Sprawozdanie z algorytmów sortowania

Eryk Andrzejewski (145 277)

Politechnika Poznańska, Marzec 2020

Spis treści

Wstęp

Celem niniejszego sprawozdania jest prezentacja wyników testów, które przeprowadziłem w celu porównania wydajności różnych algorytmów sortujących, w różnych sytuacjach.

W teście wzięły udział następujące algorytmy:

- sortowanie bąbelkowe (bubble sort)
- sortowanie przez wstawianie (insertion sort)
- sortowanie przez wybieranie (selection sort)
- sortowanie Shella (Shell sort)
- sortowanie przez scalanie (merge sort)
- sortowanie szybkie (quick sort) - przetestowałem dwie wersje algorytmu: jedna z nich jako pivot obiera zawsze prawy element, a druga wybiera losowy, by zminimalizować ryzyko wystąpienia sytuacji pesymistycznej
- sortowanie przez kopcowanie (heap sort)
- sortowanie przez zliczanie (counting sort)

Testy odbywały się na różnych zbiorach danych i o różnych rozmiarach (ilość sortowanych elementów oznaczam jako N):

- przebieg losowy - całkowicie losowo wybrane N liczb całkowitych
- przebieg rosnący - kolejne liczby całkowite z zakresu od 1 do N, czyli dane już posortowane w porządku rosnącym
- przebieg malejący - kolejne liczby całkowite z zakresu od N do 1, czyli dane już posortowane w porządku malejącym
- przebieg stały - N-krotnie powtórzona ta sama liczba całkowita
- przebieg A-kształtny - ciąg N liczb całkowitych, które najpierw rosną (od 1 do $\frac{N}{2}$), a następnie maleją (od $\frac{N}{2}$ do 1)

Algorytmy zaimplementowałem w języku C++, mimo wcześniejszych zamiarów wykonania tego w Pythonie - o wyborze zadecydowała wydajność programów działających natywnie, która pozwala testować algorytmy na większych zbiorach danych.

Testowanie umożliwił mi skrypt, napisany w Pythonie, który generuje dane odpowiedniego typu i o odpowiednim rozmiarze, a następnie testuje kolejne algorytmy dla tych danych. Skrypt sterowany jest poprzez plik konfiguracyjny w formacie JSON. Wyniki również zapisywane są w pliku JSON, który następnie jest odczytywany przez kolejny skrypt, generujący wykresy. Wyniki testów, w postaci wykresów, zostały zamieszczone w niniejszym sprawozdaniu.

Każdy identyczny test (czyli takiego samego algorytmu, dla takiego samego typu i rozmiaru danych) został powtórzony kilkukrotnie, a pośrednie wyniki uśrednione, w celu zminimalizowania wpływu losowych czynników, powiązanych z wydajnością komputera, na wyniki testów.

Ze względu na duże różnice czasowe w działaniu pomiędzy pewnymi grupami algorytmów, postanowiłem podzielić je na dwa wykresy.

Wykresy oznaczone jako *część 1* obejmują algorytmy, które w danej kategorii są powolniejsze. Testy wykonano na danych o rozmiarze z zakresu: [10 000, 100 000]

Wykresy oznaczone jako *część 2* obejmują algorytmy, które w danej kategorii są wyraźnie szybsze niż pozostałe. Testy wykonano na danych o rozmiarze z zakresu: [5 000 000, 50 000

000]

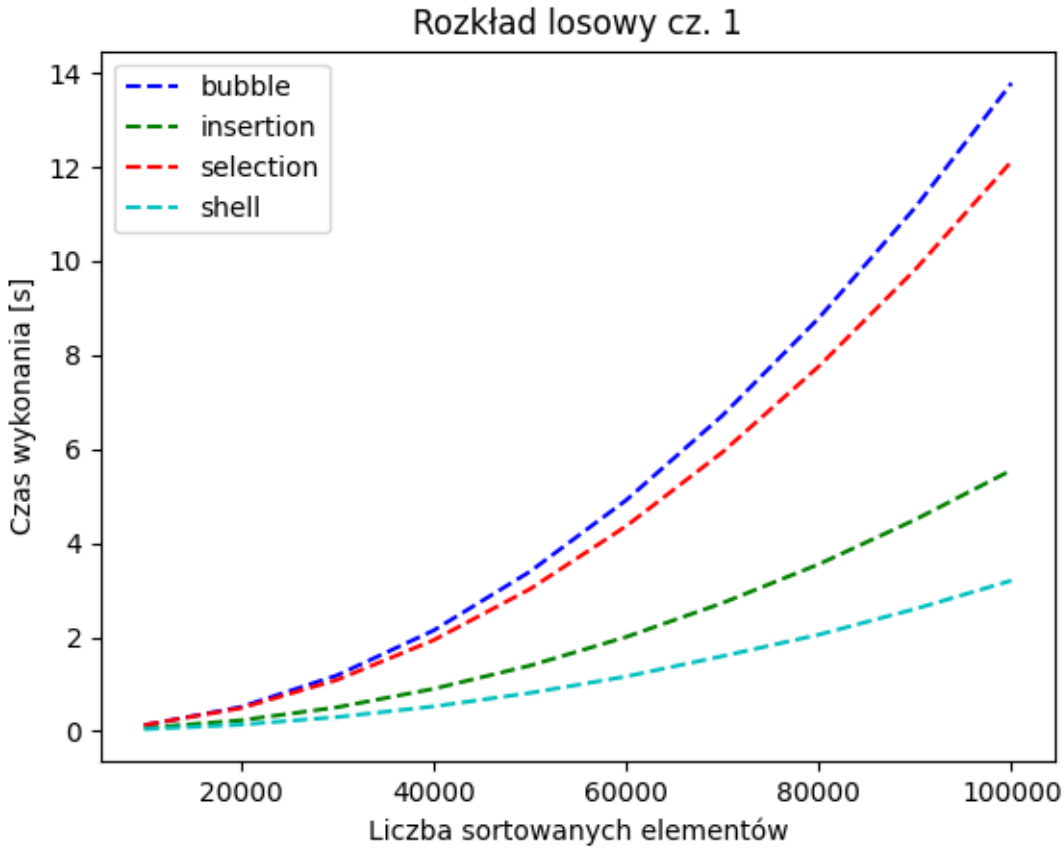
Wszystkie testy, które zostały zamieszczone w tym sprawozdaniu, odbyły się na laptopie Dell Latitude 7480 (Intel i5-7300U, 8GiB RAM) pracującym na systemie Linux Mint. Zaznaczę jednak, że każdy test został przeze mnie powtórzony wielokrotnie, a część testów wykonałem również na laptopie Lenovo Thinkpad x230 (Intel i5-3320M, 8GiB), również pracującym na systemie Linux Mint. Różnice pomiędzy testami wykonanymi na tym samym sprzęcie były znikome, wręcz niezauważalne. Mimo tego, można zauważyć wyraźne różnice podczas porównywania wyników testów z różnych laptopów (na Dellu jeden algorytm okazuje się być szybszy niż inny, choć testy na ThinkPadzie wskazywały inaczej). Być może ma to jakiś związek z optymalizacjami sprzętowymi i szczegółami architektury danej linii i generacji procesorów; raczej na pewno nie ma to związku z samym oprogramowaniem, gdyż obydwa laptopy pracują pod kontrolą tej samej wersji systemu (instalowane były niedawno, z dokładnie tego samego nośnika), obydwa laptopy korzystały również z tej samej kompilacji programu **sorter** (czyli program podlegający testowaniu, podczas kompilacji zastosowano optymalizacje na poziomie -O2). Oczywiście nie wykluczam popełnienia przeze mnie jakiegoś błędu, postaram się to jeszcze zweryfikować.

