Algorytmy i Struktury Danych

Sortowanie 1

(c) Marcin Sydow

Tematy wykładu:

- problem sortowania
- sortowanie przez wybór (SelectionSort)
- sortowanie przez wstawianie (InsertionSort)
- sortowanie przez złaczanie (MergeSort)
- struktura danych list dowiązaniowych

Sortowanie

Input: S - ciąg elementów, które mogą być uporządkowane za pomocą pewnej relacji porządku liniowego $\leq_R$(np. liczby naturalne); len - długość ciągu S (liczba naturalna)

Output: S' - ciąg składający się z tych samych elementów co S, uporządkowany niemalejąco (tzn. $\forall_{0<i<len} S[i-1] \leq_R S[i]$)

W tym kursie, dla uproszczenia dyskusji, rozważamy jako elementy sortowane liczby naturalne, ale nie ma to wpływu (poza CountSort) na większość omawianych algorytmów.

Waga problemu sortowania

Sortowanie jest jednym z najważniejszych podstawowych zadań dotyczących praktycznego przetwarzania danych. Tematyka sortowania była bardzo intensywnie badana od połowy 20. wieku i większość używanych obecnie algorytmów powstała wiele dekad temu.

Sortowanie jest używane w bardzo wielu kontekstach, np.:

- przyspieszenie wyszukiwania
- przyspieszenie operacji na danych, np. operacji na bazach danych, etc.
- wizualizacja danych
- obliczanie pewnych ważnych statystyk

I wiele innych

Sortowanie przez wybór (Selection Sort)

Pomysł sortowania przez wybór jest bardzo prosty. W sortowanym ciągu znajdujemy minimum, zamieniamy je miejscami z pierwszym elementem ciągu i kontynuujemy to samo na ciągu zaczynającym się od drugiego indeksu, dopóki bieżący ciąg ma więcej niż 1 element.

```
selectionSort(S, len){
  i = 0
  while(i < len){
    mini = indexOfMin(S, i, len)
    swap(S, i, mini)
    i++
  }
}
```

gdzie:

`indexOfMin(S, i, len)` - jest pomocniczą funkcją zwracającą indeks minimum wśród elementów $S[j]$, gdzie $i \leq j < len$

`swap(S, i, mini)` - zamiana miejscami elementów S[i] i S[mini]

Jaki jest niezmiennik pętli (zewnętrznej)?

Sortowanie przez wybór (Selection Sort)

Pomysł sortowania przez wybór jest bardzo prosty. W sortowanym ciągu znajdujemy minimum, zamieniamy je miejscami z pierwszym elementem ciągu i kontynuujemy to samo na ciągu zaczynającym się od drugiego indeksu, dopóki bieżący ciąg ma więcej niż 1 element.

```
selectionSort(S, len){
  i = 0
  while(i < len){
    mini = indexOfMin(S, i, len)
    swap(S, i, mini)
    i++
  }
}
```

gdzie:

`indexOfMin(S, i, len)` - jest pomocniczą funkcją zwracającą indeks minimum wśród elementów $S[j]$, gdzie $i \leq j < len$

`swap(S, i, mini)` - zamiana miejscami elementów S[i] i S[mini]

Jaki jest niezmiennik pętli (zewnętrznej)?
(pierwsze i elementów ciągu jest posortowane)

Selection Sort - Analiza

Operacja dominująca: porównanie 2 elementów w tablicy
Rozmiar danycyh: długość ciągu (len)

W pętli zewnętrznej wykonane będzie $len - 1$ iteracji, w i-tej iteracji szukamy minimum w ciągu o długości (len-i) elementów (i odpowiednio tyle jest porównań)

$$W(len) = \sum_{i=1}^{len-1} i = \frac{len(len-1)}{2} = \Theta((len)^2)$$

Algorytm ten ma więc **kwadratową** pesymistyczną złożoność czasową.

