Average time constraints of organization methods

basic methods: O(N2)

Advanced methods: O(NlogN)

special cases: O(N) if the key has special features that can be utilized at the bit level.

In addition, for all the methods of organization based on comparisons of full keys, it is true that their worst case time is Ω(NlogN). That is, this is the lower limit of their performance, regardless of the algorithm. Thus, methods capable of linear time complexity need access to key bit information, i.e. they do not compare whole keys (but key parts) with each other. These will be examples later.

All \*advanced\* comparison-based sorting algorithms have Θ(NlogN)Θ(NlogN) running time in the average case.

A sorting algorithm is said to be stable if... it preserves the relative ordering of records with identical key values.

We might use sorting to help an algorithm to solve some other problem. For example, [**Kruskal's algorithm**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-kruskal-s-algorithm) to find a [**minimal-cost spanning tree**](https://traky.cs.hut.fi/Books/CSE-A1141/html/MCST.html#mcst) must sort the edges of a graph by their lengths before it can process them.

For example, a natural way to sort your cards in a bridge hand is to go from left to right, and place each card in turn in its correct position relative to the other cards that you have already sorted. This is the idea behind [**Insertion Sort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/InsertionSort.html#insertionsort). For example, no normal person would use [**Quicksort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Quicksort.html#quicksort) to order a pile of bills by date, even though Quicksort is the standard sorting algorithm of choice for most software libraries.

[**Mergesort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Mergesort.html#mergesort) divides a list in half. [**Quicksort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Quicksort.html#quicksort) divides a list into big values and small values. [**Radix Sort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/RadixSort.html#radixsort) divides the problem by working on one digit of the key at a time.

Quicksort illustrates that it is possible for an algorithm to have an [**average case**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-average-case) whose growth rate is significantly smaller than its [**worst case**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-worst-case). It is possible to speed up one sorting algorithm (such as [**Shellsort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Shellsort.html#shellsort) or Quicksort) by taking advantage of the [**best case**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-best-case) behavior of another algorithm (Insertion Sort).  Special case behavior by some sorting algorithms makes them a good solution for special niche applications ([**Heapsort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Heapsort.html#heapsort)). Sorting provides an example of an important technique for analyzing the lower bound for a problem. [**External Sorting**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-external-sort) refers to the process of sorting large files stored on disk.

three simple, but relatively slow, algorithms that require Θ(n2)Θ(n2) time in the average and worst cases to sort nn records. Several algorithms with considerably better performance are then presented, some with Θ(nlogn)Θ(nlog⁡n) worst-case running time. The final sorting method presented requires only Θ(n)Θ(n) worst-case time under special conditions (but it cannot run that fast in the general case). The chapter concludes with a proof that sorting in general requires Ω(nlogn)Ω(nlog⁡n) time in the worst case.

 the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

# Insertion sort

**def** inssort(A):

**for** i **in** range(len(A)): *# Insert i'th record*

j = i;

**while** (j != 0) **and** (A[j] < A[j-1]):

swap(A, j, j - 1)

j -= 1

**void** inssort(Comparable\* A[], **int** n) { *// Insertion Sort*

**for** (**int** i = 1; i < n; i++) *// Insert i'th record*

**for** (**int** j = i; (j > 0) && (\*A[j] < \*A[j-1]); j--)

swap(A, j, j-1);

}

As soon as a key value less than or equal to xx is encountered, inssort is done with that record because all records to its left in the array must have smaller keys.

Worst case: n(n-1)/2 theta(n2)

Best case: n-1 theta(n)

Average: (n-1)(n+4)/4 theta(n2)

While the best case is significantly faster than the average and worst cases, the average and worst cases are usually more reliable indicators of the "typical" running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight.  And even when the input is not perfectly sorted, Insertion Sort's cost goes up in proportion to the number of inversions. So a "nearly sorted" list will always be cheap to sort with Insertion Sort. Examples of algorithms that take advantage of Insertion Sort's near-best-case running time are [**Shellsort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Shellsort.html#shellsort) and [**Quicksort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Quicksort.html#quicksort).

Thus, the number of swaps for the entire sort operation is n−1n−1 less than the number of comparisons. This is 0 in the best case, and Θ(n2)Θ(n2) in the average and worst cases.

 for larger arrays, Insertion Sort will not be so good a performer as other algorithms. So Insertion Sort is not the best sorting algorithm to use in most situations. But there are special situations where it is ideal. We already know that Insertion Sort works great when the input is sorted or nearly so. Another good time to use Insertion Sort is when the array is very small, since Insertion Sort is so simple.