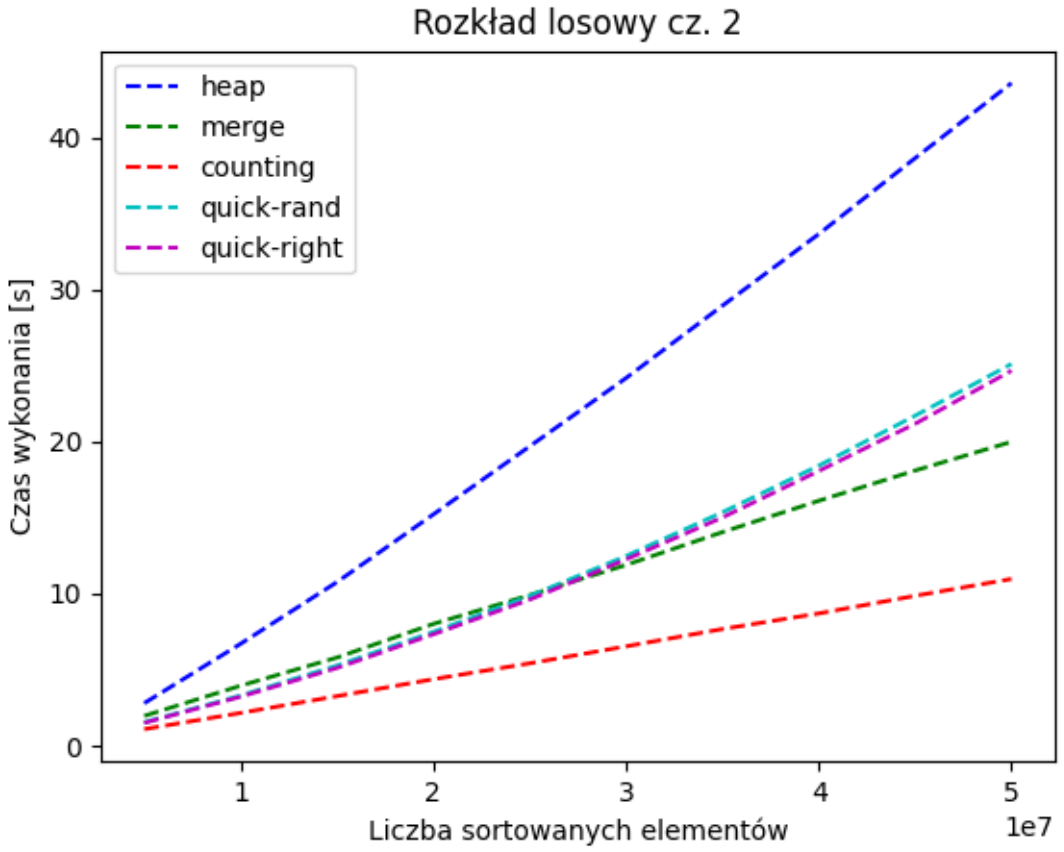
Prezentacja wyników

Przeprowadzając testy dołożyłem wszelkich starań, aby ich wyniki były rzetelne, a wykresy odzwierciedlały złożoności obliczeniowe algorytmów. Wielokrotnie wykonywałem testy, starałem się znaleźć możliwie najlepszy zakres wielkości danych poddawanych testowi. Częściowo się to udało, mam tu na myśli głównie wykresy z części pierwszej, a częściowo nie - te z części drugiej są nieco odkształcone, na co miały wpływ zapewne jakieś czynniki zewnętrzne. Mimo stosunkowo długich czasów wykonywania testów, nie udało się w całości tego problemu wyeliminować. Niemniej jednak można zaobserwować pewne wyraźne zależności czasowe, pomiędzy określonymi algorytmami.

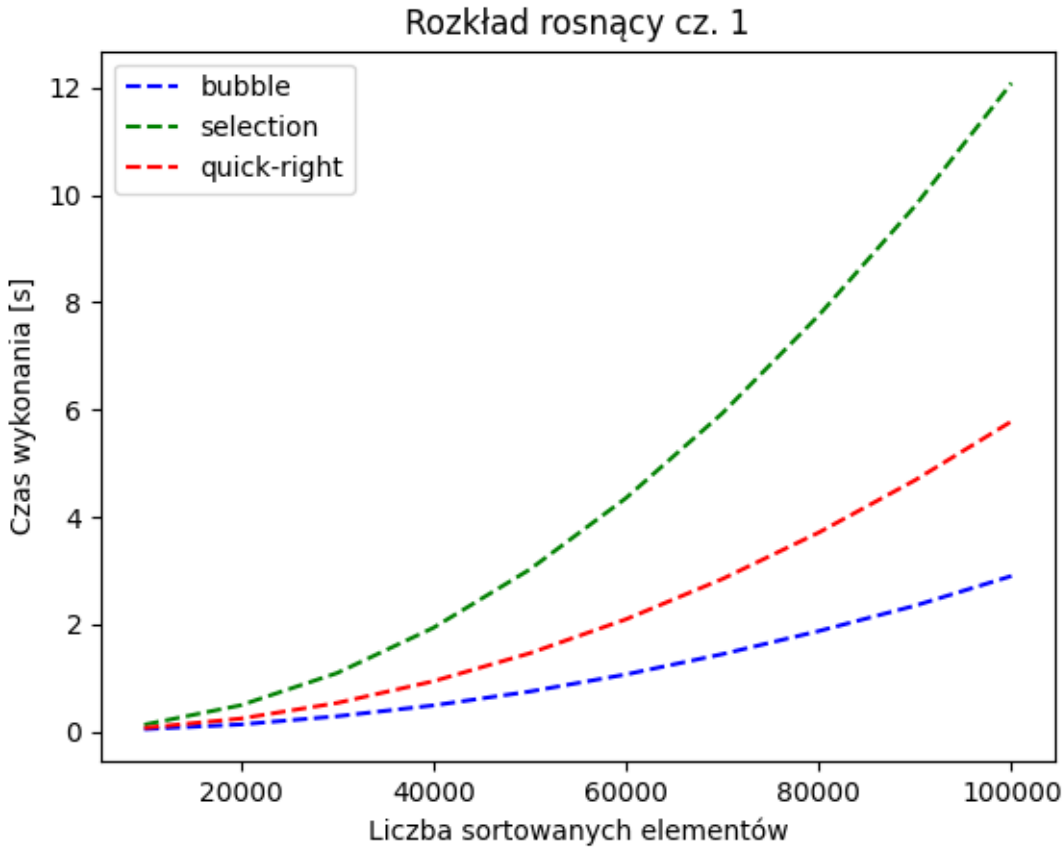
Zaznaczę jeszcze, że nie zawsze widać wyraźnie linie każdego algorytmu - jest to spowodowane nakładaniem się poszczególnych wykresów. Nie powinno to jednak stanowić dużego problemu w ich analizie.