Zauważmy, że przeciętna złożoność czasowa tego algorytmu $A(len)$ jest taka sama jak pesymistyczna, gdyż algorytm SelectionSort dla danej długości ciągu *zawsze* (niezależnie od zawartości ciągu) wykonuje taką samą liczbę porównań - nawet dla ciągu wejściowego, który **już jest posortowany**

Sortowanie przez wstawianie (Insertion Sort)

Począwszy od drugiej pozycji w sortowanym ciągu, w iteracji *next* "przepychamy" bieżący (o indeksie *next*) element wstecz, aż znajdziemy (używając porównań) dla niego właściwą pozycję, tak, żeby ciąg pierwszych $next + 1$ elementów był posortowany. Następnie zwiększamy bieżącą pozycję (*next*) o 1 aż do ostatniego elementu ciągu.

```
insertionSort(arr, len){

  for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp < arr[curr - 1])){

      arr[curr] = arr[curr - 1];
      curr--;
    }

    arr[curr] = temp;
  }
}
```

Jaki jest niezmiennik pętli zewnętrznej?

Sortowanie przez wstawianie (Insertion Sort)

Począwszy od drugiej pozycji w sortowanym ciągu, w iteracji *next* "przepychamy" bieżący (o indeksie *next*) element wstecz, aż znajdziemy (używając porównań) dla niego właściwą pozycję, tak, żeby ciąg pierwszych $next + 1$ elementów był posortowany. Następnie zwiększamy bieżącą pozycję (*next*) o 1 aż do ostatniego elementu ciągu.

```
insertionSort(arr, len){

  for(next = 1; next < len; next++){

    curr = next;
    temp = arr[next];

    while((curr > 0) && (temp < arr[curr - 1])){

      arr[curr] = arr[curr - 1];
      curr--;
    }

    arr[curr] = temp;
  }
}
```

Jaki jest niezmiennik pętli zewnętrznej? Analogiczny do SelectionSort.

Insertion Sort - Analiza

(operacja dominująca i rozmiar danych (oznaczmy dla wygody przez n) takie same jak w poprzednim algorytmie)

Jaki jest najgorszy przypadek? (najwięcej porównań)?

Insertion Sort - Analiza

(operacja dominująca i rozmiar danych (oznaczmy dla wygody przez n) takie same jak w poprzednim algorytmie)

Jaki jest najgorszy przypadek? (najwięcej porównań)?

Kiedy dane są odwrotnie posortowane. Wtedy złożoność jest następująca:

$$W(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 + \Theta(n) = \Theta(n^2)$$

Zauważmy też, że jeśli dane wejściowe już są posortowane, algorytm działa bardzo szybko i wymaga tylko n-1 porównań. Zatem ten algorytm jest "bardziej inteligentny" niż poprzedni, gdyż **dostosowuje ilość pracy** do stopnia posortowania danych i w najgorszym przypadku (dane odwrotnie posortowane) wykonuje tyle pracy co Selection Sort dla każdego przypadku.

Analiza przeciętnej złożoności czasowej

Przypomnienie: do analizy przeciętnej złożoności czasowej należy założyć pewien model losowości danych wejściowych. Załóżmy prosty naturalny model losowości: dane są liczbami naturalnymi od 1 do n i każda permutacja jest jednakowo prawdopodobna. Wtedy, w itej iteracji zewnętrznej pętli algorytm wykona następującą przeciętną liczbę porównań:

$$\frac{1}{i}\sum_{j=1}^{i} j = \frac{1}{i}\frac{(i+1)i}{2} = \frac{i+1}{2}$$

W ten sposób, we wszystkich iteracjach otrzymamy łącznie:

$$A(n) = \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2}\sum_{k=2}^{n} k = \frac{1}{4}n^2 + \Theta(n) = \Theta(n^2)$$

Analiza, c.d.

Jak wynika z analizy, w przeciętnym przypadku algorytm ten jest 2 razy szybszy od algorytmu sortowania przez wybór, ale wciąż ma **kwadratową** złożoność czasową (czyli np. 3-krotny wzrost rozmiaru danych spowoduje ok. 9-krotny wzrost czasu działania niezależnie od implementacji i sprzętu).

