Kolejki Priorytetowe i Sortowanie Liczb Całkowitych

Jan Sochiera

Seminarium: Zaawansowane Struktury Danych

Na podstawie wykładu prof. Erik Demaine'a

7 lutego 2013

Plan Wykładu

Równoważność Pomiędzy Kolejką Priorytetową a Sortowaniem

M. Thorup wykazał, sortowanie n kluczy w czasie $O(nS(n,\omega))$ jest równoważne kolejce priorytetowej w której operacje wstawiania, usuwania elementu i znajdowania minimum wykonamy w $O(s(n,\omega))$, gdzie s jest funkcją od liczby elementów do posortowania i rozmiaru słowa mszynowego.

Dlatego sortowanie liniowe $\iff$ kolejka priorytetowa w czasie stałym!

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \log n)$

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$
- Sortowanie pozycyjne (radix sort): $O(n \frac{\omega}{log \ n})$

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$
- Sortowanie pozycyjne (radix sort): $O(n \frac{\omega}{log \ n})$
- van Emde Boas: $O(n \ log \frac{\omega}{log \ n})$

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$
- Sortowanie pozycyjne (radix sort): $O(n \frac{\omega}{log \ n})$
- van Emde Boas: $O(n \ log \frac{\omega}{log \ n})$
- Han: $O(n \ \ log \ log \ n)$ - deterministyczny i $AC^0 \ RAM$

Algorytmy Sortowania Liczb Całkowitych

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

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$
- Sortowanie pozycyjne (radix sort): $O(n \frac{\omega}{log \ n})$
- van 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 $\omega = \Omega(b \ log \ n \ \ log \ log \ n)$

Algorytmy Sortowania Liczb Całkowitych

- Porównania między elementami: $O(n \ log \ n)$
- Sortowanie przez zliczanie: $O(n + 2^{\omega})$
- Sortowanie pozycyjne (radix sort): $O(n \frac{\omega}{log \ n})$
- van 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 $\omega = \Omega(b \ log \ n \ \ log \ log \ n)$
- <u>Signature sort: $O(n)$ dla $\omega = \Omega(log^{2+\varepsilon} n)$</u>

Packed Sort

Algorytm "Packed Sorting" sortuje n b-bitowych liczb całkowitych w czasie $O(n)$, o ile rozmiar słowa $\omega \geq 2(b+1)\, log\ n\ log\ log\ n$.

Ograniczenie to pozwala zmieścić $k = log\ n\ log\ log\ n$ liczb w jednej połowie słowa dodatkowo zostawiając 1 bit wolny na początku każdej liczby.

Packed Sort

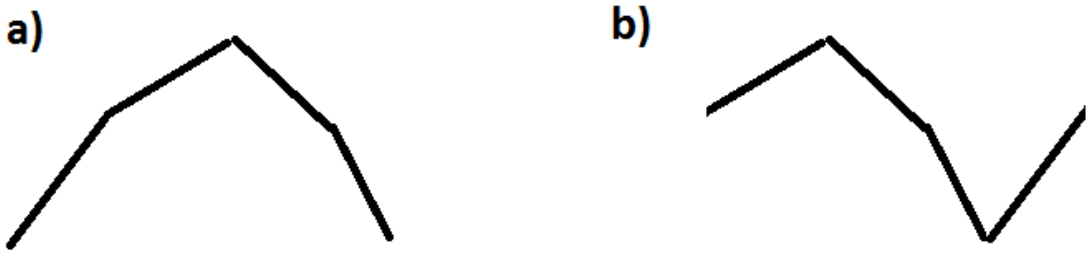
Algorytm "Packed Sorting" sortuje n b-bitowych liczb całkowitych w czasie $O(n)$, o ile rozmiar słowa $\omega \geq 2(b+1)\, log\ n\ log\ log\ n$.

Ograniczenie to pozwala zmieścić $k = log\ n\ log\ log\ n$ liczb w jednej połowie słowa dodatkowo zostawiając 1 bit wolny na początku każdej liczby.

Algorytm korzysta z pomocniczego algorytmu "Bitonic Sort", który zostanie przedstawiony najpierw.

Ciągi Bitoniczne

Ciąg Bitoniczny(?) (Bitonic Sequence) to cyklicznie przesunięte następujące po sobie 2 ciągi - niemalejący i nierosnący.



