Algorithm	Function	Worst case time complexity	Best case time complexity	Average case time complexity	Space complexity

The simplest primality by trial division: Given an input number n, check whether any prime integer m from 2 to √n evenly divides n (the division leaves no remainder). If n is divisible by any m then n is composite, otherwise it is prime.
(Source: Wikipedia)
	def trial_division(n):
	"""Return a list of the prime factors for a natural number."""
	a = [] #Prepare an empty list.
	f = 2 #The first possible factor.
	while n > 1: #While n still has remaining factors...
	 if (n % f == 0): #The remainder of n divided by f might be zero.
	  a.append(f) #If so, it divides n. Add f to the list.
	  n /= f #Divide that factor out of n.
	 else: #But if f is not a factor of n,
	  f += 1 #Add one to f and try again.
	return a #Prime factors may be repeated: 12 factors to 2,2,3.
		O(2^(n/2))	O(2^(n/2)/(pi/2*ln2))	O(2^(n/2))	O(n)

Binary Search
(Source: Wikipedia)
	function binary_search(A, n, T):
	  L := 0
	  R := n − 1
	  while L <= R:
	    m := floor((L + R) / 2)
	    if A[m] < T:
	      L := m + 1
	    else if A[m] > T:
	      R := m - 1
	    else:
	      return m
	  return unsuccessful
		O(log n)	O(1)	O(log n)	O(1)

Finding the smallest or largest item in an unsorted array
(Source: Wikipedia)
	function select(list[1..n], k)
	  for i from 1 to k
	    minIndex = i
	    minValue = list[i]
	    for j from i+1 to n
	      if list[j] < minValue
	        minIndex = j
	        minValue = list[j]
	        swap list[i] and list[minIndex]
	  return list[k]
	Partial selection sort
		O(n^2)	O(1)	O(n log n)	O(1)

Kadane's algorithm (Source: https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/)	Initialize: max_so_far = 0 max_ending_here = 0 Loop for each element of the array (a) max_ending_here = max_ending_here + a[i] (b) if(max_ending_here < 0) max_ending_here = 0 (c) if(max_so_far < max_ending_here) max_so_far = max_ending_here return max_so_far	O(n)	O(n)	O(n)	O(1)-O(n)
Sieve of Eratosthenes (Source: Wikipedia)	Input: an integer n > 1. Let A be an array of Boolean values, indexed by integers 2 to n, initially all set to true. for i = 2, 3, 4, ..., not exceeding √n: if A[i] is true: for j = i2, i2+i, i2+2i, i2+3i, ..., not exceeding n: A[j] := false. Output: all i such that A[i] is true.	O(n (log n) (log log n))	O(√nlog log n/log n)	O(n log log n)	O(n)

<table>
<tr>
<td>Merge Sort
(Source: Wikipedia)</td>
<td><pre>
// Array A[] has the items to sort; array B[] is a work array.
TopDownMergeSort(A[], B[], n)
{
    CopyArray(A, 0, n, B);   // duplicate array A[] into B[]
    TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

// Sort the given run of array A[] using array B[] as a source.
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if(iEnd - iBegin < 2)   // if run size == 1
        return;   //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;   // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin, iMiddle, B); // sort the left run
    TopDownSplitMerge(A, iMiddle,   iEnd, B); // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

// Left source half is A[iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is   B[iBegin:iEnd-1   ].
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

CopyArray(A[], iBegin, iEnd, B[])
{
    for(k = iBegin; k < iEnd; k++)
        B[k] = A[k];
}
</pre></td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n)</td>
</tr>
</table>

Heap Sort (Source: Wikipedia)	procedure heapsort(a, count) is input: an unordered array a of length count (Build the heap in array a so that largest value is at the root) heapify(a, count) (The following loop maintains the invariants that a[0:end] is a heap and every element beyond end is greater than everything before it (so a[end:count] is in sorted order)) end ← count - 1 while end > 0 do (a[0] is the root and largest value. The swap moves it in front of the sorted elements.) swap(a[end], a[0]) (the heap size is reduced by one) end ← end - 1 (the swap ruined the heap property, so restore it) siftDown(a, 0, end)	O(n log n)	O(n log n)	O(n log n)	O(1)

Quick Sort
(Source: Wikipedia)
	algorithm quicksort(A, lo, hi) is
	if lo < hi then
	p := partition(A, lo, hi)
	quicksort(A, lo, p - 1)
	quicksort(A, p + 1, hi)
	
	algorithm partition(A, lo, hi) is
	pivot := A[hi]
	i := lo
	for j := lo to hi - 1 do
	if A[j] < pivot then
	swap A[i] with A[j]
	i := i + 1
	swap A[i] with A[hi]
	return i
		O(n^2)	O(n log n)	O(n log n)	O(n)

