

algoritmus	popis	sløžkost min Q	sløžkost max O	sløžkost průměr θ	stabilita (zachovává pořadí stejných prvků)	datové dtěví, pñizený (rychlejší) zpracuje částečně seřazenou posloupnost)	in/out of place	implementace	pñíklad
Bubble sort (bublinkové řazení)	<ul style="list-style-type: none"> - iterativní algoritmus - porovná první dvě hodnoty - pokud je b > a, pak swap - projde celé pole - po každém průchodu umístíme max prvek - poslední prvek není nutno řadit (je triviálně seřazen) - po n-1 průchodech je pole seřazeno 	$O(n)$	$O(n^2)$	$\theta(n^2)$	ano	ano	in place	<pre>//řazení od nejvyššího function bubbleSort(int[] a) { for (i = 0; i < a.length-1; i++) { for (j = 0; j < a.length-i-1; j++) { if (a[j] < a[j+1]) { help = a[j]; a[j] = a[j+1]; a[j+1] = help; } } } }</pre>	<p>(3 2 8 7 6) // zadání pole, řadíme od největšího k nejmenšímu (3 2 8 7 6) // 3 a 2 jsou v korektním pořadí, posuneme se o index (3 2 8 7 6) // 8 > 2, prohodíme je (3 8 2 7 6) // 7 > 2, prohodíme je (zde je vidět probubávání největší dvojky vzhůru) (3 8 7 2 6) // 6 > 2, prohodíme je (3 8 7 6 2) // nový průchod polem: na posledním místě je nejlehčí prvek, tudíž se nám řazení úloha o jedna zkrátí, 8 > 3, prohodíme je (8 3 7 6 2) // 7 > 3, prohodíme je (8 7 3 6 2) // 6 > 3, prohodíme je (8 7 6 3 2) // seřazeno</p>
Shaker sort (obousměrný bubble sort)	<ul style="list-style-type: none"> - optimalizace Bubble sortu - dtto Bubble sort, ale řadí oběma směry - kratší čas než Bubble sort 	$O(n)$	$O(n^2)$	$\theta(n^2)$	ano	ano	in place	<pre>public static void shakerSort(int[] a) { for (int i = 0; i < a.length/2; i++) { boolean swapped = false; for (int j = i; j < a.length-i-1; j++) { if (array[j] < a[j+1]) { int help = a[j]; a[j] = a[j+1]; a[j+1] = help; swapped = true; } } for (int j = a.length-2-i; j > i; j--) { if (a[j] > a[j-1]) { int help = a[j]; a[j] = a[j-1]; a[j-1] = help; swapped = true; } } if (!swapped) break; } }</pre>	
Insertion sort (řazení vkládáním)	<ul style="list-style-type: none"> - iterativní algoritmus - lze řadit z L nebo z P - najde menší/větší prvek než předchozí - umístí ho - přesune, na správné místo posloupnosti, tzn. prvky posune, není to swap - v každém kroku je seřazeno o jeden prvek víc než v předchozím - u téměř seřazených polí dochází pouze k průchodu = n náročnost - často bývá doplňkem u merge nebo quick sort k seřazení malých polí - dokud pole obsahuje neseřazené prvky, tak GOTO: 2 - použití - tam, kde není na vstupu celá posloupnost; data přicházejí postupně 	$O(n)$	$O(n^2)$	$\theta(n^2)$	ano	ano	in place	<pre>public static void insertionSort(int[] a) { for (int i = 0; i < a.length - 1; i++) { int j = i + 1; int help = a[j]; while (j > 0 && help > a[j-1]) { a[j] = a[j-1]; j--; } a[j] = help; } }</pre>	<p>(3 2 8 7 6) // Zadání, prvek 3 je triviálně seřazen (3 2 8 7 6) // Vezmeme dvojku a vložíme jí na správné místo (tam už je) (3 2 8 7 6) // 8 vložíme na první místo, zbytek čísel posuneme (8 3 2 7 6) // 7 vložíme mezi 8 a 3, 3 a 2 posuneme (8 7 3 2 6) // 6 vložíme mezi 7 a 3, čísla 3 a 2 posuneme (8 7 6 3 2) // seřazeno</p>
