# Signature Sort

Jan Sochiera

23 października 2012

Zaawansowane Struktury Danych 2012

## Wstęp

W artykule porównane są różne metody sortowania liczb całkowitych oraz opisana dokładnie jedna z nich - Signature Sort.

Jako model obliczeniowy przyjmujemy model word-RAM, czyli taki, w którym wszystkie operacje bitowe oraz arytmetyczne wykonywane na jednym słowie maszynowym wykonują się w czasie stałym.

## 1 Porównanie metod sortowania

Poniższe zestawienie pokazuje złożoność różnych algorytmów sortowania liczb całkowitych w modelu RAM.

$\omega$ oznacza długość słowa maszynowego, jakim dysponujemy, a $b$ to maksymalna liczba bitów zajmowana przez liczbę. Domyślnie $b = \omega$.

- Porównania między elementami: $O(n \log n)$
- Sortowanie przez zliczanie: $O(n + 2^\omega)$
- Sortowanie pozycyjne (radix sort): $O(n\frac{\omega}{\log n})$
- Emde Boas: $O(n \log\frac{\omega}{\log n})$
- Han: $O(n \log\log n)$ - deterministyczny i $AC^0$ $RAM$
- Han and Thorup: $O\left(n\sqrt{\log\frac{\omega}{\log n}}\right)$
- Packed sort: $O(n)$ dla $\omega = \Omega(b \log n \log\log n)$
- Signature sort: $O(n)$ dla $\omega = \Omega(\log^{2+\varepsilon} n)$

# 2 Operacja Merge w czasie logarytmicznym

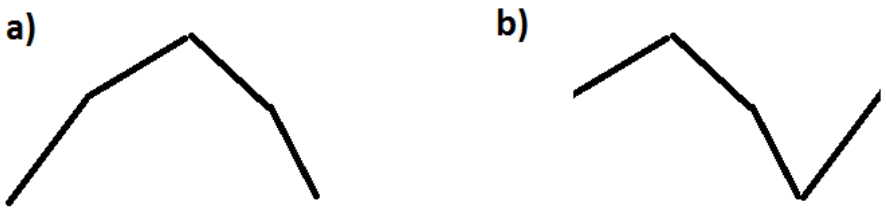
W tym dziale dowiemy się, jak wykonać operację scalania dwóch posortowanych ciągów w czasie logarytmicznym od długości ciągu. Zakładamy, że liczby są poukładane w dwóch połowach jednego słowa maszynowego i tworzą ciągi rosnące od początku do połowy oraz od połowy do końca. Chcemy otrzymać jedno słowo, w którym wszystkie liczby są posortowane.

Możemy tego dokonać odwracając pierwszą połowę słowa, a następnie sortując całe słowo będące ciągiem bitonicznym przy pomocy algorytmu Bitonic Sort. Jeżeli obie operacje wykonamy w czasie logarytmicznym, to cała operacja scalania również zostanie wykonana w czasie logarytmicznym.

## 2.1 Bitonic Sort

Bitonic Sort jest pomocniczym algorytmem służącym do sortowania ciągów bitonicznych.

**Ciąg bitoniczny** - cykliczny ciąg posiadający jedno minimum lokalne i jedno maksimum lokalne. Na rysuknu zarówno a) i b) są ciągami bitonicznymi.



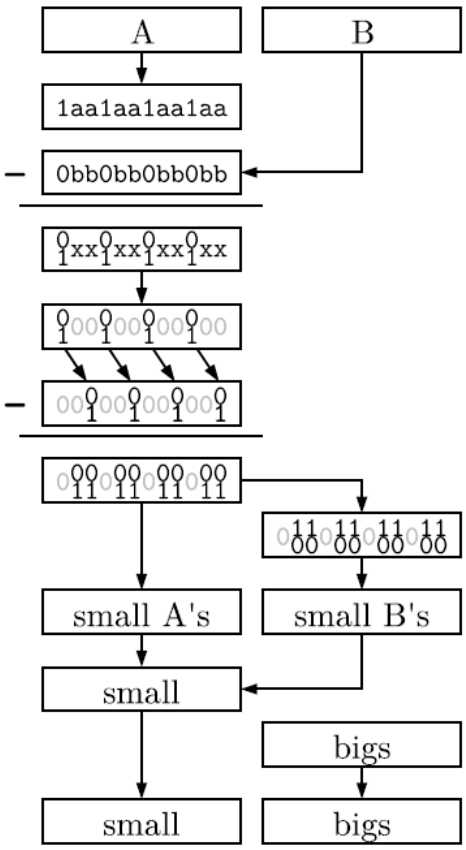

Algorytm Bitonic Sort:

- Podziel ciąg bitoniczny na 2 równe części: $L$ oraz $P$
- $\forall i$ jeśli $L_i > P_i$, zamień $L_i$ oraz $P_i$ miejscami
- Posortuj nowe $L$ i $P$ równolegle

Zakładając, że wszystkie liczby ciągu bitonicznego mieszczą się w jednym słowie maszynowym chcemy, by drugi krok algorytmu wykonywany był w czasie stałym oraz aby sortować wszystkie ciągi otrzymane z podziałów jednego ciągu jednocześnie. Możemy to wykonać wykorzystując operacje bitowe.

