include <algorithm>
#include <set>
#include <vector>
using namespace std;

vector<string> subwords(string s);

int main() {
  string s;
  cin >> s;
  vector<string> L = subwords(s);
  for (auto w : L) cout << w << endl;
}
```

```
vector<string> subwords(string s) {
  int n = s.length();
  set<string> S;
  for (int i = 0; i < n; ++i) {
    for (int j = i; j < n; ++j) {
      string w = s.substr(i, j - i + 1);
      S.insert(w);
  } }
  vector<string> L;
  for (auto x : S) L.push_back(x);
  sort(L.begin(), L.end());
  return L;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Subwords 2 — X43423_en

A nucleic acid or amino acid sequence of length n can be seen as composed of a number of possibly overlapping k-mers or words of length k, for $1 \leq k \leq n$. An interesting problem is the generation of all the words of length k contained in a genomic sequence with n nucleotides, for all k with $1 \leq k \leq n$. That is, the generation of all the subwords of a genomic sequence of length n.

Write code for the subwords problem. The program must implement and use the SUBWORDS function in the pseudocode discussed in class, which is recursive and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is a sorted list of all the nonempty subwords of s, without repetitions.

Sample input

```
TATAAT
```

Sample output

```
A
AA
AAT
AT
ATA
ATAA
ATAAT
T
TA
TAA
TAAT
TAT
TATA
TATAA
TATAAT
```

Problem information

Author : Gabriel Valiente
Generation : 2022-07-07 18:30:28

```
function SUBWORDS(s)
    S_1, S_2 = SUBWORDS(s, 1)
    let L be an empty list
    for all w ∈ S_2 do
        append w to L
    sort L
    return L

function SUBWORDS(s, i)
    if i > |s| then
        return ∅, ∅
    else
        S_1, S_2 = SUBWORDS(s, i + 1)
        insert the empty string in S_1
        let S'_1 be an empty set (of strings)
        for all w ∈ S_1 do
            let w' be the concatenation of s[i] and w
            insert w' in S'_1          subwords of s starting at position i
            insert w' in S_2           subwords of s starting at a position ⩾ i
        return S'_1, S_2
```

- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 1)$
  - $S_1 = \{\text{T}, \text{TA}, \text{TAT}, \text{TATA}, \text{TATAA}, \text{TATAAT}\}$
  - $S_2 = \{\text{A}, \text{AA}, \text{AAT}, \text{AT}, \text{ATA}, \text{ATAA}, \text{ATAAT}, \text{T}, \text{TA}, \text{TAA}, \text{TAAT}, \text{TAT}, \text{TATA}, \text{TATAA}, \text{TATAAT}\}$
- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 2)$
  - $S_1 = \{\text{A}, \text{AT}, \text{ATA}, \text{ATAA}, \text{ATAAT}\}$
  - $S_2 = \{\text{A}, \text{AA}, \text{AAT}, \text{AT}, \text{ATA}, \text{ATAA}, \text{ATAAT}, \text{T}, \text{TA}, \text{TAA}, \text{TAAT}\}$
- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 3)$
  - $S_1 = \{\text{T}, \text{TA}, \text{TAA}, \text{TAAT}\}$
  - $S_2 = \{\text{A}, \text{AA}, \text{AAT}, \text{AT}, \text{T}, \text{TA}, \text{TAA}, \text{TAAT}\}$
- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 4)$
  - $S_1 = \{\text{A}, \text{AA}, \text{AAT}\}$
  - $S_2 = \{\text{A}, \text{AA}, \text{AAT}, \text{AT}, \text{T}\}$
- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 5)$
  - $S_1 = \{\text{A}, \text{AT}\}$
  - $S_2 = \{\text{A}, \text{AT}, \text{T}\}$
- $S_1, S_2 = \text{SUBWORDS}(\text{TATAAT}, 6)$
  - $S_1 = \{\text{T}\}$
  - $S_2 = \{\text{T}\}$

```
import sys

def subwords(s):
  S1, S2 = subwords_rec(s, 0)
  return sorted([x for x in S2 if len(x) != 0])

def subwords_rec(s, i):
  if i == len(s):
    return set(), set()
  else:
    S1, S2 = subwords_rec(s, i + 1)
    S1.add('')
    SS1 = set()
    for w in S1:
      ww = s[i] + w
      SS1.add(ww)
      S2.add(ww)
    return SS1, S2

s = sys.stdin.readline().strip()
for w in subwords(s):
  print(w)
```

```
#include <iostream>
#include <algorithm>
#include <set>
#include <vector>
using namespace std;

vector<string> subwords(string s);

pair<set<string>, set<string>>
subwords(string s, int i);

int main() {
  string s;
  cin >> s;
  vector<string> S = subwords(s);
  for (auto w : S) cout << w << endl;
}
```

```
vector<string> subwords(string s) {
  pair<set<string>, set<string>> P = subwords(s, 0);
  vector<string> L;
  for (auto x : P.second) L.push_back(x);
  sort(L.begin(), L.end());
  return L;
}
```

```
pair<set<string>, set<string>>
subwords(string s, int i) {
  if (i == s.length()) {
    set<string> S1, S2;
    return make_pair(S1, S2);
  } else {
    pair<set<string>, set<string>>
    P = subwords(s, i + 1);
    P.first.insert("");
    set<string> SS1;
    for (auto w : P.first) {
      string ww = s[i] + w;
      SS1.insert(ww);
      P.second.insert(ww);
    }
    return make_pair(SS1, P.second);
  }
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, 4th edition, 2022

When an algorithm contains a recursive call, you can often describe its running time by a recurrence, which describes the overall running time on a problem of size n in terms of the running time of the same algorithm on smaller inputs.

Let $T(n)$ be the worst-case running time on a problem of size n.

If the problem size is small enough, say $n < n_0$ for some constant $n_0 > 0$, the straightforward solution takes constant time, which we write as $\Theta(1)$.

Otherwise, suppose the algorithm makes a recursive calls, each on a subproblem of size n/b, that is, $1/b$ the size of the original.

It takes $T(n/b)$ time to solve one subproblem of size n/b, and so it takes $aT(n/b)$ to solve all a of them.

We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ aT(n/b) + \Theta(\ldots) & \text{otherwise} \end{cases}$$

The substitution method for solving recurrences consists of guessing the form of the solution using symbolic constants, and using mathematical induction to show that the solution works, and find the constants.

The recursion-tree method for solving recurrences consists of drawing up a recursion tree (where each node represents the cost of a single subproblem somewhere in the set of recursive function invocations) to generate intuition for a good guess, which you can then verify by the substitution method.

The master method for solving recurrences consists of memorizing three cases of recurrences of the form $T(n) = aT(n/b) + f(n)$, which describe the running time of algorithms that divide a problem of size n into a subproblems, each of size $n/b < n$, which are solved recursively, and the function $f(n)$ encompasses the cost of dividing the problem before the recursion, as well as the cost of combining the results of the recursive solutions to subproblems.

```
function LINEAR-SEARCH(A, x, low, high)
    i = low
    while i ⩽ high and x ≠ A[i] do
        i = i + 1
    if i > high then
        return nil
    return i