Shell sort (řazení se snižujícím se přírůstkem)	<ul style="list-style-type: none"> - optimalizace Insertion sortu - je rychlejší - vezme 1 a n-tý prvek a zařadí, pak 2 a n-1 a zařadí, pak 3 a n-2... - nejvýkonnější kvadratický algoritmus 	$O(n^2)$	$O(n^2)$	$\theta(n^2)$	ne	ano	in place	<pre>public static int[] shellSort(int[] a) { int delka = a.length / 2; while (delka > 0) { //dokud máme co porovnávat for (int i = 0; i < a.length- delka; i++) { //upravený insertion sort int j = i + delka; int help = a[j]; while (j >= delka && help > a[j - delka]) { a[j] = a[j - delka]; j -= delka; } a[j] = help; } if (delka == 2) { //změna velikosti mezery delka = 1; } else { delka /= 2.2; } } return a; }</pre>	
Selection sort (řazení výběrem)	<ul style="list-style-type: none"> - rychlejší než bubble, ale pomalejší než insertion sort - lze řadit z L nebo z P - najde min/max prvek a prohodí ho s a[0]/a[n] 	$O(n^2)$	$O(n^2)$	$\theta(n^2)$	ano	ne	out of place	<pre>public static void selectionSort(int[] a) { for (int i = 0; i < a.length-1; i++) { int maxIndex = i; for (int j = i + 1; j < a.length; j++) { if (a[j] > a[maxIndex]) { maxIndex = j; } } int help = a[i]; a[i] = a[maxIndex]; a[maxIndex] = help; } }</pre>	<p>(3 2 8 7 6) // zadání pole, řadíme od největšího k nejmenšímu (3 2 8 7 6) // největší číslo je 8, prohodíme ho tedy s číslem 3 na indexu 0 (8 2 3 7 6) // největší číslo je 7, prohodíme ho tedy s číslem 2 na indexu 1 (8 7 3 2 6) // největší číslo je 6, prohodíme ho tedy s číslem 3 na indexu 2 (8 7 6 2 3) // největší číslo je 3, prohodíme ho tedy s číslem 2 na indexu 3 (8 7 6 3 2) // seřazeno</p>
Merge sort (řazení sléváním)	<ul style="list-style-type: none"> - typ rozděl a panuj (rekurze) - půlení pole až jsou jen dvojice, ty seřadí - slévání již seřazených částí pole - hloubka je log n, protože vždy pole půlíme - mergujeme n prvků - obecně alg. rozděl a panuj - rozdělí P na menší podproblémy - rekurzi volají samy sebe na menší podproblémy - sestaví výsledné řešení 	$O(n \log n)$	$O(n \log n)$	$\theta(n \log n)$	ano, pokud se to seřadí	ano	out of place	<pre>//řazení od nejvyššího a2 = pomocné pole stejné delky jako array left = první index na který se smí sahout right = poslední index, na který se smí sahout public static void mergeSort(int[] a, int[] a2, int left, int right) { if (left == right) return; int middleIndex = (left + right)/2; mergeSort(a, a2, left, middleIndex); mergeSort(a, a2, middleIndex + 1, right); merge(a, a2, left, right); for (int i = left; i <= right; i++) { a[i] = a2[i]; } } //slévání pro Merge sort private static void merge(int[] a, int[] a2, int left, int right) { int middleIndex = (left + right)/2; int leftIndex = left; int rightIndex = middleIndex + 1; int auxIndex = left; while (leftIndex <= middleIndex && rightIndex <= right) { if (a[leftIndex] <= a[rightIndex]) { a2[auxIndex] = a[leftIndex++]; } else { a2[auxIndex] = a[rightIndex++]; } auxIndex++; } while (leftIndex <= middleIndex) { a2[auxIndex] = a[leftIndex++]; auxIndex++; } while (rightIndex <= right) { a2[auxIndex] = a[rightIndex++]; auxIndex++; } }</pre>	

algoritmus	popis	slóžitost min Q	slóžitost max O	slóžitost průměr Θ	stabilita (zachovává pořadí stejných prvků)	datové dtívy, přehrány (rychleji) zpracuje částečně seřazenou posloupnost)	in/out of place	implementace	příklad