Zakładamy, że liczba zajmuje $b$ bitów, a przed każdą liczbą znajduje się bit pomocniczy. W pierwszej połowie ciągu ustawiamy go na 1, a w drugiej na 0. Odejmujemy od siebie otrzymane w ten sposób słowa. Na miejscu bitu pomocniczego pozostanie 1, jeżli liczba z pierwszej połowy ciągu była większa bądź równa albo zmieni się na 0 w przeciwnym przypadku. Zerujemy wszystkie bity niepomocnicze, a nastepnie odejmujemy od słowa jego przesunięcie w prawo o $b$ bitów. W ten sposób otrzymamy maskę bitową, w której całe słowo będzei pokryte jednykami jesli będzie większe niż jego odpowiednik w drugiej połowie ciągu i zerami jeśli mniejsze. Koniunkcja słowa z maską pozostawi tylko liczby większe niż ich konkurenci, a z negacją maski tylko mniejsze.

Wszystko przedstawia poniższy rysunek.

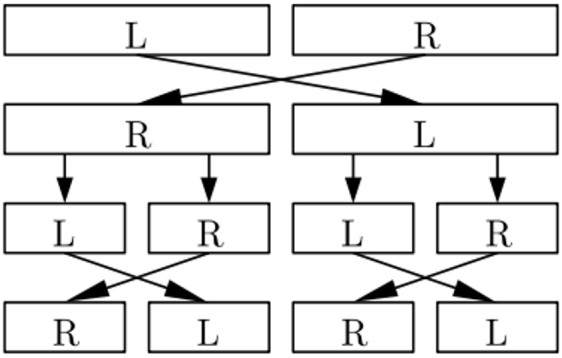


Ponieważ wszystkie operacje bitowe są identyczne dla ciągów tej samej wielkości, możemy wykonywać drugi krok algorytmu Bitonic Sort dla wszystkich otrzymanych ciągów naraz, gdyż mieszczą się one w jednym słowie maszynowym.

### 2.2 Odwracanie ciągu

Aby z dwóch ciągów rosnących otrzymać ciąg bitoniczny, nalezy odwrócić jeden z ciągów. Wykorzystując operacje bitowe można to zrobić szybciej niż liniowo.

Dzielimy ciąg na połowy i zamieniamy je miejscami. Nakładając odpowiednie maski bitowe i przesuwając słowa, możemy wykonać podzielenie słów na połowy i zamienienie połów miejcsami w czasie stałym.




### 2.3 Złożoność czasowa

Ponieważ każdy krok operacji Bitonic Sort oraz Reverse wykonuje się w czasie stałym w modelu RAM, przy czym po każdym kroku zmniejszamy długość pojedyczego fragmentu o połowę, wykonamy logarytmicznie wiele kroków od długości ciągu. W takim wypadku również czas merge wynosi $O(\log n)$.

## 3 Packed Sort

Algorytm Packed Sort jest warianetem sortowania przez scalanie działającym w czsie $O(n)$, o ile dłogość słowa $\omega \geqslant 2(b+1)\log n \log\log n$, gdzie $b$ jest liczbą bitów zajmowaną przez jedną liczbę.

Powyższe ograniczenie pozwala nam na upakowanie do jednej połowy słowa $\log n \log\log n$ liczb, gdzie przed każdą liczbą zostawiamy jeden wolny bit, a druga połowa słowa jest pusta.

Sortowanie przebiega w 3 krokach:

- Poukładanie po jednej liczbie w słowie, zostawiając 1 bit wolny przed każdą liczbą
- Scalenie liczb, aby w jednym słowie mieściło się ich $k = \log n \log\log n$.
- Scalenie wszystkich posortowanych ciągów $k$ liczb.

Drugi oraz trzeci punkt wykonujemy używając algorytmu Merge Sort stosując scalanie opisane w poprzednim rozdziale. Należy tylko zwrócić uwagę na to, że podczas scalania dwóch list posortowanych słów otrzymujemy słowo, którego pierwszą połowę przekazujemy na listę wynikową, a drugą wrzucamy z powrotem do listy zaczynającej się od większego elementu.