Sortowanie Bitoniczne

Bitonic Sort:

- Podziel ciąg bitoniczny na 2 równe części: L oraz P

Sortowanie Bitoniczne

Bitonic Sort:

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

Sortowanie Bitoniczne

Bitonic Sort:

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

Sortowanie Bitoniczne

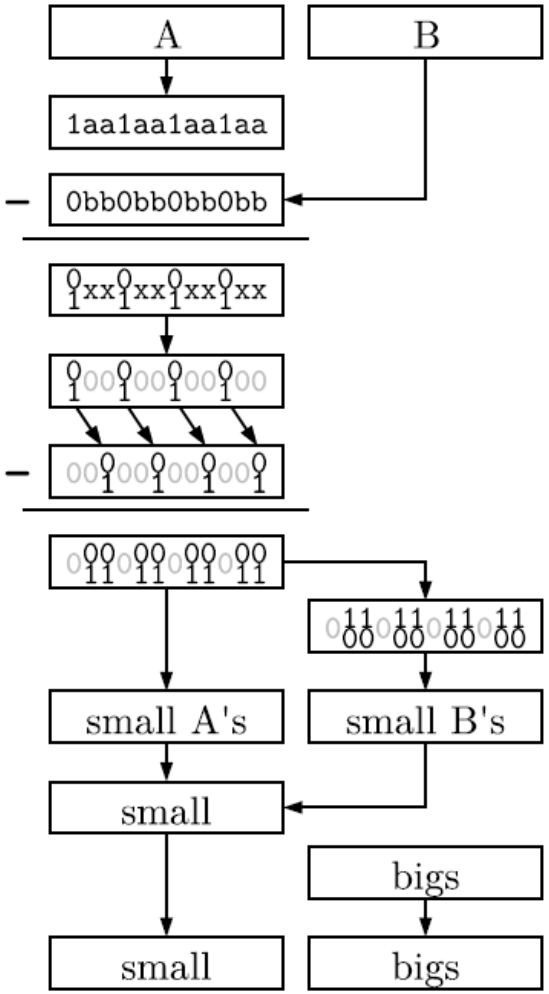
Bitonic Sort:

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

Ciągi Bitoniczne sortujemy równolegle, czyli wykonujemy krok algorytmu dla całego słowa.

Chcemy wykonać jeden krok algorytmu w czasie stałym.

Sortowanie Bitoniczne



Sortowanie Bitoniczne

Złożoność algorytmu:

$$T(k) = T(k/2) + O(1)$$

Sortowanie Bitoniczne

Złożoność algorytmu:

$$T(k) = T(k/2) + O(1)$$

$$\mathbf{T(k) = O(\log k)}$$

Packed Sort c.d.

Packed Sorting:

- Wszystkie liczby umieść w słowach po $k = log\ n\ log\ log\ n$ liczb w jednym słowie
- Posortuj każde słowo oddzielnie algorytmem sortowania porzez scalanie
- Scal wszystkie posortowane słowa tymże algorytmem

Packed Sort c.d.

Packed Sorting:

- Wszystkie liczby umieść w r słowach po $k = log\ n\ log\ log\ n$ liczb w jednym słowie
- Posortuj każde słowo oddzielnie algorytmem sortowania porzez scalanie
- Scal wszystkie posortowane słowa tymże algorytmem

Potrzebujemy operacji "Merge"

Scalanie Posortowanych Słów

Operację scalania na dwóch posortowanych słowach redukujemy do sortowania bitonicznego:

$$\text{Merge}(A,B) = \text{BitonicSort}(\ A + \text{Reverse}(B)\ )$$

Scalanie Posortowanych Słów

Reverse(S, i):

- Podziel słowo S na i par
- zamień miejscami elementy z każdej pary
- Reverse(S, 2i)

Scalanie Posortowanych Słów

Reverse(S, i):

- Podziel słowo S na i par
- zamień miejscami elementy z każdej pary
- Reverse(S, 2i)

$$(LP)^R = P^R L^R$$

Każdy krok algorytmu wykonujemy w czasie stałym.

Scalanie Posortowanych Słów

Koszt operacji Reverse(S,1):

$$T(k) = T(k/2) + O(1)$$

$$T(k) = O(log\ k)$$

Scalanie Posortowanych Słów

Koszt operacji Merge:

$$T(k) = O(log\ k) + O(log\ k) + O(1) = O(log\ k)$$

Scalanie Posortowanych List Posortowanych Słów

Koszt scalenia dwóch list po s słów każde:

$$T(2s) = 2s \cdot O(log\ k) = O(s\ log\ k)$$

Packed Sort c.d.

Packed Sorting:

- Wszystkie liczby umieść w r słowach po $k = log\ n\ log\ log\ n$ liczb w jednym słowie -> $O(n)$

Packed Sort c.d.