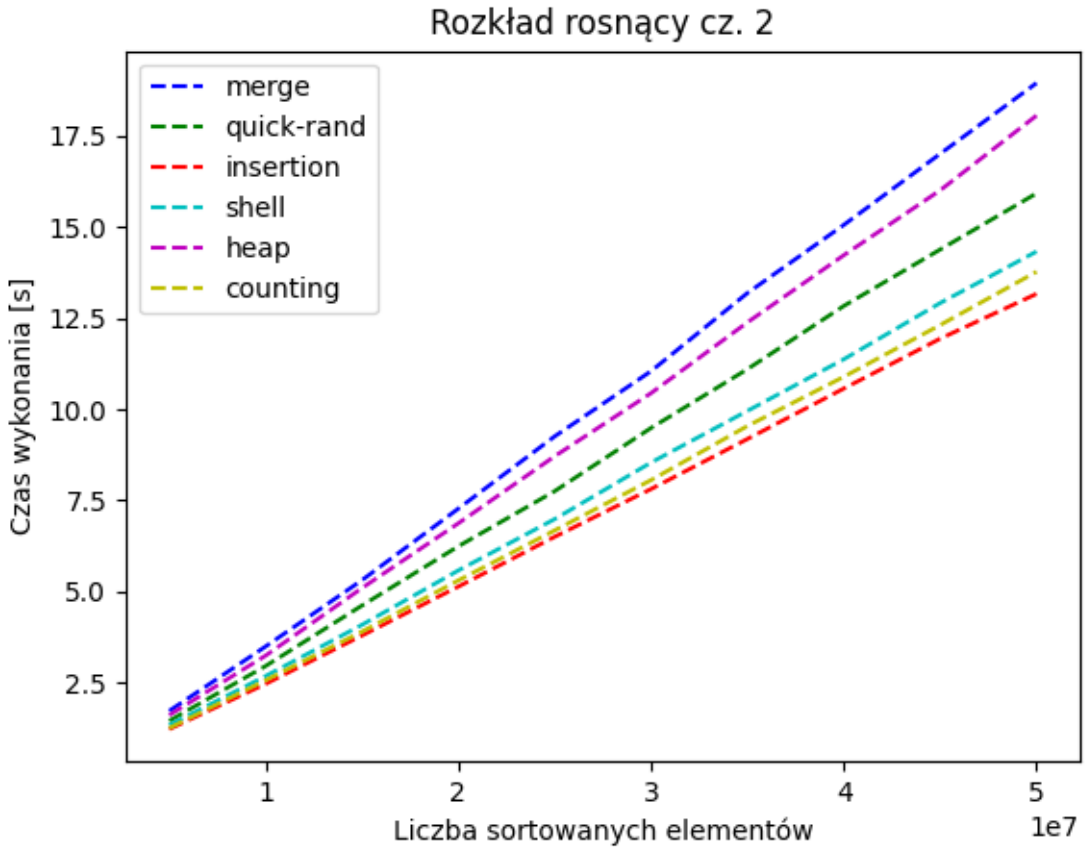
Rozkład losowy



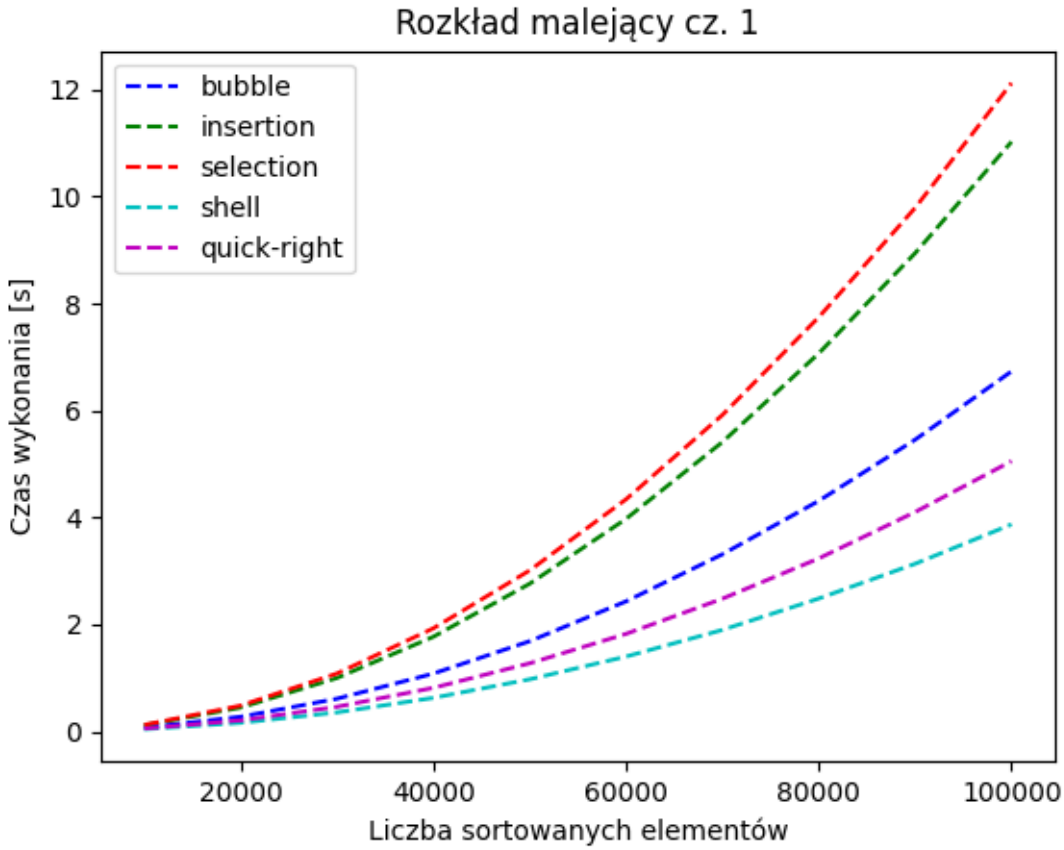


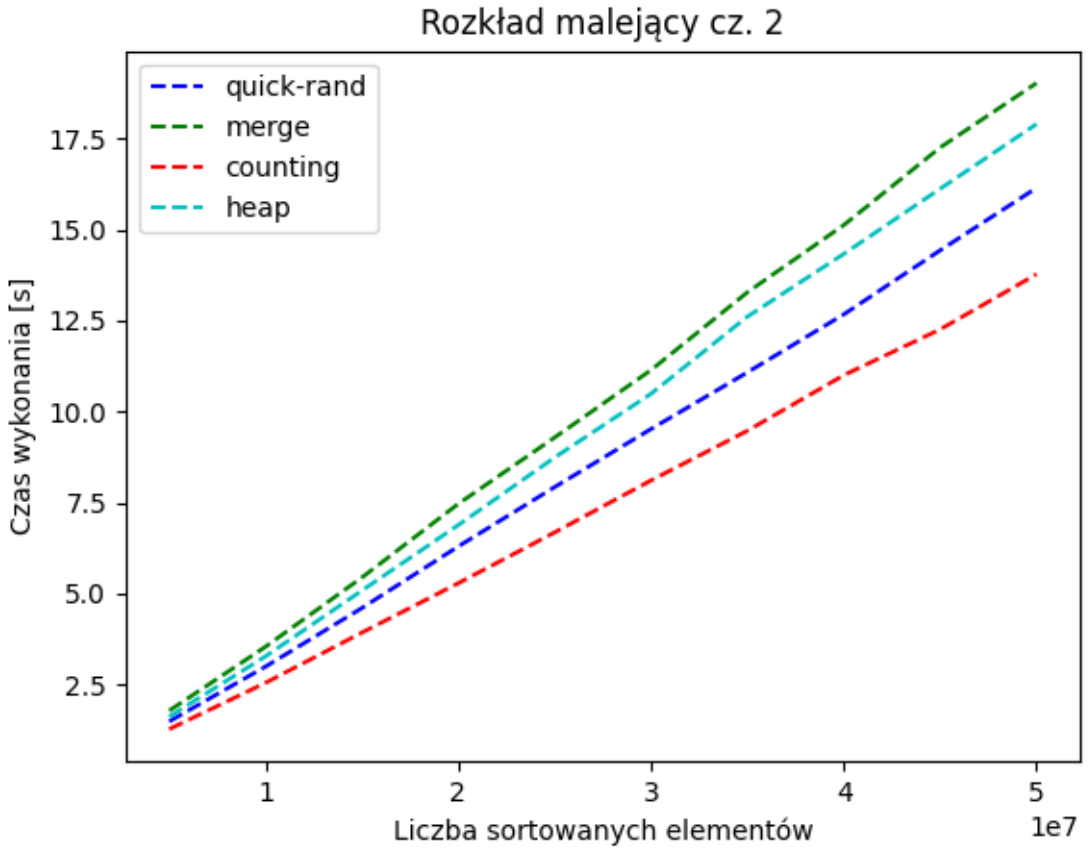
Rozkład rosnący





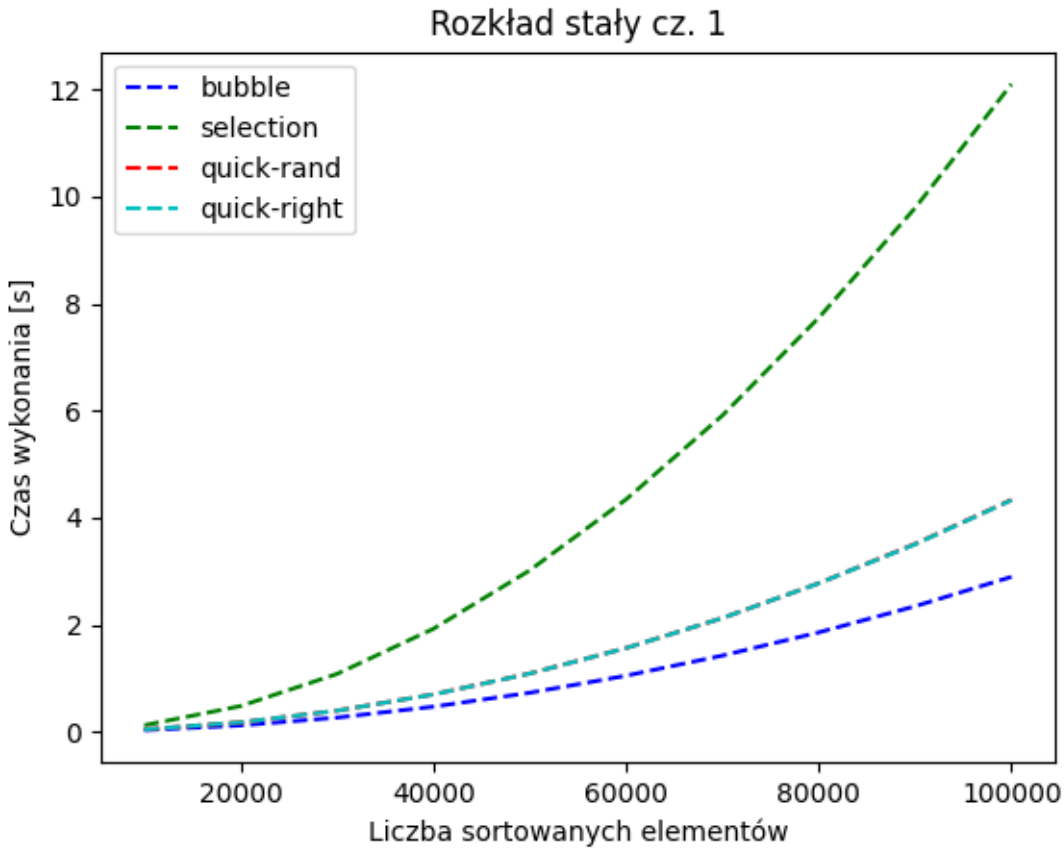
Rozkład malejący

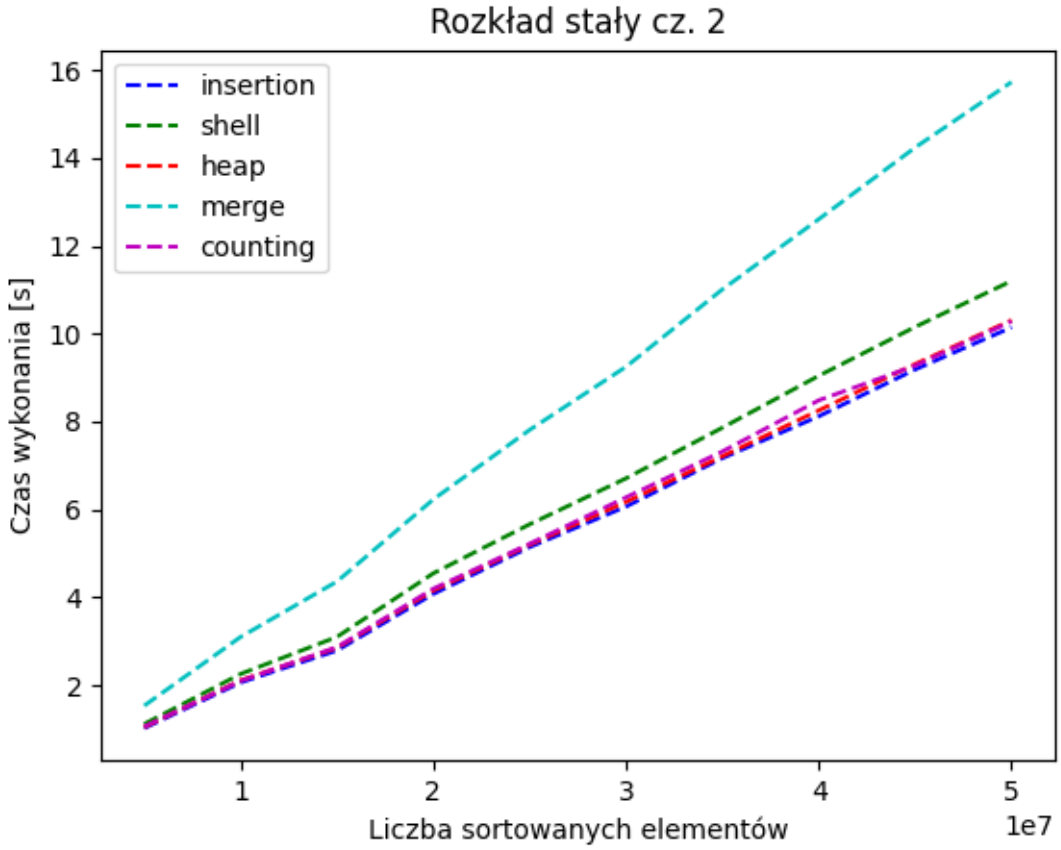




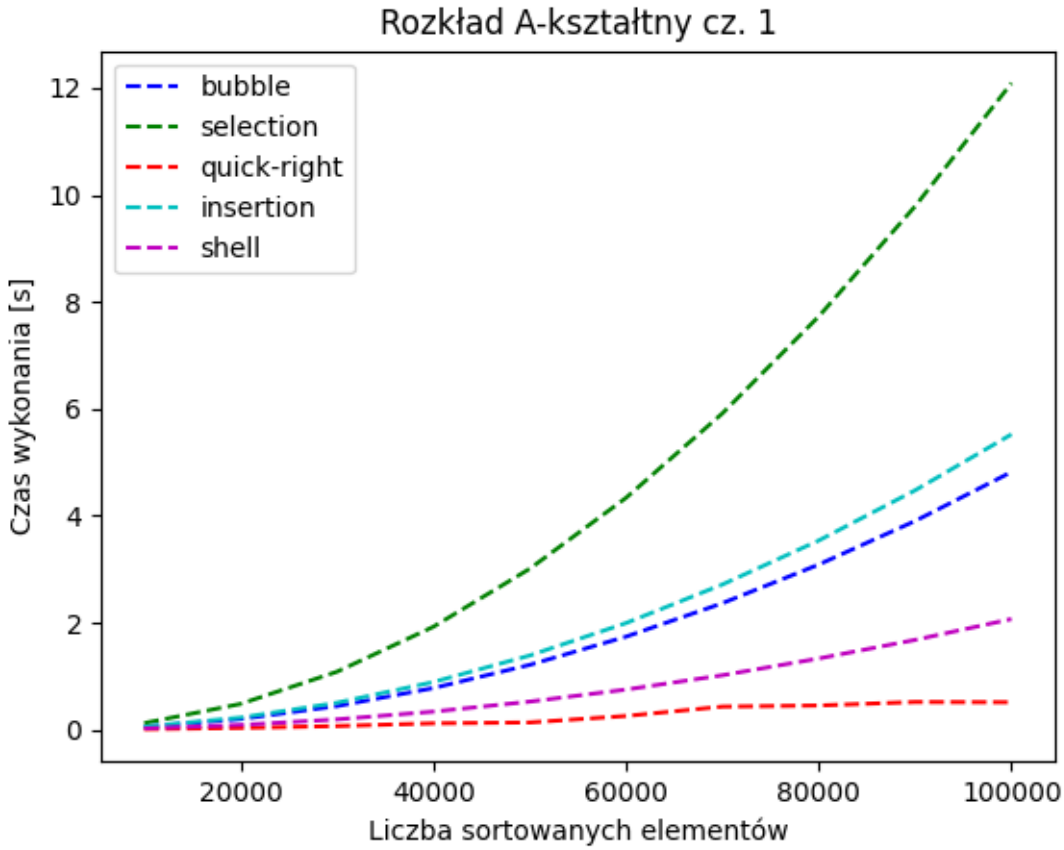
Rozkład stały

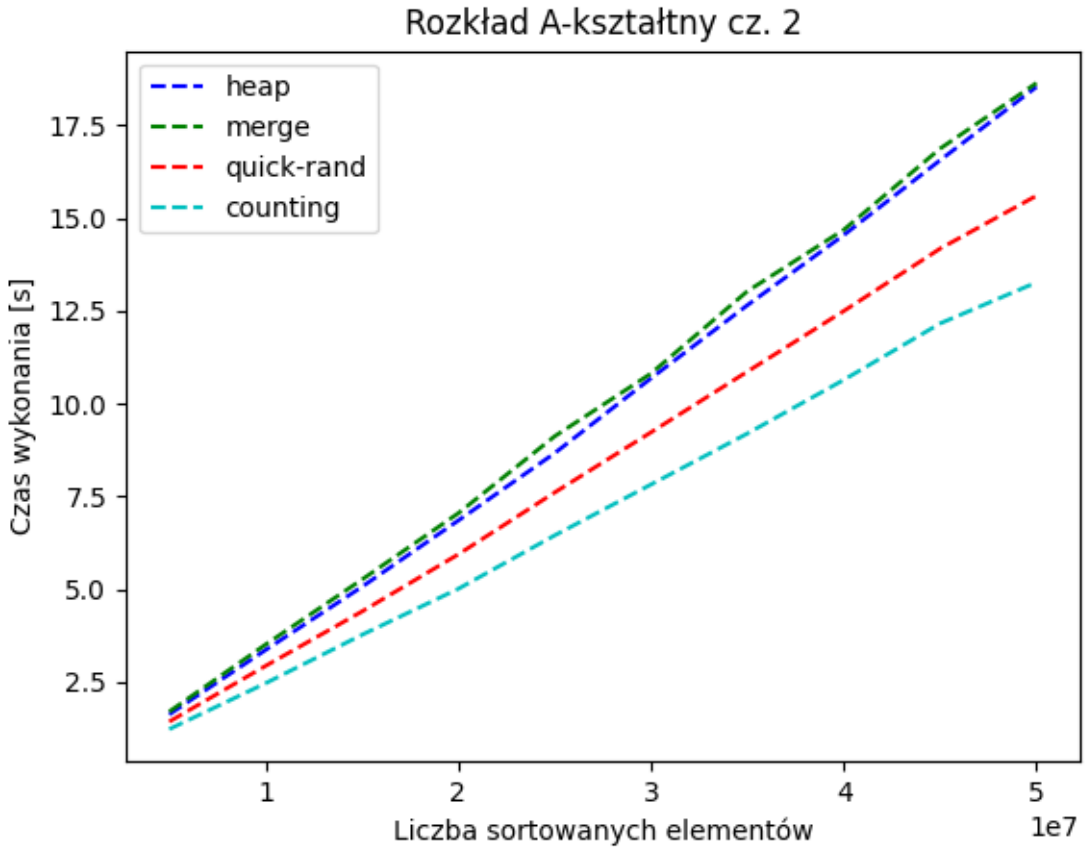
Uwaga, wykresy obu wariantów quick sorta się pokrywają!




Rozkład stały cz. 2
insertion
shell
heap
merge
counting
Czas wykonania [s]
16
14
12
10
8
6
4
2
1
2
3
4
5
1e7
Liczba sortowanych elementów

Rozkład A-kształtny





Wnioski

Bubble sort

Algorytm *sortowania bąbelkowego* jest bardzo prostym w założeniu algorytmem, ale jednocześnie bardzo powolnym. Algorytm porównuje dwa sąsiednie elementy i zamienia je, jeżeli ich kolejność jest niezgodna z kierunkiem sortowania. Po każdej takiej turze (trzeba za każdym razem iterować po nieposortowanej jeszcze części tablicy), kolejna liczba znajdzie się na swojej prawidłowej pozycji.