### 3.1 Złożoność czasowa

$$T(k) = 2T(\frac{k}{2}) + O(\log n) \Rightarrow T(k) = O(k)$$

Dowód:

$$\begin{aligned}
&\sum_{i=0}^{\log k-1} 2^i(\log k - i) = \sum_{i=0}^{\log k-1} 2^i \cdot \log k \; - \sum_{i=0}^{\log k-1} 2^i \cdot i \\
&= (1 + 2 + 4 + ... + k/2)\log k \; - \; 2^{logk}(\log k - 2) + 2 \\
&= (k-1) \cdot \log k - k \log k + 2k - 2 = 2(k-1) = O(k) \\
&\Rightarrow T(k) = O(k)
\end{aligned}$$

## 4 Signature Sort

Wykorzystując opisany wyżej algorytm Packed Sort, za pomocą Signature Sort możemy posortować $n$ liczb $\omega$-bitowych w czasie $O(n)$. Zakładamy przy tym, że $\omega \geqslant \log^{2+\varepsilon} n \log\log n$. Algorytm słada się z 7 oddzielnych części:

1. Podział liczb na kawałki
2. Tworzenie sygnatur
3. Sortowanie sugnatur
4. Budowa skompresowanego drzewa Trie
5. Rekurencyjne sortowanie krawędzi drzewa
6. Przywracanie prawidłowej permutacji krawędzi
7. Wypisywanie wszystkich wartości liści w porządku in-order

## 4.1 Podział liczb na kawałki

Po prostu dzielimy liczbę na $\log^{\varepsilon} n$ równych części. Podział jest tylko wirtualny, gdyż wszystkie kawałki nadal trzymane są w jednym słowie.

## 4.2 Tworzenie sygnatur

Następnie każdy z kawałków osobno haszujemy zastępując go $O(\log n)$-bitowym hashem. Wszystkie $\log^{\varepsilon} n$ częsci musimy posortować w czasie stałym, żeby hashowanie nie zdominowało czasu działania algorytmu. Możemy to zrobić maskując co drugi kawałek i mnożąc słowo przez jakąś liczbę - wtedy każdy kawałek zostanie pomnożony osobno. To samo robimy dla drugiej połowy kawałków. Szansa na kolizję jest bardzo bardzo mała, więc w razie czego możemy powtórzyć haszowanie.

W ten sposób otrzymaliśmy z liczby jej $O(\log^{1+\varepsilon} n)$-bitowy podpis.

## 4.3 Sortowanie sygnatur

Każdy z pospisów zajmuje $O(\log^{1+\varepsilon} n)$ bitów, a rozmiar słowa wynosi $\log^{2+\varepsilon} n \log\log n$, więc iloraz mniędzy nimi będzie równy $\Omega(\log n \log\log n)$. Takie ograniczenia pozdala nam posortować podpisy w czasie stałym za pomocą algorytmu Packed Sort.

## 4.4 Budowa skompresowanego drzewa Trie

Ponieważ posortowane sygnatury nie zachowały porządku pomiędzy prawdziwymi warościami liczb, mogłoby się wydawać, że sortowanie nie przynoiosło oczekiwanego efektu. Ale ponieważ każdy kawałek haszowany był osobno, okazuje się, że drzewo Trie utworzone z liczb w którym kawałki liczb są krawędziami, jest izomorficzne z drzewem utworonym z pospisów. Żeby zachować czas liniowy tworzymy skompresowane drzew Trie, czyli takie w którym wierzchołki nieposiadające rozgałezień zawarte są w swoich rodzicach.

## 4.5 Rekurencyjne sortowanie krawędzi drzewa

Aby przywrócić prawidłową kolejność wierzchołków, sortujemy je rekurencyjnie po trzech wartościach:

(numer wierzchołka, prawidłowa wartość kawałka, nr krawędzi)

Numer wierzchołka to jego wartość in-order, a prawidłowa wartość to wartość przed hashowaniem. Nr krawędzi zostawiamy, by móc następnie dokonać permutacji krawędzi i znaleźć odpowiednią kolejność liczb.

Pierwsza i trzecia wartość są $O(\log n)$-bitowe. Zostaną zdominowane przez drugą wartość, która zajmuje $\omega/\log^{\varepsilon} n$ bitów. Jako algorytmu sortującego używamy rekurencyjnie algorytmu Signature Sort. Po zagłębieniu się w rekurencję $1 + 1/\varepsilon$ razy, czyli stało liczbę, otrzymujemy do sortowania wyrażenie zajmujące $O(\log n + \frac{\omega}{\log^{1+\varepsilon} n}) = O(\frac{\omega}{\log^{1+\varepsilon} n}) = O(\frac{\omega}{\log n \log\log n})$ bitów. Wtedy możemy posortować je w czasie liniowym za pomocą algorytmu Packed Sort.

## 4.6 Przywracanie prawidłowej permutacji krawędzi

Mając posortowane krawędzie drzewa możemy teraz pozamienia ich kolejność na taką, jaka byłaby w oryginalnym drzewie Trie.

## 4.7 Wypisywanie wszystkich wartości liści w porządku in-order

Następnie wystarczy przejrzeć drzewo w porządku in-order i wypisywać właściwe wartości liści jako już posortowanych liczb całkowitych.

## Literatura

[1] Prof. Erik Demaine, Advanced Data Structures, Spring 2012

[2] Wolfram—Alpha: Computational Knowledge Engine