function LINEAR-SEARCH(A, x, low, high)
    if low > high then
        return nil
    else if x = A[low] then
        return low
    else
        return LINEAR-SEARCH(A, x, low + 1, high)
```

Both procedures terminate the search unsuccessfully when the range is empty (that is, $low > high$) and terminate it successfully if the value x has been found.

Based on the comparison of x to the first element in the searched range, the search continues with the remaining elements in the range.

The recurrence for these procedures is therefore

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n-1) + \Theta(1) & \text{otherwise} \end{cases}$$

whose solution is $\Theta(n)$.

Denote by c the constant hidden in the $\Theta(1)$ term.

Guess that $T(n) \leqslant dn$ for a constant d to be chosen. We have

$$T(n) \leqslant T(n-1) + c \leqslant d(n-1) + c = dn - d + c$$

Now, $dn - d + c \leqslant dn$ if $-d + c \leqslant 0$, which is equivalent to $d \geqslant c$, and this holds for all n.

Guess that $T(n) \geqslant dn$ for a constant d to be chosen. We have

$$T(n) \geqslant T(n-1) + c \geqslant d(n-1) + c = dn - d + c$$

Now, $dn - d + c \geqslant dn$ if $-d + c \geqslant 0$, which is equivalent to $d \leqslant c$, and this holds for all n.

Thus, $T(n) = \Theta(n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

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

is $T(n) = n$.

The base case is when $n = 2$, and we have $n = 2$.

For the inductive step, our hypothesis is that $T(n-1) = n - 1$. Then

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= n - 1 + 1 \\ &= n \end{aligned}$$

which completes the inductive proof for exact powers of 2.

```
function BINARY-SEARCH(A, x, low, high)
    while low ⩽ high do
        mid = ⌊(low + high)/2⌋
        if x = A[mid] then
            return mid
        else if x > A[mid] then
            low = mid + 1
        else
            high = mid − 1
    return nil
```

```
function BINARY-SEARCH(A, x, low, high)
    if low > high then
        return nil
    m = ⌊(low + high)/2⌋
    if x = A[m] then
        return m
    else if x > A[m] then
        return BINARY-SEARCH(A, x, m + 1, high)
    else
        return BINARY-SEARCH(A, x, low, m − 1)
```

Both procedures terminate the search unsuccessfully when the range is empty (that is, $low > high$) and terminate it successfully if the value x has been found.

Based on the comparison of x to the middle element in the searched range, the search continues with the range halved.

The recurrence for these procedures is therefore

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n/2) + \Theta(1) & \text{otherwise} \end{cases}$$

whose solution is $\Theta(\lg n)$.

Denote by c the constant hidden in the $\Theta(1)$ term.

Guess that $T(n) \leqslant c \lg n$. We have

$$T(n) \leqslant T(n/2) + c \leqslant c \lg(n/2) + c = c \lg n - c \lg 2 + c = c \lg n$$

and this holds for all n.

Guess that $T(n) \geqslant c \lg n$. We have

$$T(n) \geqslant T(n/2) + c \geqslant c \lg(n/2) + c = c \lg n - c \lg 2 + c = c \lg n$$

and this holds for all n.

Thus, $T(n) = \Theta(\lg n)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ T(n/2) + 1 & \text{if } n > 2 \end{cases}$$

is $T(n) = \lg n$.

The base case is when $n = 2$, and we have $\lg n = \lg 2 = 1$.

For the inductive step, our hypothesis is that $T(n/2) = \lg(n/2)$. Then

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= \lg(n/2) + 1 \\ &= (\lg n - \lg 2) + 1 \\ &= \lg n - 1 + 1 \\ &= \lg n \end{aligned}$$

which completes the inductive proof for exact powers of 2.

The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n - 1$ elements and one with 0 elements.

Let us assume that this unbalanced partitioning arises in each recursive call.

```
function PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] ⩽ x then
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

The running time of PARTITION on a subarray $A[p : r]$ of $n = r - p + 1$ elements is $\Theta(n)$, because the **for** loop iterates $n - 1$ times, each iteration takes $\Theta(1)$ time, and the parts of the procedure outside the loop take $\Theta(1)$ time, for a total of $\Theta(n)$ time.

Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ T(n-1) + \Theta(n) & \text{otherwise} \end{cases}$$

whose solution is $T(n) = \Theta(n^2)$.

Denote by c the constant hidden in the $\Theta(n)$ term.

Guess that $T(n) \leqslant dn^2$ for a constant d to be chosen. We have

$$T(n) \leqslant T(n-1) + cn \leqslant d(n-1)^2 + cn = dn^2 - 2dn + d + cn$$

Now, $dn^2 - 2dn + d + cn \leqslant dn^2$ if $-2dn + d + cn \leqslant 0$, which is equivalent to $d \geqslant cn/(2n-1)$, and this holds for all $n \geqslant 1$ and $d \geqslant c$.