Quick sort	<ul style="list-style-type: none"> - typ rozdělení a panuj (rekurziv) - závisí na volbě pozice pivotu - nejprve hledá z L, když najde prvek větší než pivot, prohodí je a tím se dostává pivot na jiný index - další průchod je z P a když najde prvek menší než pivot, tak je prohodí - postupně se takto projde celé pole, po průchodu jsou na L straně prvky menší než pivot, na P větší - max slóžitost nastává při špatné volbě pivotu 	$O(n \log n)$	$O(n^2)$	$\Theta(n \log n)$	ne	ne	in place	<pre> //radí od nejvyššího prvku //left index prvního prvku, na který můžeme sahout (leva mez (včetně)) //right index prvního prvku, na který nemůžeme sahout (prava mez (bez)) public static void quicksort(int[] a, int left, int right) { if (left < right) { int boundary = left; for (int i = left + 1; i < right; i++) { if (a[i] > a[left]) { swap(a, i, ++boundary); } } swap(a, left, boundary); quicksort(a, left, boundary); quicksort(a, boundary + 1, right); } } //prohodí prvky v sadanem poli //left prvek 1 //right prvek 2 private static void swap(int[] a, int left, int right) { int help = a[right]; a[right] = a[left]; a[left] = help; } </pre>	
Counting sort (ultra sort, math sort)	<ul style="list-style-type: none"> - pracuje na bázi výpočtu výskytu hodnot, tzn. neporovnává hodnoty - známe přesné univerzum - prvků je omezený počet - vytvoří pole četností (= kolikrát je daná hodnota zastoupena v poli) - přepočte na pole "posledních indexů" = index i značí pozici posledního výskytu daného prvku v seřazeném poli - např. při zařazení obcí do kraje nejprve spočtu obce v jednotlivých krajích a pak spočtu indexy pro jednotlivé kraje a seřídím obce do jednoho pole 	$O(n)$	$O(n)$	$\Theta(n)$	ano	ne	out of place	<pre> //return pole seřazené od nejnižší hodnoty po nejvyšší public static int[] countingSort(int[] a) { // pole do kterého budeme radit, v případě primitiv nemá smysl // da se radit i bez něj, ale v případě objektu by to jinak neslo int[] a2 = new int[a.length]; // najdeme maximum a minimum int min = array[0]; int max = array[0]; for (int i = 1; i < a.length; i++) { if (a[i] < min) min = a[i]; else if (a[i] > max) max = a[i]; } // vytvoříme pole do kterého budeme počítat int[] counts = new int[max - min + 1]; // inicializujeme počty výskytu for (int i = 0; i < a.length; i++) { counts[a[i] - min]++; } // přepočítáme výskyt na poslední index dané hodnoty counts[0]--; for (int i = 1; i < counts.length; i++) { counts[i] = counts[i] + counts[i-1]; } // seřad pole sprava doleva // 1) vyhledá poslední výskyt dané hodnoty v poli výskytu // 2) uloží hodnotu na příslušné místo v seřazeném poli // 3) sniž index posledního výskytu dané hodnoty // 4) pokračuj s předchozí hodnotou vstupního pole (goto: 1), // sestupný, pokud již všechny hodnoty byly seřazeny for (int i = a.length - 1; i >= 0; i--) { a2[counts[a[i] - min]--] = a[i]; } return a2; } </pre>	<p>Vstupní pole: 9 6 6 3 2 0 4 2 9 3 Pole četností: 1 0 2 2 1 0 2 0 0 2 Pole výskytů: 0 0 2 4 5 5 7 7 9 Seřazené pole: 0 2 2 3 3 4 6 6 9 9</p> <p>Vstupní pole: 2 8 9 8 0 8 8 9 4 6 Pole četností: 1 0 1 0 1 0 1 0 4 2 Pole výskytů: 0 0 1 1 2 2 3 3 7 9 Seřazené pole: 0 2 4 6 8 8 8 8 9 9</p> <p>Vstupní pole: 9 2 1 9 4 1 5 7 5 3 Pole četností: 2 1 1 1 2 0 1 0 2 Pole výskytů: 1 2 3 4 6 6 7 7 9 Seřazené pole: 1 1 2 3 4 5 5 7 9 9</p>