Tim Sort (Source: Wikipedia)	void timSort(int arr[], int n)
	{
	    // Sort individual subarrays of size RUN
	    for (int i = 0; i < n; i+=RUN)
	        insertionSort(arr, i, min((i+31), (n-1)));
	
	    // start merging from size RUN (or 32). It will merge
	    // to form size 64, then 128, 256 and so on ....
	    for (int size = RUN; size < n; size = 2*size)
	    {
	        // pick starting point of left sub array. We
	        // are going to merge arr[left..left+size-1]
	        // and arr[left+size, left+2*size-1]
	        // After every merge, we increase left by 2*size
	        for (int left = 0; left < n; left += 2*size)
	        {
	            // find ending point of left sub array
	            // mid+1 is starting point of right sub array
	            int mid = left + size - 1;
	            int right = min((left + 2*size - 1), (n-1));
	
	            // merge sub array arr[left.....mid] &
	            // arr[mid+1....right]
	            merge(arr, left, mid, right);
	        }
	    }
	}
		O(n log n)	O(n)	O(n log n)	O(n)

Divide and conquer (Convex Hull)
(Source: https://www.geeksforgeeks.org/convex-hull-using-divide-and-conquer-algorithm/)
	vector<pair<int, int>> divide(vector<pair<int, int>> a)
	{
	// If the number of points is less than 6 then the
	// function uses the brute algorithm to find the
	// convex hull
	if (a.size() <= 5)
	return bruteHull(a);
	
	// left contains the left half points
	// right contains the right half points
	vector<pair<int, int>>left, right;
	for (int i=0; i<a.size()/2; i++)
	left.push_back(a[i]);
	for (int i=a.size()/2; i<a.size(); i++)
	right.push_back(a[i]);
	
	// convex hull for the left and right sets
	vector<pair<int, int>>left_hull = divide(left);
	vector<pair<int, int>>right_hull = divide(right);
	
	// merging the convex hulls
	return merger(left_hull, right_hull);
	} return ret;
	}
		O(n log n)	O(n log n)	O(n log n)	O(n)

Insertion Sort (Source: Wikipedia)	i ← 1 while i < length(A) j ← i while j > 0 and A[j-1] > A[j] swap A[j] and A[j-1] j ← j - 1 end while i ← i + 1 end while	O(n^2)	O(n)	O(n^2)	O(n)

Dijkstra's algorithm
(Source: Wikipedia)
	function Dijkstra(Graph, source):
	
	create vertex set Q
	
	for each vertex v in Graph: // Initialization
	dist[v] ← INFINITY // Unknown distance from source to v
	prev[v] ← UNDEFINED // Previous node in optimal path from source
	add v to Q // All nodes initially in Q (unvisited nodes)
	
	dist[source] ← 0 // Distance from source to source
	
	while Q is not empty:
	u ← vertex in Q with min dist[u] // Node with the least distance
	// will be selected first
	remove u from Q
	
	for each neighbor v of u: // where v is still in Q.
	alt ← dist[u] + length(u, v)
	if alt < dist[v]: // A shorter path to v has been found
	dist[v] ← alt
	prev[v] ← u
	
	return dist[], prev[]
		O(|E|+|V|^2)	O(|E|+|V|*log |V|)	O(|E|+|V|log(|E|/|V|)log |V|)	When arc weights are small integers (bounded by a parameter C), O(|E|loglogC)

Naive Matrix Multiplication (Source: Wikipedia)	$A = (a_{ij})_{1\le i\le m, 1\le j\le n} \quad B = (b_{jk})_{1\le j\le n, 1\le k\le p} \quad C = (c_{ik})_{1\le i\le m, 1\le k\le p}$ $c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$	O(n^3)	O(n^3)	O(n^3)	O(n^2)

Floyd–Warshall algorithm
(Source: Wikipedia)
	let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
	for each edge (u,v)
	dist[u][v] ← w(u,v) // the weight of the edge (u,v)
	for each vertex v
	dist[v][v] ← 0
	for k from 1 to |V|
	for i from 1 to |V|
	for j from 1 to |V|
	if dist[i][j] > dist[i][k] + dist[k][j]
	dist[i][j] ← dist[i][k] + dist[k][j]
	end if
		O(|V|^3)	O(|V|^3)	O(|V|^3)	O(|V|^2)