The algorithms that have better asymptotic growth rates tend to be more complicated, which leads to larger constant factors in their running time. That means they typically need fewer comparisons for larger arrays, but they cost more per comparison.  even an algorithm with high cost per comparison will be fast on small input sizes.

# Selection Sort

The ii'th pass of Selection Sort "selects" the ii'th largest key in the array, placing that record at the end of the array. In other words, Selection Sort first finds the largest key in an unsorted list, then the next largest, and so on. Its unique feature is that there are few record swaps. To find the next-largest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record into place. Thus, the total number of swaps required will be n−1

**def** selsort(A):

**for** i **in** range(len(A)): *# Select i'th biggest record*

bigindex = 0; *# Current biggest index*

**for** j **in** range (1, len(A) - i): *# Find the max value*

**if** (A[j] > A[bigindex]): *# Found something bigger*

bigindex = j; *# Remember bigger index*

swap(A, bigindex, len(A) - i - 1); *# Put it into place*

**void** selectionsort(Comparable\* A[], **int** n) {

**for** (**int** i = 0; i < n-1; i++) { *// Select i'th biggest record*

**int** bigindex = 0; *// Current biggest index*

**for** (**int** j = 1; j < n-i; j++) *// Find the max value*

**if** (\*A[j] > \*A[bigindex]) *// Found something bigger*

bigindex = j; *// Remember bigger index*

swap(A, bigindex, n-i-1); *// Put it into place*

}

}

 we could have written Selection Sort to find the smallest record, the next smallest, and so on. We wrote this version of Selection Sort to mimic the behavior of our Bubble Sort implementation as closely as possible.  This shows that Selection Sort is essentially a Bubble Sort except that rather than repeatedly swapping adjacent values to get the next-largest record into place, we instead remember the position of the record to be selected and do one swap at the end.

Comparisons: n(n-1)/2 theta(n2)

Swaps: theta(n)

There is another approach to keeping the cost of swapping records low, and it can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values. The large records do not need to move. This technique is illustrated by Figure [**4.5.1**](https://traky.cs.hut.fi/Books/CSE-A1141/html/SelectionSort.html#pointerswap). Additional space is needed to store the pointers, but the return is a faster swap operation.

# Mergesort

 To use divide and conquer when sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind [**Mergesort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-mergesort).

Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Unfortunately, even though it is based on a simple concept, it is relatively difficult to implement in practice. Here is a pseudocode sketch of Mergesort:

List mergesort(List inlist) {

**if** (inlist.length() <= 1) **return** inlist;;

List L1 = half of the items from inlist;

List L2 = other half of the items from inlist;

**return** merge(mergesort(L1), mergesort(L2));

}

The hardest step to understand about Mergesort is the merge function. The merge function starts by examining the first record of each sublist and picks the smaller value as the smallest record overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front records of the sublists and continually appending the smaller to the output list until no more input records remain.

List merge(List L1, List L2) {

List answer = new List();

**while** (L1 != NULL || L2 != NULL) {

**if** (L1 == NULL) { *// Done L1*

answer.append(L2);

L2 = NULL;

}

**else** **if** (L2 == NULL) { *// Done L2*

answer.append(L1);

L1 = NULL;

}

**else** **if** (L1.value() <= L2.value()) {

answer.append(L1.value());

L1 = L1.next();

}

**else** {

answer.append(L2.value());

L2 = L2.next();

}

}

**return** answer;

}

Mergesort lends itself well to sorting a singly linked list because merging does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list. Implementing merge for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list.

i = **0**; j = **0**; /\* i,j: askeltajat; lomitettavat taulukot \*/

a[M] = INT\_MAX; b[N] = INT\_MAX;

**for** (k = **0**; k < M + N; k++) /\* k: askeltaja; kohdetaulukko \*/

**if** ( a[i] < b[j] ) {

c[k] = a[i];

i = i + **1**;

} **else** {

c[k] = b[j];

j = j + **1**;

}

 A simpler method, which does not rely on knowing the length of the list in advance, assigns elements of the input list alternating between the two sublists. The first element is assigned to the first sublist, the second element to the second sublist, the third to first sublist, the fourth to the second sublist, and so on. This requires one complete pass through the input list to build the sublists.

