Quicksort – Sprawozdanie

Najlepszy przypadek: O()

Średni przypadek: O()

Najgorszy przypadek: O()

Quicksort z prawym kluczem

Quicksort z prawym kluczem - iteracja

Quicksort z prawym kluczem - rekurencja

Jak widać na wykresach Quicksort z prawym kluczem najgorzej zachowuje się dla ciągu malejącego (lub rosnącego). Dzieje się tak, ponieważ najgorszym możliwym przypadkiem dla tego algorytmu jest uczynienie kluczem elementu najmniejszego lub największego, bo podział zbioru wówczas jest bardzo nierówny. Jedna część jest zwyczajnie pusta, a w drugiej zostaje cała reszta liczb (bez klucza). Gdy kluczem ustanawiamy element skrajnie prawy to dla ciągu malejącego (lub rosnącego), zawsze wybieramy liczbę największą (lub najmniejszą). W przypadku ciągu A‑kształtnego wybór skrajnego elementu też nie jest najkorzystniejszy, bo również nie dzieli zbioru na mniej więcej równe części. Najlepiej wypada pod względem złożoności czasowej Quicksort z prawym kluczem dla danych losowych. Wtedy, wybierając liczbę najbardziej po prawej, mamy największą szansę na trafienie w medianę zbioru.

Quicksort z kluczem losowym

Quicksort z kluczem losowym – iteracja

Quicksort z kluczem losowym – rekurencja

Jak widać na wykresach Quicksort z losowym kluczem zachowuje się lepiej niż ten z prawym. Jest to dość logiczne, ponieważ, losując klucz wśród wszystkich elementów tablicy, mamy znacznie większe szanse na trafienie mediany niż w przypadku brania cały czas liczby najbardziej po prawej (przede wszystkim, gdy mówimy o ciągach monotonicznych). W przypadku takiej implementacji ułożenie danych wejściowych nie ma praktycznie znaczenia. Algorytm ten zachowuje się jedynie nieciekawie dla ciągów stałych, dzieląc zbiór na dwa nierówne podzbiory. Jeden pusty, a drugi zawierający resztę elementów. Nie znajduje on bowiem liczb większych od klucza, których poszukuje, by podzielić liczby w tablicy.

Dane wejściowe – ciąg losowy

Dla ciągu losowego trudno jest powiedzieć, jaki wybór klucza jest lepszy. Wszystko zależy od ułożenia tych danych. Średnio algorytmy te zachowują się podobnie. Raz trafi się mediana na końcu zbioru, raz uda się ją wylosować. Na wykresie widać, że może się zdarzyć (przy nieszczęśliwym losowaniu), że nawet dla mniejszych ilości danych czas działania może być dłuższy niż przy większej ilości danych (ale szczęśliwym wylosowaniu odpowiedniego klucza).

Dane wejściowe – ciąg stały

W przypadku ciągu stałego oczywiste jest, że kluczem zostaje wybrana liczba taka sama, jak wszystkie inne. W tej implementacji algorytmu szukaliśmy liczb większych od klucza i nie większych niż on, dlatego przypadek ciągu stałego jest niezbyt opytmistyczny. Wszystkie elementy (bez klucza) trafiają do jednego zbioru. Dzieje się tak n-1 razy, bo cały czas odchodzi ze zbioru jedna liczba, którą uznajemy za posortowaną.

Dane wejściowe – ciąg malejący

Wykres bardzo dobrze ilustruje to, co zostało już napisane. Algorytmy, które dla ciągu malejącego wybierają prawy klucz, działają dużo gorzej od tych, które go losują. W przypadku prawego klucza wszystkie inne elementy są od niego mniejsze i trafiają do jednego zbioru, który musimy posortować. Z każdym wywołaniem funkcji liczba elementów zmniejsza się tylko o jeden. Gdy losujemy klucz, mamy szansę trafić medianę i podzielić elementy na dwa prawie równoliczne podzbiory, dla których wykonamy sortowanie szybkie, co znacznie zmniejsza czas działania programu.

Dane wejściowe – ciąg A-kształtny

Chociaż algorytmy wybierające na klucz element skrajnie prawy zachowują się lepiej w przypadku ciągu A‑kształtnego niż malejącego, to i tak wypadają gorzej niż algorytmy losujące klucz. Dzieje się tak z podobnej przyczyny. Dla tych danych wejściowych prawy klucz, co prawda, dzieli zbiór na dwa podzbiory, dla których wykonane zostanie sortowanie, ale są one bardzo nieproporcjonalne. W przypadku losowania klucza możemy trafić dużo lepiej i podzielić zbiór na prawie równe części.