Jego złożoność obliczeniowa jest rzędu $O(n^2)$, ponieważ implementacja składa się z dwóch pętli (jedna jest zagnieżdżona w drugiej), które wykonują się pewną ilość razy, związaną w sposób liniowy z n. Wysoka złożoność przyczynia się do tego, że czas wykonania algorytmu wzrasta szybko, wraz ze wzrostem wielkosści danych.

Złożoność pamięciowa jest stała ($O(1)$), ponieważ algorytm wykonuje się w miejscu i nie wymaga dodatkowej pamięci do działania.

Algorytm jest stabilny.

Jest to jeden ze słabszych algorytmów, które zostały poddane przeze mnie testom. Ze względu na niską wydajność w praktycznie każdej sytuacji, trudno mi wskazać dla niego jakieś poważne (nieedukacyjne) zastosowanie.

Selection sort

Algorytm *sortowania przez wybieranie* jest kolejnym algorytmem, który jest bardzo prosty i intuicyjny - ale tak samo, jak sortowanie bąbelkowe, jest bardzo powolny. Założenie jest proste - szukamy najmniejszej liczby w tablicy i umieszczamy ją na jej początku, w miejscu gdzie powinna się znaleźć.

Jego złożoność obliczeniowa jest rzędu $O(n^2)$, ponieważ implementacja składa się z dwóch pętli (jedna jest zagnieżdżona w drugiej - w moim kodzie nie jest to wyraźnie widoczne, ze względu na wydzielenie tej wewenętrznej do osobnej funkcji); obydwie te pętle wykonują się pewną ilość razy, która jest zależna liniowo od n.

Złożoność pamięciowa jest stała ($O(1)$), ponieważ algorytm wykonuje się w miejscu i nie wymaga dodatkowej pamięci do działania.

Ten algorytm, poza prostotą, również nie ma zbyt wielu zalet. Jest bardzo powolny, w niektórych szczególnych przypadkach nawet wolniejszy, niż sortowanie bąbelkowe, więc nie nadaje się do wydajnego sortowania większych zbiorów danych.

Insertion sort

Algorytm *sortowania przez wstawianie* jest kolejnym przedstawicielem grupy algorytmów, które są mało wydajne - również ma złożoność kwadratową, o czym wspomnę za chwilę. Niemniej jednak posiada pewną bardzo dużą zaletę - bardzo szybko wykonuje się dla zbiorów danych, które już zostały posortowane - to wyróżnia go już na tle sortowania bąbelkowego i sortowania przez wybieranie.