When the input to Mergesort is an array, splitting input into two subarrays is easy if we know the array bounds. Merging is also easy if we merge the subarrays into a second array. Note that this approach requires twice the amount of space as any of the sorting methods presented so far, which is a serious disadvantage for Mergesort.

Mergesort is recursively called until subarrays of size 1 have been created, requiring logn levels of recursion.

 A much simpler approach is to copy the sorted sublists to the auxiliary array first, and then merge them back to the original array.

An optimized Mergesort implementation is shown below. It reverses the order of the second subarray during the initial copy. Now the current positions of the two subarrays work inwards from the ends, allowing the end of each subarray to act as a sentinel for the other. Unlike the previous implementation, no test is needed to check for when one of the two subarrays becomes empty. This version also has a second optimization: It uses Insertion Sort to sort small subarrays whenever the size of the array is smaller than a value defined by THRESHOLD.

**void** mergesortOpt(Comparable\* A[], Comparable\* temp[], **int** left, **int** right) {

**int** i, j, k, mid = (left+right)/2;*// Select the midpoint*

**if** (left == right) **return**; *// List has one record*

**if** ((mid-left) >= THRESHOLD) mergesortOpt(A, temp, left, mid);

**else** inssort(A, left, mid);

**if** ((right-mid) > THRESHOLD) mergesortOpt(A, temp, mid+1, right);

**else** inssort(A, mid+1, right);

*// Do the merge operation. First, copy 2 halves to temp.*

**for** (i=left; i<=mid; i++) \*temp[i] = \*A[i];

**for** (j=right; j>mid; j--) \*temp[i++] = \*A[j];

*// Merge sublists back to array*

**for** (i=left,j=right,k=left; k<=right; k++)

**if** (\*temp[i] <= \*temp[j]) \*A[k] = \*temp[i++];

**else** \*A[k] = \*temp[j--];

}

# Quicksort

[**Quicksort**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-quicksort) is aptly named because, when properly implemented, it is the fastest known general-purpose in-memory sorting algorithm in the average case. It does not require the extra array needed by Mergesort, so it is space efficient as well. Quicksort is widely used, and is typically the algorithm implemented in a library sort routine such as the UNIX qsort function.

the root of the BST (i.e., the first node inserted) splits the list into two sublists: The left subtree contains those values in the list less than the root value while the right subtree contains those values in the list greater than or equal to the root value. Thus, the BST implicitly implements a "divide and conquer" approach to sorting the left and right subtrees. Quicksort implements this same concept in a much more efficient way.

The records are then rearranged in such a way that the kk values less than the pivot are placed in the first, or leftmost, kk positions in the array, and the values greater than or equal to the pivot are placed in the last, or rightmost, n−kn−k positions. This is called a [**partition**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-partition) of the array.

**void** quicksort(Comparable\* A[], **int** i, **int** j) {

**int** pivotindex = findpivot(i, j);

swap(A, pivotindex, j); *// Stick pivot at end*

*// k will be the first position in the right subarray*

**int** k = partition(A, i, j-1,A[j]);

swap(A, k, j); *// Put pivot in place*

**if** ((k-i) > 1) quicksort(A, i, k-1); *// Sort left partition*

**if** ((j-k) > 1) quicksort(A, k+1, j); *// Sort right partition*

}

Function partition will move records to the appropriate partition and then return k, the first position in the right partition. Note that the pivot value is initially placed at the end of the array (position j). Thus, partition must not affect the value of array position j. After partitioning, the pivot value is placed in position k, which is its correct position in the final, sorted array. By doing so, we guarantee that at least one value (the pivot) will not be processed in the recursive calls to qsort. Even if a bad pivot is selected, yielding a completely empty partition to one side of the pivot, the larger partition will contain at most n−1 records.

Selecting a pivot can be done in many ways. The simplest is to use the first key. However, if the input is sorted or reverse sorted, this will produce a poor partitioning with all values to one side of the pivot. It is better to pick a value at random, thereby reducing the chance of a bad input order affecting the sort. Unfortunately, using a random number generator is relatively expensive, and we can do nearly as well by selecting the middle position in the array. Here is a simple findpivot function.