Guess that $T(n) \geqslant dn^2$ for a constant d to be chosen. We have

$$T(n) \geqslant T(n-1) + cn \geqslant d(n-1)^2 + cn = dn^2 - 2dn + d + cn$$

Now, $dn^2 - 2dn + d + cn \geqslant dn^2$ if $-2dn + d + cn \geqslant 0$, which is equivalent to $d \leqslant cn/(2n-1)$, and this holds for all $n \geqslant 1$ and $d \leqslant c/2$.

Thus, $T(n) = \Theta(n^2)$.

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

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

is $T(n) = (n-1)n/2 + 2$.

The base case is when $n = 1$, and we have $(n-1)n/2 + 2 = 0 + 2 = 2$.

For the inductive step, our hypothesis is that
$T(n-1) = (n-2)(n-1)/2 + 2$. Then

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ &= (n-2)(n-1)/2 + 2 + n - 1 \\ &= n(n-1)/2 - 2(n-1)/2 + 2 + n - 1 \\ &= n(n-1)/2 - (n-1) + 2 + (n-1) \\ &= (n-1)n/2 + 2 \end{aligned}$$

which completes the inductive proof for exact powers of 2.

Let $T(n)$ be the worst-case running time of merge sort on n numbers.

Since the MERGE procedure on an n-element subarray takes $\Theta(n)$ time, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

whose solution is $T(n) = \Theta(n \lg n)$.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of the assignment statements takes constant time, and the **for** loops take $\Theta(n_L + n_R) = \Theta(n)$ time.

To account for the three **while** loops, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once.

Therefore, these three loops together make a total of n iterations.

Since each iteration of each of the three loops takes constant time, the total time spent on these three loops is $\Theta(n)$.

To see that the solution is $T(n) = \Theta(n \lg n)$, assume that n is an exact power of 2 and that the base case is $n = 1$.

Then the recurrence is essentially

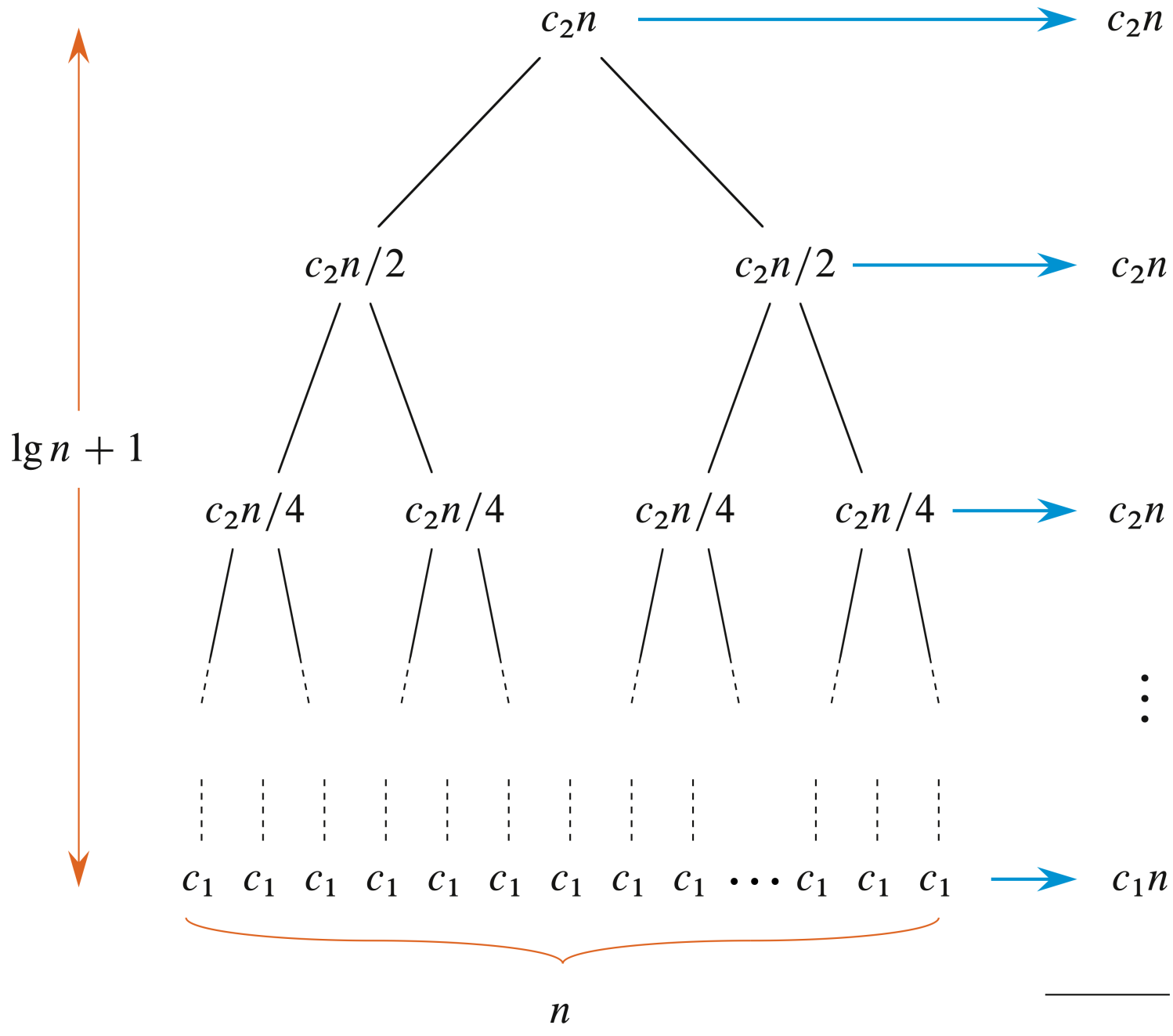
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