Kwadratowa złożoność czasowa jest zbyt wysoka w przypadku dużych danych (rozważmy np. sortowanie miliarda liczb - ile porównań należałoby wykonać? Ile by to trwało nawet na szybkiej maszynie? Zauważmy, że miliard 8-bajtowych liczb to zaledwie 8GB danych w RAM)

Analiza, c.d.

Jak wynika z analizy, w przeciętnym przypadku algorytm ten jest 2 razy szybszy od algorytmu sortowania przez wybór, ale wciąż ma **kwadratową** złożoność czasową (czyli np. 3-krotny wzrost rozmiaru danych spowoduje ok. 9-krotny wzrost czasu działania niezależnie od implementacji i sprzętu).

Kwadratowa złożoność czasowa jest zbyt wysoka w przypadku dużych danych (rozważmy np. sortowanie miliarda liczb - ile porównań należałoby wykonać? Ile by to trwało nawet na szybkiej maszynie? Zauważmy, że miliard 8-bajtowych liczb to zaledwie 8GB danych w RAM)

Czy jest możliwe zaprojektowanie algorytmu sortującego istotnie szybciej niż z kwadratową złożonością?

Sortowanie przez złączanie (Merge Sort)

Jest to zastosowanie podejścia "Dziel i rządź" do sortowania. Rozważmy następujący pomysł na sortowanie:

1. podziel ciąg na 2 połowy
2. posortuj każdą połówkę oddzielnie
3. połącz posortowane połówki w całość

Zauważmy, że połączenie 2 posortowanych ciągów w jeden posortowany wymaga stosunkowo niewiele (liniowo wiele) porównań (dlaczego?) więc powyższy schemat ma szanse na bycie szybkim sposobem sortowania.

Ponadto, w punkcie 2, każda z połówek może być posortowana też zgodnie z całym schematem (**rekursja**), dopóki połówki mają długość większość niż 1.

Zatem, dostajemy gotowy rekurencyjny algorytm sortowania (przez złączanie).

Sortowanie przez złączanie (Merge Sort) - schemat


```
mergeSort(S, len){
  if(len <= 1) return S[0:len]
  m = len/2
  return merge(mergeSort(S[0:m], m), m,
               mergeSort(S[m:len], len-m), len-m)
}
```

używamy tu nieco nietypowej notacji (bardziej podobnej np. do Pythona), żeby zwięźle wyrazić algorytm.

- oznaczenie S[a:b] oznacza tu podciąg składający się z elementów S[i] takich, że $a \leq i < b$
- funkcja `merge(a1, len1, a2, len2)` złącza dwa posortowane podciągi a1 i a2 (o długościach len1 i len2, odpowiednio) i zwraca połączony i posortowany ciąg

Szczegóły funkcji merge

input: a1, a2 - posortowane ciągi liczb o długościach len1 i len2, odpowiednio
output: zwraca posortowany ciąg powstały z elementów ciągów a1 i a2

```
merge(a1, len1, a2, len2){

  i = j = k = 0;
  result[len1 + len2] // (alokacja pamięci)

  while((i < len1) && (j < len2))
    if(a1[i] < a2[j]) result[k++] = a1[i++];
    else result[k++] = a2[j++];

  while(i < len1) result[k++] = a1[i++];

  while(j < len2) result[k++] = a2[j++];

  return result;
}
```

Analiza funcji merge

`merge(a1, len1, a2, len2)`

dominująca operacja: porównanie 2 elementów lub indeksów

rozmiar danych: łączna długość obu ciągów: $n = len1 + len2$

złożoność czasowa: W(n) = A(n) =

[1] Przy zastosowaniu wyszukiwania binarnego liczba porównań w `merge` wynosi $O(log(n))$, ale i tak potrzeba $O(n)$ operacji przypisania wartości przy przesuwaniu

Analiza funcji merge

`merge(a1, len1, a2, len2)`

dominująca operacja: porównanie 2 elementów lub indeksów

rozmiar danych: łączna długość obu ciągów: $n = len1 + len2$

złożoność czasowa: $W(n) = A(n) = \Theta(n)$[1]

niestety, złożoność pamięciowa jest wysoka, gdyż alokujemy tablicę na połączone ciągi:

