Problem sortowania:

1. Dana jest lista parami porównywalnych obiektów (np. liczb, stringów, ...).
2. Chcemy znaleźć permutację listy, dla której kolejne obiekty ułożone są w niemalejącym porządku.

Elementy równe ze względu na porządek[1] nie muszą być tymi samymi obiektami (np. sortowanie listy osób według ich nazwisk).

[1] Ściślej: praporządek

Różnice między algorytmami sortującymi

1. Zastosowanie: ogólne lub do szczególnych zadań.
2. Złożoność czasowa (ze względu na porównania).
3. Złożoność pamięciowa (następny slajd).
4. Stabilność (dalsze slajdy).
5. Inne (paralelizacja, szczególne przypadki, ...).

Złożoność pamięciowa

Złożoność pamięciowa algorytmu (dwa warianty): całkowita lub dodatkowa ilość pamięci użyta do wykonania algorytmu w zależności od rozmiaru wejścia.

Rozważamy jej zachowanie asymptotyczne (podobnie jak dla złożoności czasowej): notacja $\Omega/\Theta/O$, przypadek średni i pesymistyczny, etc.

Stabilność

Algorytm sortujący jest **stabilny**, gdy zachowuje kolejność równych elementów.

Przykład: sortowanie osób względem nazwisk:

```
Lista oryginalna:
  ['Anna Kowalska', 'Marek Nowak', 'Zuzanna Kowalska']
Lista posortowana stabilnie:
  ['Anna Kowalska', 'Zuzanna Kowalska', 'Marek Nowak']
Lista posortowana niestabilnie:
  ['Zuzanna Kowalska', 'Anna Kowalska', 'Marek Nowak']
```

Stabilność

Zastosowania stabilności: sortowania po wielu kluczach (np. najpierw imiona, potem nazwiska – typowe sortowanie w arkuszach kalkulacyjnych):

Anna Nowak
Maria Nowak
Zuzanna Kowalska
Marek Nowak
Maria Kowalska
Adam Nowak
Marek Wójcik
Anna Kowalska
Anna Wójcik

Adam Nowak
Anna Nowak
Anna Kowalska
Anna Wójcik
Marek Nowak
Marek Wójcik
Maria Nowak
Maria Kowalska
Zuzanna Kowalska

Anna Kowalska
Maria Kowalska
Zuzanna Kowalska
Adam Nowak
Anna Nowak
Marek Nowak
Maria Nowak
Anna Wójcik
Marek Wójcik

Podstawowe algorytmy

Niektóre podstawowe algorytmy ogólnego zastosowania:

1. Selection sort (sortowanie przez wybieranie),
2. Insertion sort (sortowanie przez wstawianie),
3. Merge sort (sortowanie przez scalanie),
4. Quicksort (sortowanie szybkie).

Oprócz tego:

1. Inne podstawowe algorytmy: np. Heapsort, Tree sort, Shellsort.
2. Algorytmy do zastosowań szczególnych (np. sortowania list liczb: Radix sort, Bucket sort; dla ograniczenie wielu wartości: Counting sort; ...),
3. Algorytmy hybrydowe: Timsort, Introsort, ...,
4. ...

Selection sort (sortowanie przez wybieranie)

Idea: iterujemy przez kolejne indeksy listy. Pod kolejny indeks wstawiamy najmniejszy element, który nie został jeszcze wstawiony pod swój indeks.

Kroki algorytmu $\mathrm{selection_sort}$ (*lst*: lista, *n*: długość listy):

1. Dla $i = 0, 1, \ldots, n-1$:
   1. Znajdujemy minimalny element spośród $lst[i], lst[i+1], \ldots, lst[n-1]$.
   2. Zamieniamy go miejscami z $lst[i]$.

Niezmienniki pętli: niech i to numer (kroku) iteracji:

1. PRE(i): na początku i-tej iteracji, wszystkie elementy o indeksach $< i$ są uporządkowane i nie większe od elementów o indeksach $\geqslant i$.
2. POST(i): na końcu i-tej iteracji, wszystkie elementy o indeksach $\leqslant i$ są uporządkowane i nie większe od elementów o indeksach $> i$.

Insertion sort (sortowanie przez wstawianie)

Idea: budujemy listę przez wstawianie kolejnych elementów na właściwą pozycję, jeśli trzeba przesuwając niektóre elementy.

Kroki algorytmu $\mathrm{insertion_sort}$ (*lst*: lista, *n*: długość listy):

1. Dla $i = 0, 1, \ldots, n-1$:
   1. Zapamiętujemy element $x = lst[i]$.
   2. Szukamy pierwszego indeksu $j \leqslant i$ takiego, że elementy o mniejszych indeksach są nie większe od $lst[i]$.
   3. Przesuwamy element x pod indeks j poprzez kolejne zamienianie go miejscami z elementami o indeksach (kolejno) $i-1, i-2, \ldots, j$.