**int** findpivot(**int** i, **int** j)

{ **return** (i+j)/2; }

We now turn to function partition. If we knew in advance how many keys are less than the pivot, partition could simply copy records with key values less than the pivot to the low end of the array, and records with larger keys to the high end. Because we do not know in advance how many keys are less than the pivot, we use a clever algorithm that moves indices inwards from the ends of the subarray, swapping values as necessary until the two indices meet. Here is an implementation for the partition step.

**int** partition(Comparable\* A[], **int** left, **int** right, Comparable\* pivot) {

**while** (left <= right) { *// Move bounds inward until they meet*

**while** (\*A[left] < \*pivot) left++;

**while** ((right >= left) && (\*A[right] >= \*pivot)) right--;

**if** (right > left) swap(A, left, right); *// Swap out-of-place values*

}

**return** left; *// Return first position in right partition*

}

Function partition returns the first index of the right partition (the place where left ends at) so that the subarray bound for the recursive calls to qsort can be determined.

Quicksort's average-case behavior falls somewhere between the extremes of worst and best case. Average-case analysis considers the cost for all possible arrangements of input, summing the costs and dividing by the number of cases.

Worst case: n2

Best case: nlogn

Average: T(n)= cn+ sum[(T(k)+T(n-1-k))

# Radix Sort[¶](https://traky.cs.hut.fi/Books/CSE-A1141/html/RadixSort.html#radix-sort)

Consider a sequence of records with keys in the range 0 to 99. If we have ten bins available, we can first assign records to bins by taking their key value modulo 10. Thus, every key will be assigned to the bin matching its rightmost decimal digit. We can then take these records from the bins **in order**, and reassign them to the bins on the basis of their leftmost (10's place) digit. We will define values in the range 0 to 9 to have a leftmost digit of 0. In other words, assign the ii'th record from array A to a bin using the formula A[i]/10. If we now gather the values from the bins **in order**, the result is a sorted list. In this example, we have r=10r=10 bins and key values in the range 0 to r2−1r2−1. The total computation is Θ(n)Θ(n), because we look at each record and each bin a constant number of times.

**void** radix\_exchange(**int** left; **int** right; /\* left, right: kulkijat; alueen reunat \*/

**int** pos) { /\* askeltaja; tarkasteltava bittipositio \*/

**if** ((right > left) && (pos >= **0**)) {

i = left; /\* i: askeltaja; vasemmalta oikealle \*/

j = right; /\* j: askeltaja; oikealta vasemmalle \*/

**do** {

**while** (((bits(a[i], pos, **1**) = **0**) && (i < j)))

i++;

**while** (((bits(a[j], pos, **1**) = **1**) && (i < j)))

j--;

swap(a[i], a[j]);

} **while** (j != i);

**if** (bits(a[right], b, **1**) = **0**)

j++;

radix\_exchange(left, j-**1**, pos-**1**);

radix\_exchange(j, right, pos-**1**);

}

}

**static** **void** radixsort(**int** A[], **int** k, **int** r, **int** n) {

**int** B[n];

**int** count[r];

**int** i, j, rtok;

**for** (i = 0, rtok = 1; i < k; i++, rtok \*= r) { *// For k digits*

**for** (j = 0; j < r; j++) count[j] = 0; *// Initialize count*

*// Count the number of records for each bin on this pass*

**for** (j = 0; j < n; j++) count[(A[j]/rtok)%r]++;

*// count[j] will be index in B for last slot of bin j.*

*// First, reduce count[0] because indexing starts at 0, not 1*

count[0] = count[0] - 1;

**for** (j = 1; j < r; j++) count[j] = count[j-1] + count[j];

*// Put records into bins, working from bottom of bin*

*// Since bins fill from bottom, j counts downwards*

**for** (j = n-1; j >= 0; j--) {

B[count[(A[j]/rtok)%r]] = A[j];

count[(A[j]/rtok)%r] = count[(A[j]/rtok)%r] - 1;

}

**for** (j = 0; j < n; j++) A[j] = B[j]; *// Copy B back*

}

}

The first inner for loop initializes array cnt. The second loop counts the number of records to be assigned to each bin. The third loop sets the values in cnt to their proper indices within array B. Note that the index stored in cnt[j] is the *last* index for bin j; bins are filled from high index to low index. The fourth loop assigns the records to the bins (within array B). The final loop simply copies the records back to array A to be ready for the next pass. Variable rtoi stores riri for use in bin computation on the ii'th iteration. The following visualization illustrates the process.