Packed Sorting:

- Wszystkie liczby umieść w r słowach po $k = log\ n\ log\ log\ n$ liczb w jednym słowie -> $O(n)$
- Posortuj każde słowo oddzielnie algorytmem sortowania porzez scalanie

  Koszt dla jednego słowa: $T(k) = 2T(k/2) + O(log\ k) = O(k)$

Packed Sort c.d.

Packed Sorting:

- Wszystkie liczby umieść w r słowach po $k = log\ n\ log\ log\ n$ liczb w jednym słowie -> $O(n)$
- Posortuj każde słowo oddzielnie algorytmem sortowania porzez scalanie

  Koszt dla jednego słowa: $T(k) = 2T(k/2) + O(log\ k) = O(k)$
- Scal wszystkie posortowane słowa tymże algorytmem

Packed Sort c.d.

Teza: jeśli w słowie zawiera się $k = log\ n\ log\ log\ n$ liczb, to algorytm Packed Sort wykona się w czasie $O(n)$.

Dowód.

Równanie rekurencyjne opisujące czas działania algorytmu jest następujące:

$$T(n) = 2T(n/2) + O(n/k\ log\ k)$$

$$T(k) = O(k)$$

Na każdym poziomie wykonamy $O(n/k\ log\ k)$ operacji, a poziomów rekurencji jest $log\ \frac{n}{k}$.

$$T(n) = O(\frac{n}{k} \cdot log\ k \cdot \log\ \frac{n}{k}) \leq O(\frac{n}{log\ n\ log\ log\ n} \cdot log\ log\ n \cdot \log\ n) = O(n)$$

□

Signature Sort

Signature Sort

Złożoność algorytmu liniowa, o ile $\omega = \Omega(log^{2+\varepsilon} n)$.

Załóżmy, że $\omega \geq log^{2+\varepsilon} n \; log \; log \; n$.

Signature Sort

- Podziel każdą liczbę na $log^{\varepsilon} n$ kawałków.

Signature Sort

- Podziel każdą liczbę na $log^{\varepsilon} n$ kawałków.
- W liniowym czasie oblicz dla każdego kawałka jego $O(log\ n)$-bitowy hasz.

W ten sposób każda liczbą będzie reprezentrowana przez $O(log^{1+\varepsilon})$-bitowy podpis (signature).

Signature Sort

- Podziel każdą liczbę na $log^{\varepsilon} n$ kawałków.
- W liniowym czasie oblicz dla każdego kawałka jego $O(log\ n)$-bitowy hasz.

  W ten sposób każda liczbą będzie reprezentrowana przez $O(log^{1+\varepsilon})$-bitowy podpis (signature).
- Posortuj podpisy w liniowym czasie za pomocą Packed Sorting.

Signature Sort

- Podziel każdą liczbę na $log^{\varepsilon} n$ kawałków.
- W liniowym czasie oblicz dla każdego kawałka jego $O(log\ n)$-bitowy hasz.

  W ten sposób każda liczbą będzie reprezentrowana przez $O(log^{1+\varepsilon})$-bitowy podpis (signature).
- Posortuj podpisy w liniowym czasie za pomocą Packed Sorting.
- Utwórz skompresowane drzewo trie, dla którego krawędziami będą wartości kawałków po haszowaniu.

Signature Sort

- Utwórz skompresowane drzewo trie, dla którego krawędziami będą wartości kawałków po haszowaniu.
- Rekurencyjnie posortuj wszystkie krawędzie drzewa leksykograficznie po (NodeID, właściwa wartość kawałka, Nr krawędzi w danym węźle).
  Liczba rekursji jest stała i wynosi $1 + 1/\varepsilon$.

Signature Sort

- Utwórz skompresowane drzewo trie, dla którego krawędziami będą wartości kawałków po haszowaniu.
- Rekurencyjnie posortuj wszystkie krawędzie drzewa leksykograficznie po (NodeID, właściwa wartość kawałka, Nr krawędzi w danym węźle).
  Liczba rekursji jest stała i wynosi $1 + 1/\varepsilon$.
- Zmień kolejność krawędzi w wierzchołkach na taką, jaka została obliczona podzas ostatniego sortowania.

Signature Sort

- Utwórz skompresowane drzewo trie, dla którego krawędziami będą wartości kawałków po haszowaniu.
- Rekurencyjnie posortuj wszystkie krawędzie drzewa leksykograficznie po (NodeID, właściwa wartość kawałka, Nr krawędzi w danym węźle).
  Liczba rekursji jest stała i wynosi $1 + 1/\varepsilon$.
- Zmień kolejność krawędzi w wierzchołkach na taką, jaka została obliczona podzas ostatniego sortowania.
- Przeszukaj drzewo umieszczając liście kolejno na liście wynikowej.

Dziękuję za uwagę.