$S(n) =$

[1] Przy zastosowaniu wyszukiwania binarnego liczba porównań w `merge` wynosi $O(log(n))$, ale i tak potrzeba $O(n)$ operacji przypisania wartości przy przesuwaniu

Analiza funcji merge

`merge(a1, len1, a2, len2)`

dominująca operacja: porównanie 2 elementów lub indeksów

rozmiar danych: łączna długość obu ciągów: $n = len1 + len2$

złożoność czasowa: $W(n) = A(n) = \Theta(n)$[1]

niestety, złożoność pamięciowa jest wysoka, gdyż alokujemy tablicę na połączone ciągi:

$S(n) = \Theta(n)$

(można tego uniknąć jeśli dane są w formie tzw. **list dowiązaniowych** a nie tablic)

[1] Przy zastosowaniu wyszukiwania binarnego liczba porównań w `merge` wynosi $O(log(n))$, ale i tak potrzeba $O(n)$ operacji przypisania wartości przy przesuwaniu

Analiza algorytmu mergeSort

Aby obliczyć złożoność algorytmu mergeSort wystarczy obliczyć łączną złożoność wszystkich wywołań funkcji `merge`.

Na każdym poziomie rekursji funkcja `merge(s1, len1, s2, len2)` jest wywoływana na ciągach o łącznej długości *len* (najpierw na 2 potem na 4, etc.)

Ponieważ ciąg dzielony jest na 2 połowy, więc głębokość (liczba poziomów) rekursji wynosi $\Theta(log_2(len))$ (podobnie jak byłoby w rekurencyjnej wersji algorytmu binarySearch).

Analiza złożoności czasowej algorytmu mergeSort

`mergeSort(S, len)`

operacja dominująca: porównanie 2 elementów lub indeksów

rozmiar danych: długość ciągu wejściowego (len)

Zatem, mamy $\Theta(log_2(len))$ poziomów rekursji i na każdym poziomie wszystkie wywołania funkcji `merge` mają łączną złożoność $\Theta(len)$.

Podsumowując, otrzymujemy:

$$W(len) = A(len) = \Theta(len \cdot log(len))$$

- czyli złożoność jest **liniowo-logarytmiczna**
(lepsza od kwadratowej!)

Krótkie podsumowanie omówionych 3 algorytmów

sortowanie przez wybór i przez wstawianie:
proste algorytmy mające **kwadratową** złożoność czasową ($\Theta(n^2)$)

natomiast sortowanie przez złączanie (mergeSort) jest istotnie szybsze: **liniowo-logarytmiczne** ($\Theta(nlog_2(n))$), chociaż (w wersji, gdy ciągi przechowywane są w tablicach) niestety zużywa dużo (liniowo wiele) pamięci na kopiowane tablice **oraz** $\Theta(log_2(n))$ **na rekursję**.

Złożoność kwadratowa a liniowo-logarytmiczna

Zauważmy, że różnica pomiędzy złożonością kwadratową i liniowo-logarytmiczną jest praktycznie istotna i dla dużych danych może być bardzo znacząca.
rozważmy np. 100 milionów logów serwera WWW do posortowania (np. po adresie IP)

załóżmy, że można porównać miliard par logów na sekundę.

Złożoność kwadratowa a liniowo-logarytmiczna

Zauważmy, że różnica pomiędzy złożonością kwadratową i liniowo-logarytmiczną jest praktycznie istotna i dla dużych danych może być bardzo znacząca.
rozważmy np. 100 milionów logów serwera WWW do posortowania (np. po adresie IP)

załóżmy, że można porównać miliard par logów na sekundę.

Ile czasu zabierze to algorytmowi insertionSort?

Złożoność kwadratowa a liniowo-logarytmiczna

Zauważmy, że różnica pomiędzy złożonością kwadratową i liniowo-logarytmiczną jest praktycznie istotna i dla dużych danych może być bardzo znacząca.
rozważmy np. 100 milionów logów serwera WWW do posortowania (np. po adresie IP)

załóżmy, że można porównać miliard par logów na sekundę.

Ile czasu zabierze to algorytmowi insertionSort?