where the constant $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the nonrecursive steps.

We can figure out the solution by means of an equivalent recursion tree representing the recurrence, where each node in the tree is expanded by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1.

The total number of levels of the recursion tree is $\lg n + 1$, where n is the number of leaves, corresponding to the input size.

The levels above the leaves each cost $c_2 n$, and the leaf level costs $c_1 n$, for a total cost of $c_2 n \lg n + c_1 n = \Theta(n \lg n)$.


c_2n
c_2n
$c_2n/2$
$c_2n/2$
c_2n
$\lg n + 1$
$c_2n/4$
$c_2n/4$
$c_2n/4$
$c_2n/4$
c_2n
c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 c_1 $\cdots$ c_1 c_1 c_1
c_1n
n
Total: $c_2 \lg n + c_1 n$

We can use mathematical induction to show that when $n \geqslant 2$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n > 2 \end{cases}$$

is $T(n) = n \lg n$.

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our hypothesis is that $T(n/2) = (n/2) \lg(n/2)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2) \lg(n/2) + n \\ &= n(\lg n - \lg 2) + n \\ &= n(\lg n - 1) + n \\ &= n \lg n - n + n \\ &= n \lg n \end{aligned}$$

which completes the inductive proof for exact powers of 2.

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Permutations 1 — X48980_en

Some genome rearrangements change the order of the nucleotides in a nucleic acid sequence, resulting in a permutation of the nucleic acid sequence. For example, TATATA is a frequent rearrangement of TATAAT. An interesting problem is the generation of all the permutations of a genomic sequence of length n.

Write code for the permutations problem. The program must implement and use the PERMUTATIONS function in the pseudocode discussed in class, which is iterative and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is a sorted list of all the permutations of s, without repetitions.