Radix Sort (příhradkové řazení)	<ul style="list-style-type: none"> - používá se zejména k řazení řetězců stejné délky - pokud řadíme podle A a pak podle B, tak je seřazené podle B, kde jsou struktury seřazené podle klíče A - seřadí podle posledního znaku, pak podle znaku n-1, pak n-2 atd. - musí být pouze celá čísla - mohou být čísla libovolné poziční soustavy, ale musíme vědět jaké - lineární, protože na řazení je použit Counting sort (univerzum 0-9 u desítkové soustavy) - konstantní délka řazení řetězců 	$O(n)$	$O(n)$	$\Theta(n)$	ano	ne	out of place	<pre> //radí od nejnižší hodnoty - vnitřní stabilní řazení: counting sort //dimension = součet det (řádků) public static int[] radixSort(int[] a, int dimension) { for (int i = dimension - 1; i >= 0; i--) { a = countingSortForRadix(a, i); } return a; } //Counting sort pro Radix sort - radí pole dle hodnoty na pozici //position = pozice, podle které se bude radit public static int[] countingSortForRadix(int[] a, int position) { // pole do kterého budeme radit, v případě primitiv nemá smysl // da se radit i bez něj, ale v případě objektu by to jinak neslo int[] a2 = new int[a.length]; . . . } </pre>	
Bucket sort (bin sort, příhradkové třídění, kýblů)	<ul style="list-style-type: none"> - známe předem univerzum, tzn. známe MIN a MAX řazených prvků - vstupní pole rozdělíme rovnoměrně na několik bucketů (kýblů), v každé příhradce je n/d prvků (d = počet kýblů) - na každou příhradku je zavolán vhodný třídící algoritmus pro seřazení (zpravidla Insertion sort) - nakonec jsou všechna data nakopírována do výstupního pole - s lineární slóžitostí lze použít pokud bude rovnoměrně rozložení prvků v "kýblech", tzn. předem známe univerzum a předem můžeme určit intervaly pro přidělení "kýblů" 	$O(n \log n)$	$O(n \log n)$	$\Theta(n \log n)$	ano	ne	out of place	<pre> //bucketCount = počet bucketu //return seřazené pole (od nejnižšího k nejvyššímu) public static int[] bucketSort(int[] a, int bucketCount) { if (bucketCount <= 0) throw new IllegalArgumentException("Neplatný počet bucketů"); if (a.length <= 1) return a; //triviálně seřazené int high = a[0]; int low = a[0]; for (int i = 1; i < a.length; i++) { //najdeme nejvyšší a nejnižší if (a[i] > high) high = a[i]; if (a[i] < low) low = a[i]; } double interval = ((double) (high - low + 1)) / bucketCount; //počet císel každých jedním bucketem = //počet císel celkem / počet bucketu ArrayList<Integer> buckets[] = new ArrayList[bucketCount]; for (int i = 0; i < bucketCount; i++) { //inicializace bucketu buckets[i] = new ArrayList(); } for (int i = 0; i < a.length; i++) { //zahusení císel do bucketu buckets[i] = ((array[i] - low) / interval).add(array[i]); } int pointer = 0; for (int i = 0; i < buckets.length; i++) { Collections.sort(buckets[i]); //mergeSort for (int j = 0; j < buckets[i].size(); j++) { //separatně a[pointer] = buckets[i].get(j); pointer++; } } return array; } </pre>	