Naive Matrix Inversion
(Source: Wikipedia)
	h := 1 /* Initialization of the pivot row */
	k := 1 /* Initialization of the pivot column */
	while h ≤ m and k ≤ n
	  /* Find the k-th pivot: */
	  i_max := argmax (i = h ... m, abs(A[i, k]))
	  if A[i_max, k] = 0
	    /* No pivot in this column, pass to next column */
	    k := k+1
	  else
	     swap rows(h, i_max)
	     /* Do for all rows below pivot: */
	     for i = h + 1 ... m:
	        f := A[i, k] / A[h, k]
	        /* Fill with zeros the lower part of pivot column: */
	        A[i, k] := 0
	        /* Do for all remaining elements in current row: */
	        for j = k + 1 ... n:
	           A[i, j] := A[i, j] - A[h, j] * f
	     /* Increase pivot row and column */
	     h := h+1
	     k := k+1
		O(n^3)	O(n^3)	O(n^3)	O(n^2)

Calculate the permutations of n distinct elements without repetitions
(Source: https://www.geeksforgeeks.org/distinct-permutations-string-set-2/)
	#include <bits/stdc++.h>
	using namespace std;
	
	// Returns true if str[curr] does not matches with any of the
	// characters after str[start]
	bool shouldSwap(char str[], int start, int curr)
	{
	    for (int i = start; i < curr; i++)
	        if (str[i] == str[curr])
	            return 0;
	    return 1;
	}
	
	// Prints all distinct permutations in str[0..n-1]
	void findPermutations(char str[], int index, int n)
	{
	    if (index >= n) {
	        cout << str << endl;
	        return;
	    }
	
	    for (int i = index; i < n; i++) {
	
	        // Proceed further for str[i] only if it
	        // doesn't match with any of the characters
	        // after str[index]
	        bool check = shouldSwap(str, index, i);
	        if (check) {
		O(1)	O(n-k)
			At most (last - first) comparisons and (last - first) swaps.
			13 [Note:In order to prepare the range [first,last) for an enumeration of all partial permutations in lexicographic order, std::sort(first,last) or
				O(n-k)	O(1)

	if (check) {
	swap(str[index], str[i]);
	findPermutations(str, index + 1, n);
	swap(str[index], str[i]);
	}
	}
	}
	
	// Driver code
	int main()
	{
	char str[] = "ABCA";
	int n = strlen(str);
	findPermutations(str, 0, n);
	return 0;
	}
			std::sort(first,last,comp).
			— end Note]		

	replicateM(3, {1, 2})) ->
	-- {{1, 1, 1}, {1, 1, 2}, {1, 2, 1}, {1, 2, 2}, {2, 1, 1},
	-- {2, 1, 2}, {2, 2, 1}, {2, 2, 2}}
	
	-- replicateM :: Int -> [a] -> [[a]]
	on replicateM(n, xs)
	script go
	script cons
	on |λ|(a, bs)
	{a} & bs
	end |λ|
	end script
	on |λ|(x)
	if x ≤ 0 then
	{{}}
	else
	liftA2List(cons, xs, |λ|(x - 1))
	end if
	end |λ|
	end script
	
	go's |λ|(n)
	end replicateM
	
	-- TEST --
	on run
	
	replicateM(2, {1, 2, 3})				

<table>
<tr><th></th><th></th><th></th><th></th><th></th><th></th></tr>
<tr>
<td>Calculate the permutations of n distinct elements with repetitions (Source: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.353.930&rep=rep1&type=pdf)</td>
<td>   replicateM(2, {1, 2, 3})

    -- {{1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2}, {2, 3}, {3, 1}, {3, 2}, {3, 3}}
end run

-- GENERIC FUNCTIONS ---

-- concatMap :: (a -> [b]) -> [a] -> [b]
on concatMap(f, xs)
    set lng to length of xs
    set acc to {}
    tell mReturn(f)
        repeat with i from 1 to lng
            set acc to acc & |λ|(item i of xs, i, xs)
        end repeat
    end tell
    return acc
end concatMap

-- liftA2List :: (a -> b -> c) -> [a] -> [b] -> [c]
on liftA2List(f, xs, ys)
    script
        property g : mReturn(f)'s |λ|
        on |λ|(x)
            script
                on |λ|(y)
                    {g(x, y)}
                end |λ|</td>
<td>O(1)</td>
<td>O(n-k)
At most (last - first) decrements of BidirectionalIterator and (last - first) increments of T.</td>
<td>O(n-k)</td>
<td>O(1)</td>
</tr>
</table>

	<pre> end script
	        concatMap(result, ys)
	    end |λ|
	  end script
	  concatMap(result, xs)
	end liftA2List
	
	-- Lift 2nd class handler function into 1st class script wrapper
	-- mReturn :: First-class m => (a -> b) -> m (a -> b)
	on mReturn(f)
	  if class of f is script then
	    f
	  else
	    script
	      property |λ| : f
	    end script
	  end if
	end mReturn</pre>				