Wtedy:

1. Na początku i-tej iteracji, wszystkie elementy o indeksach $< i$ są uporządkowane.
2. Na końcu i-tej iteracji, wszystkie elementy o indeksach $\leqslant i$ są uporządkowane.

„Dziel i zwyciężaj”

„Dziel i zwyciężaj” („divide and conquer”, „divide et impera”) – technika w programowaniu polegająca na rekurencyjnym dzieleniu zadanego problemu na podproblemy, aż staną się dostatecznie małe, by je z osobna rozwiązać, oraz skonstruować z ich rozwiązań rozwiązanie głównego problemu.

Scalanie

Scalanie: łączenie dwóch (posortowanych) list w jedną.

Wejście: dwie posortowane listy długości n i m.
Wyjście: posortowana lista długości $n + m$ składająca się z wszystkich elementów list z wejścia.

Przykład:

```
l1 = [2, 3, 6, 10, 12]
l2 = [1, 4, 5, 7, 10, 16, 20]
result = merge(l1, l2)
# [1, 2, 3, 4, 5, 6, 7, 10, 10, 12, 16, 20]
```

Merge sort (sortowanie przez scalanie)

Idea: dzielimy i zwyciężamy.

Kroki algorytmu $\mathrm{merge_sort}$ (*lst*: lista):

1. Jeśli lista ma mniej niż dwa elementy, zwracamy ją.
2. Dzielimy *lst* na dwie połówki: *left* i *right*.
3. Każdą połówkę sortujemy rekurencyjnie algorytmem $\mathrm{merge_sort}$.
4. Posortowane połówki scalamy i zwracamy.

Quicksort (wersja pierwsza)

Idea: dzielimy listę na podlisty elementów „małych" i „dużych", sortujemy je osobno, łączymy.

Kroki algorytmu $\mathrm{quick_sort_easy}$ (*lst*: lista):

1. Jeśli lista ma mniej niż dwa elementy, zwracamy ją.
2. Wybieramy element z listy (tzw. *pivot*).
3. Tworzymy trzy listy:
   left: złożona z elementów *lst* mniejszych od pivota.
   middle: złożona z elementów *lst* równych pivotowi.
   right: złożona z elementów *lst* większych od pivota.
4. Zwracamy:
   $\mathrm{quick_sort_easy}(\mathit{left}) + \mathit{middle} + \mathrm{quick_sort_easy}(\mathit{right})$

Partycjonowanie listy

Fragment listy: wszystkie elementy pomiędzy zadanymi indeksami i, j włącznie (w skrócie: „fragment i, j").
Lista *lst* jest swoim własnym fragmentem (dla indeksów $0, len(lst) - 1$).

Partycjonowanie listy:
Wejście: lista obiektów (niekoniecznie posortowana) długości n. Cel: zmienić kolejność elementów tak, aby dla pewnego indeksu $0 \leqslant i \leqslant n - 1$:
(*) każdy element z fragmentu $0, i - 1$ był mniejszy lub równy od każdego elementu z fragmentu $i, n - 1$

Podobnie: partycjonowanie fragmentu na (niepuste) podfragmenty.

Partycjonowanie listy

Partycjonowanie fragmentu:

Krok 0: wybieramy element z fragmentu (tzw. *pivot*) i **zamieniamy go** z ostatnim elementem fragmentu.
Kroki algorytmu $\mathrm{partition}$ (*lst* – lista, *lo*, *hi* – końce partycjonowanego fragmentu)

1. Niech $i = lo, pivot = lst[hi]$.
2. Dla $j = lo, lo + 1, \ldots, hi$:
   Jeśli $lst[j] < pivot$: zamień $lst[i]$ z $lst[j]$, zwiększ i o 1.
3. Zamień $lst[i]$ z $lst[hi]$.
4. Zwróć i.

Niezmiennik pętli: Na początku iteracji, elementy z fragmentu $0, i-1$ są mniejsze od *pivot*, elementy z fragmentu $i, j-1$ nie; $i \leqslant j$.

Quicksort (sortowanie szybkie)

Idea: to co poprzednio, ale bez nowych list; sortujemy fragmenty.

Kroki algorytmu $\mathrm{quick_sort}$ (*lst*: lista, *lo*, *hi* – fragment do posortowania):

1. Jeśli $lo \geqslant hi$ (fragment jest pusty), koniec.
2. Partycjonujemy fragment *lo*, *hi* otrzymując *i* (indeks pivota).
3. Sortujemy fragmenty $lo, i-1$ oraz $i+1, hi$ algorytmem $\mathrm{quick_sort}$.

Porównanie algorytmów

Wszystkie złożoności asymptotycznie są podane dokładnie ($\Theta(\cdot)$) dla każdego przypadku.

Algorytm	BEST	AVG	WORST	Pamięć	Stabilny
Selection sort	n^2	n^2	n^2	1	NIE
Insertion sort	n	n^2	n^2	1	TAK
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	TAK
Quicksort	$n \log n$	$n \log n$	n^2	$\log n, n$	NIE
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	NIE
Timsort	n	$n \log n$	$n \log n$	n	TAK
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	NIE