$(10^8)^2 p / 10^9 s = 10^7 s$ (115 dni)
A ile w przypadku algortymu mergeSort?
(załóżmy stałą multiplikatywną 1)

Złożoność kwadratowa a liniowo-logarytmiczna

Zauważmy, że różnica pomiędzy złożonością kwadratową i liniowo-logarytmiczną jest praktycznie istotna i dla dużych danych może być bardzo znacząca.
rozważmy np. 100 milionów logów serwera WWW do posortowania (np. po adresie IP)

załóżmy, że można porównać miliard par logów na sekundę.

Ile czasu zabierze to algorytmowi insertionSort?

$(10^8)^2 p / 10^9 s = 10^7 s$ (115 dni)
A ile w przypadku algortymu mergeSort?
(załóżmy stałą multiplikatywną 1)
$2.65 * 10^9 p / 10^9 s$ (2.65 sekundy) :)

Listy dowiązaniowe

Negatywną cechą przedstawionej wersji algorytmu mergeSort było to, że wywołania funkcji `merge` musza zużywać łącznie dużo pamięci na tablice pomocnicze przechowujące wyniki (posortowane ciągi wynikowe).

Można tego uniknąć jeśli ciągi przechowywane są za pomocą innej struktury danych:
List dowiązaniowych

Listy dowiązaniowe składają się z **węzłów** i **dowiązań** (w programowaniu: wskaźników) które łączą węzły.
Każdy węzeł jest pewnym kontenerem i przechowuje jeden element (np. element ciągu). Powiązane stanowią ciąg. np.:

```
początek -> (2)-> (3)-> (5)-> (8)-> null
```

Listy dowiązaniowe

Jest wiele wariantów list dowiązaniowych np.:

- jednokierunkowe (każdy węzeł zawiera wskaźnik do następnego węzła, dodatkowy węzeł pokazuje na początek listy, ostatni węzeł pokazuje na `null`. W takiej liście można poruszać się tylko do przodu)
- dwukierunkowe (każdy węzeł zawiera wskaźniki do następnika i poprzednika w liście, możemy mieć dwa oddzielne wskaźniki na początek i koniec. Struktura zużywa 2 razy więcej pamięci na wskaźniki, ale można poruszać się po niej zarówno do przodu jak i do tyłu)
- cykliczne (jedno lub dwukierunkowe): ostatni węzeł powiązany jest (cyklicznie) z pierwszym

W przypadku implementacji algorytmu mergeSort wystarczają listy jednokierkunkowe, gdyż funkcja `merge` przechodzi każdą listę tylko w jednym kierunku.

Listy kontra tablice

tablice:

- zalety: bardzo szybki dostęp bezpośredni do dowolnego elementu, minimalne zużycie pamięci
- wady: wstawienie dowolnego elementu w środek ma liniowy złożoność czasową (przesuwanie innych elementów)

listy dowiązaniowe:

- zalety: wstawienie/usunięcie dowolnie długiej podlisty ma **stałą** złożoność czasową, niezależnie od długości podlisty (wystarczy przestawić stałą liczbę dowiązań)
- wady: wolny dostęp do elementów (liniowy) zależący od miejsca w liście, dodatkowa pamięć na dowiązania (wskaźniki)

Listy dowiązaniowe w mergeSort

Jeśli ciąg wejściowy do posortowania przechowywany jest w formie listy dowiązaniowej, można zaimplementować algorytm mergeSort tak, że ma taką samą złożoność czasową (liniowo-logarytmiczną) natomiast **nie zużywa wogóle dodatkowej pamięci** wewnątrz funkcji merge (gdyż funkcja merge jedynie przestawia dowiązania aby połączyć ciągi), jedynie zużywa pamięć na obsługę rekursji (ramki stosu wywołań funkcji).

Problemy/pytania:

- istota problemu sortowania i zastosowania
- selectionSort (pomysł, działanie, kod, analiza)
- Insertion Sort (j.w.)
- Merge Sort (j.w.)
- listy kontra tablice (wady i zalety)
- napisz funkcję merge używając list a nie tablic

Dziękuję za uwagę!