Idea jest taka, że abstrakcyjnie dzielimy tablicę na dwie części: lewa to dane już posortowane, a prawa to te, które czekają na posortowanie. Wybieramy pierwszy element z prawej tablicy, a następnie porównujemy go z kolejnymi elementami lewej tablicy (iterując od prawej strony) i zamieniamy ze sobą sąsiadujące elementy. Robimy to tak długo, aż znajdziemy jego prawidłowe położenie (czyli lewy sąsiad będzie mniejszy lub równy, a prawy sąsiad będzie większy).

Algorytm posiada złożoność kwadratową $O(n^2)$. W optymistycznym przypadku, dane już są posortowane, algorytm porównuje tylko sąsiednie elementy (dla każdej pary robi to jednokrotnie) i nie musi nawet ich ze sobą zamieniać - to czyni go bardzo szybkim dla takiego zbioru danych (złożoność w tym przypadku jest rzędu $O(n)$).

Zdecydowanie gorzej jest w przypadku pesymistycznym, czyli wtedy, gdy dane są posortowane w przeciwnym kierunku (w tym przypadku malejąco). Wtedy wartości muszą być przesuwane zawsze aż na początek tablicy i właśnie to jest przyczyną kwadratowej złożoności obliczeniowej.

Złożoność pamięciowa jest stała ($O(1)$), ponieważ algorytm wykonuje się w miejscu i nie wymaga dodatkowej pamięci do działania.

Algorytm jest stabilny.