Sample input 1

```
ACGT
```

Sample output 1

```
ACGT
ACTG
AGCT
AGTC
ATCG
ATGC
CAGT
CATG
CGAT
CGTA
CTAG
CTGA
GACT
GATC
GCAT
GCTA
GTAC
GTCA
TACG
TAGC
TCAG
TCGA
TGAC
TGCA
```

Sample input 2

```
TATAAT
```

Sample output 2

```
AAATTT
AATATT
AATTAT
AATTTA
ATAATT
```

```
function PERMUTATIONS(s)
    let S be an empty stack (of characters)
    for i = 1 to |s| do
        push s[i] onto S
    let L be an empty set (of strings)
    insert an empty string in L
    while S ≠ ∅ do
        pop from S the top element x
        let L′ be an empty set (of strings)
        for all w ∈ L do
            for i = 1 to |w| + 1 do
                insert w[1 : i − 1], x, w[i : |w|] in L′
        L = L′
    let A be an empty list (of strings)
    for all w ∈ L do
        append w to A
    sort A
    return A
```

- PERMUTATIONS(s)

  obtain all permutations of s by inserting a character of s at every possible position in each permutation of the remaining characters

- $S = \boxed{\begin{array}{c}\mathsf{C}\\ \mathsf{B}\\ \mathsf{A}\end{array}}$

  $L = \{\text{“”}\}$

- $S = \boxed{\begin{array}{c}\mathsf{B}\\ \mathsf{A}\end{array}}$

  $x = \mathsf{C}$

  $L = \{\mathsf{C}\}$

- $S = \boxed{\mathsf{A}}$

  $x = \mathsf{B}$

  $L = \{\mathsf{BC}, \mathsf{CB}\}$

- $S = \emptyset$

  $x = \mathsf{A}$

  $L = [\mathsf{ABC}, \mathsf{BAC}, \mathsf{BCA}, \mathsf{ACB}, \mathsf{CAB}, \mathsf{CBA}]$

```
import sys

def permutations(s):
  stack = list(s)
  L = {''}
  while len(stack) != 0:
    x = stack.pop()
    LL = set()
    for w in L:
      for i in range(len(w) + 1):
        LL.add(w[:i] + x + w[i:])
    L = LL
  return sorted(L)

s = sys.stdin.readline().strip()
for p in permutations(s):
  print(p)
```

```
vector<string> permutations(string s) {
  stack<char> S;
  for (int i = 0; i < int(s.length()); ++i)
    S.push(s[i]);
  set<string> L = {""};
  while (not S.empty()) {
    char x = S.top(); S.pop();
    set<string> LL;
    for (string w : L)
      for (int i = 0; i <= int(w.length()); ++i)
        LL.insert(w.substr(0, i) + x + w.substr(i));
    L = LL;
  }
  vector<string> A; for (string w : L) A.push_back(w);
  sort(A.begin(), A.end()); return A;
}

int main() { string s; cin >> s;
  vector<string> L = permutations(s);
  for (string x : L) cout << x << endl;
}
```

T1. Structured pseudocode
L1. Problem 1 (X66236, Gene finding)
L1. Problem 2 (X48860, Gene enumeration)
T2. Elementary algorithms and data structures
L2. Problem 3 (X95757, RNA to protein)
L2. Problem 4 (X28783, Read abundance)
T3. Analysis of iterative algorithms
L3. Problem 5 (X89781, Read mapping 1)
L3. Problem 6 (X38913, Read mapping 2)
T4. Linear recursion
L4. Problem 7 (X38951, Sliding window 1)
L4. Problem 8 (X99827, Sliding window 2)
L5. Problem 9 (X11585, Words 1)
L5. Problem 10 (X86108, Words 2)
T5. Multiple recursion
L6. Problem 11 (X85229, Subwords 1)
L6. Problem 12 (X43423, Subwords 2)
T6. Analysis of recursive algorithms
L7. Problem 13 (X48980, Permutations 1)
L7. Problem 14 (X87168, Permutations 2)

Permutations 2 — X87168_en

Some genome rearrangements change the order of the nucleotides in a nucleic acid sequence, resulting in a permutation of the nucleic acid sequence. For example, TATATA is a frequent rearrangement of TATAAT. An interesting problem is the generation of all the permutations of a genomic sequence of length n.

Write code for the permutations problem. The program must implement and use the PERMUTATIONS function in the pseudocode discussed in class, which is recursive and is not allowed to perform input/output operations. Make one submission with Python code and another submission with C++ code.

Input

The input is a string s over the alphabet $\Sigma = \{A, C, G, T\}$.

Output

The output is a sorted list of all the permutations of s, without repetitions.

Sample input 1

```
ACGT
```

Sample output 1

```
ACGT
ACTG
AGCT
AGTC
ATCG
ATGC
CAGT
CATG
CGAT
CGTA
CTAG
CTGA
GACT
GATC
GCAT
GCTA
GTAC
GTCA
TACG
TAGC
TCAG
TCGA
TGAC
TGCA
```

Sample input 2

```
TATAAT
```

Sample output 2

```
AAATTT
AATATT
AATTAT
AATTTA
ATAATT
```

```
function PERMUTATIONS(s)
    let R be an empty set (of strings)
    PERMUTATIONS(s, 1, R)
    return R

procedure PERMUTATIONS(s, k, R)
    if k = |s| then
        insert s in R
    else
        for i = k to |s| do
            exchange s[k] with s[i]
            PERMUTATIONS(s, k + 1, R)
```

- PERMUTATIONS(s, k)

  obtain all permutations of s changing only the characters in positions $k, \ldots, |s|$
- When $k = 1$, generate all permutations of s
- When $k = |s|$, there is only one character left, and only one way to arrange a single character
- Otherwise, exchange the k-th character with the i-th character

- PERMUTATIONS(ABC, 1)
  - PERMUTATIONS(ABC, 2)
    - PERMUTATIONS(ABC, 3)
    - PERMUTATIONS(ACB, 3)
  - PERMUTATIONS(BAC, 2)
    - PERMUTATIONS(BAC, 3)
    - PERMUTATIONS(BCA, 3)
  - PERMUTATIONS(CBA, 2)
    - PERMUTATIONS(CBA, 3)
    - PERMUTATIONS(CAB, 3)

```
import sys

def permutations(s):
  result = set()
  permutations_rec(list(s), 0, result)
  return result

def permutations_rec(s, k, result):
  if k == len(s) - 1:
    result.add(''.join(s))
  else:
    for i in range(k, len(s)):
      s[k], s[i] = s[i], s[k]
      permutations_rec(s, k + 1, result)
      s[i], s[k] = s[k], s[i]

s = sys.stdin.readline().strip()
for t in sorted(permutations(s)):
  print(t)
```

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

void permutations(string s, int k, set<string>& R) {
  if (k == int(s.length()) - 1) R.insert(s);
  else for (int i = k; i < int(s.length()); ++i) {
    swap(s[k], s[i]);
    permutations(s, k + 1, R);
} }

set<string> permutations(string s) {
  set<string> R;
  permutations(s, 0, R);
  return R;
}

int main() { string s; cin >> s;
  set<string> R = permutations(s);
  for (string t : R) cout << t << endl;
}
```

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

void permutations(string& s, int k, set<string>& R) {
  if (k == int(s.length()) - 1) R.insert(s);
  else for (int i = k; i < int(s.length()); ++i) {
    swap(s[k], s[i]);
    permutations(s, k + 1, R);
    swap(s[i], s[k]);
} }

set<string> permutations(string& s) {
  set<string> R;
  permutations(s, 0, R);
  return R;
}

int main() { string s; cin >> s;
  set<string> R = permutations(s);
  for (string t : R) cout << t << endl;
}
```

```
#include <set>
#include <string>
#include <iostream>
using namespace std;

void permutations(string& s, int k, set<string>& R) {
  if (k == int(s.length()) - 1) R.insert(s);
  else for (int i = k; i < int(s.length()); ++i) {
    swap(s[k], s[i]);
    permutations(s, k + 1, R);
    swap(s[i], s[k]);
} }

void permutations(string& s, set<string>& R) {
  permutations(s, 0, R);
}

int main() { string s; cin >> s;
  set<string> R;
  permutations(s, R);
  for (string t : R) cout << t << endl;
}
```