Shell sort

Algorytm ten został wymyślony przez pana Donalda Shella i zaprezentowany w roku 1959. Jest on pewnym ulepszeniem sortowania przez wstawianie (insertion sort), który to porównuje zawsze sąsiednie elementy. Algorytm Shella wprowadza natomiast pewną innowację - wartości porównywane są ze sobą w kilku turach. Każda tura polega na porównywaniu elementów odległych o pewną wartość (nazywam ją odległością), która w kolejnych turach ulega zmianie.

Istnieje wiele wariantów tego algorytmu, które różnią się metodą wyznaczania odległości. Dla wielu z nich złożoność obliczeniowa nadal nie została wyznaczona i jest problemem otwartym. Ja zaimplementowałem jeden z najprostszych możliwych wariantów, zgodny z oryginałem. Pierwszy odstęp to połowa wielkości danych (bierzemy tylko część całkowitą), a każdy następny jest dwukrotnie mniejszy (za każdym razem również rozpatrujemy tylko część całkowitą wyniku). Złożoność pesymistyczna tego wariantu algorytmu to $O(n^2)$.

Złożoność pamięciowa jest stała ($O(1)$), ponieważ algorytm wykonuje się w miejscu i nie wymaga dodatkowej pamięci do działania.

Wykonane przeze mnie testy wskazują, że algorytm ten w większości przypadków jest wydajniejszy, niż insertion sort (poza rozkładami: rosnącym oraz stałym).

Quick sort

Sortowanie szybkie to jeden z najpopularniejszych algorytmów sortowania. Jego idea oparta jest na zasadzie “dziel i zwyciężaj”. Zasada jest taka, że obieramy pewien element z tablicy jako tzw. pivot, czyli oś. Następnie dokonujemy partycjonowania, czyli wszystkie liczby mniejsze od pivota umieszczamy po jego lewej stronie, a większe - po prawej (wiąże się to z zamianą liczb miejscami, w tym również ze zmianą pozycji pivota). Po takim procesie, możemy być pewni, że pivot leży “na swoim” miejscu, czyli na tej pozycji, na której powinien

się znaleźć po całym procesie sortowania. Wystarczy tylko rekurencyjnie posortować mniejsze zbiory danych - po lewej oraz po prawej stronie pivota, wyłączając jego samego. Jako, że rekurencja musi mieć swój warunek stopu - przestajemy wywoływać ją w momencie, gdy dojdziemy do jednoelementowych zbiorów danych.

W optymistycznym przypadku, po procesie partycjonowania, pivot znajdzie się na środku tablicy, więc kolejne rekurencyjne wywołania funkcji sortującej będą operowały na danych o rozmiarze dwukrotnie mniejszym. Ilość rekurencyjnych wywołań będzie zależała więc logarytmicznie od ilości danych. Proces partycjonowania ma złożoność liniową (gdyż jest to pętla, która musi przeiterować jednokrotnie po całej tablicy). Wobec tego, algorytm *quick sort*, jako złożenie tych dwóch złożoności, w przypadku optymistycznym ma złożoność $O(n \log n)$.

W przypadku pesymistycznym, po procesie partycjonowania, pivot będzie znajdował się na skraju tablicy (lewym, bądź prawym). W takim przypadku, kolejne rekurencyjne wywołania funkcji *quick sort* będą operowały na zbiorze danych tylko o jeden mniejszym. To sprawia, że w tej sytuacji złożoność obliczeniowa będzie rzędu $O(n^2)$.

Przygotowałem dwa warianty tego algorytmu; obydwa zostały poddane testom. Jeden z nich zawsze obiera na pivot prawy, skrajny element. Drugi natomiast za każdym razem losuje pozycję pivota.

Dla całkowicie losowego zbioru danych, wybór wariantu praktycznie nie ma znaczenia (teoretycznie, losowanie w tym przypadku może niepotrzebnie wydłużyć czas działania algorytmu). Jednak, w sytuacji gdy ciąg danych jest już posortowany (rosnąco lub malejąco), wersja z losowaniem pivota likwiduje nam w praktyce problem pesymistycznej złożoności. W związku z tym, program jest odporny na ataki DoS (atakujący, wiedząc iż usługa korzysta z quick sorta, mógłby celowo podawać dane w określonym porządku, aby maksymalnie wydłużyć czas działania algorytmu), ale też po prostu wydajniejszy w takich sytuacjach. Uważam więc, że warto zdecydować się na wariant z losowaniem.

Złożoność pamięciowa jest zależna od implementacji, skupię się jednak na mojej własnej. Trzeba pamiętać, że ze względu na rekurencję, przy każdym wywołaniu funkcji, na stosie umieszczane są pewne informacje o poszczególnym wywołaniu (adres powrotu oraz argumenty). Wobec czego ilość pamięci, jaką używa program, jest zależna od ilości rekurencyjnych wywołań funkcji sortującej - mówiłem o tym chwilę wcześniej. Dla optymistycznego przypadku będzie to $O(\log n)$, gdyż rozmiary danych maleją dwukrotnie przy każdym wywołaniu. Dla pesymistycznego przypadku będzie to $O(n)$, ponieważ rozmiary maleją o 1 przy każdym wywołaniu.

Ze względu na to, iż informacje o kolejnych wywołaniach funkcji są przechowywane na stosie, a stos posiada stosunkowo niewielki, z góry określony rozmiar (na systemie Windows jest to zazwyczaj 1MB, a na systemie Linux zazwyczaj 8MB - stos jest przypisany do każdego wątku), istnieje możliwość przepełnienia tego stosu, w szczególności w przypadku pesymistycznym, więc należy za wszelką cenę starać się temu zjawisku zapobiec - i to jest kolejna zaleta wariantu losującego pivota, gdyż jak wspominałem wcześniej, zapobiega on wystąpieniu przypadku pesymistycznego.

Merge sort

Sortowanie przez scalanie to kolejny algorytm, który opiera się o zasadę "dziel i zwyciężaj". Idea jest taka, że dzielimy dane na dwie, mniej więcej równe, części (w przypadku danych o rozmiarze nieparzystym, jedna z części będzie o jeden dłuższa). Podziały te wykonywane są tak długo, aż dojdziemy do pojedynczych elementów. Następnie dokonuje się scalania tych części - do nowej tablicy zapisujemy liczby kolejności zgodnej z kierunkiem sortowania (wychodzimy już z założenia, że w danej części kolejność liczb jest poprawna, ponieważ dana część jest wynikiem scalania jej podczęści, albo jest pojedynczym elementem). Robimy to poprzez porównanie wartości, na które wskazuja dwa wskaźniki, dopisywanie do tablicy wynikowej mniejszego z nich, a następnie odpowiednie przesuwanie wskaźnika.

Złożoność obliczeniowa jest rzędu $O(n \log n)$, ponieważ ilość rekurencyjnych wywołań zależy logarytmicznie od wielkości danych, a operacja scalania ma złożoność liniową.

Złożoność pamięciowa zależy od wykorzystania stosu przez rekurencję (ono jest logarytmiczne, o czym była mowa wcześniej, w przypadku quick sorta), ale również od tego, że wyniki zapisujemy do innej tablicy - złożoność pamięciowa wynosi tutaj $O(n)$.

Merge sort jest wydajnym i dość prostym do zaimplementowania algorytmem. Testy wykazują jednak, że w testowanych przypadkach jest wolniejszy niż quick sort, cechuje się też większym zapotrzebowaniem na pamięć. Jego zaletą może być jednak stabilność, której brakuje quick sort-owi.

Heap sort

Algorytm sortowania przez kopcowanie wykorzystuje pojęcie kopca, czyli takiego drzewa binarnego, w którym każdy rodzic jest nie mniejszy, niż jego dzieci (a dzieci może być maksymalnie dwoje). Charakteryzuje się on tym, że maksymalny element znajduje się w jego korzeniu. Taka właściwość pozwala wykorzystać tę strukturę danych podczas sortowania. Nie trzeba jednak alokować dodatkowej pamięci, by przechowywać kopiec - wystarczy przechowywać dane w oryginalnej tablicy, a powiązania pomiędzy indeksami rodzica i dzieci wynikają z bardzo prostych zależności.

W mojej implementacji posiadam funkcję *fixHeap()*, która "naprawia" kopiec - porównuje wartości dzieci i rodzica, i w razie potrzeby zamienia je w celu przywrócenia poprawnej struktury kopca (wtedy robi to również rekurencyjnie, chyba że dane dziecko nie jest rodzicem żadnego innego dziecka). Na początku wywołuję tę funkcję dla każdego rodzica (w kolejności od ostatniego, do pierwszego - czyli korzenia). Po takiej serii wywołań, tablica danych jest poprawnym kopcem. W korzeniu znajduje się element maksymalny. Wobec tego zamieniamy korzeń z ostatnim elementem tablicy i nie bierzemy go więcej pod uwagę - jest już na swoim miejscu. Wywołujemy funkcję *fixHeap()* dla korzenia (a ona rekurencyjnie będzie się wywoływała do podrzędnych elementów), aby przywrócić kopcowi poprawną strukturę. Znowu w korzeniu znajdzie się element maksymalny (wykluczając ten element, który znajdował się w korzeniu wcześniej). Proces powtarzamy tak długo, aż w kopcu będą przynajmniej dwa elementy. W momencie, gdy zostanie tylko jeden -cały zbiór danych został posortowany.

Złożoność obliczeniowa wynosi $O(n \log n)$, ponieważ wysokość drzewa zależy logarytmicznie od wielkości danych (więc maksymalna liczba rekurencyjnych wywołań funkcji fixHeap zależy

również logarytmicznie od wielkości danych), a trzeba się przeiterować po $n-1$ elementach tablicy (złożoność liniowa) i dla nich wywołać funkcję *fixHeap()*.

Złożoność pamięciowa wynosi $O(\log n)$, ponieważ sortowanie występuje w miejscu i nie potrzebuje dodatkowego miejsca na dane, ale rekurencyjne wywoływanie funkcji fixHeap wymaga tworzenie stosu wywołań (o wysokości zależnej logarytmicznie od n).

Counting sort

Algorytm sortowania przez zliczanie jest, swego rodzaju, wyjątkiem - jako jedyny, spośród zaprezentowanych w tym sprawozdaniu algorytmów, posiada złożoność liniową. Zawdzięcza to nieco innemu podejściu, niż to, jakie towarzyszy pozostałym algorytmom.

Idea jest bardzo prosta - iterujemy po tablicy i zliczamy liczbę wystąpień poszczególnych wartości, tworząc histogram (zapisywany w osobnej tablicy). Następnie możemy użyć histogramu, aby przy pomocy kolejnej pętli wypisywać elementy.

W praktyce, algorytm ten niesie ze sobą pewne ograniczenia. Nadaje się on do sortowania liczb całkowitych z przedziału od 0 do k (lub takich danych, które można relatywnie łatwo do takiego przypadku sprowadzić). Wymaga zaalokowania pamięci o rozmiarze $k+1$ - wobec tego musimy posiadać informacje na temat zbioru danych (wartość minimalnego i maksymalnego elementu) przed sortowaniem, by móc z góry zaalokować tablicę o odpowiednim rozmiarze, albo musimy poświęcić trochę czasu na poznanie tych informacji w czasie wykonania. Jak się okazuje, jest to opłacalne, bo w większości przypadków, w których testowałem wydajność opisywanych w tym sprawozdaniu algorytmów, *counting sort* okazał się być najszybszy.

Wariant, który zastosowałem, jest nieco bardziej złożony, ponieważ wykorzystuje dodatkową pętlę dodającą wartości sąsiednich elementów histogramu, a rezultat zapisuje do nowej tablicy, iterując od tyłu na tablicy wejściowej. Takie podejście skutkuje jednak tym, że sortowanie jest stabilne (zachowujemy kolejność elementów o tej samej wartości).

Złożoność obliczeniowa jest rzędu $O(n+k)$, gdyż wykonujemy jedynie proste iteracje po elementach tablicy, nie ma tu żadnych zagnieżdżeń, ani rekurencji. Iterujemy zarówno po tablicy wejściowej (o rozmiarze n), jak i po histogramie (o rozmiarze k), więc wartości n i k sumujemy podając złożoność.

Złożoność pamięciowa jest rzędu $O(n+k)$, ponieważ wykorzystujemy dodatkową pamięć na histogram (o rozmiarze k), jak i na tablicę wynikową (o rozmiarze n).

W prezentowanych przeze mnie testach, algorytm ten osiąga tak dobre wyniki, ponieważ zbiór danych jest znacznie większy, niż zakres wartości danych. Dla coraz większych ilości danych, różnica ta jeszcze bardziej się nasila. Niestety trudność wykonania dokładnych testów, nastawionych nie tylko na rozmiar danych, ale również na zakres danych, spowodowała, że logistycznie nie byłem w stanie wyczerpać tematu (ilość obecnych wykresów zwiększyłaby się kilkakrotnie).

Do przechowywania danych oraz histogramu używam klasy std::vector z biblioteki standardowej języka C++, która jest kontenerem alokującym dane w sposób dynamiczny. Pozwala to uniknąć problemu związanego z przepełnieniem stosu, który pojawiłby się w przypadku automatycznej alokacji (czyli na stosie) tablic o zbyt dużym rozmiarze (nie pozwoliłby na zadeklarowanie histogramu o z góry zadanej wielkości, potrafiącej obsłużyć pełen zakres dla intów).