Metoda Dziel i Zwyciężaj.

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

1 Schemat ogólny.

Algorytmy skonstruowane metodą "dziel i zwyciężaj" składają się z trzech zasadniczych kroków:

1. transformacja danych x na dane $x_1, \ldots, x_k$ o mniejszym rozmiarze;
2. rozwiązanie problemu dla danych x_i $(i = 1, \ldots, k)$;
3. obliczenie rozwiązania dla danych x na podstawie wyników otrzymanych w punkcie 2.

W naturalny sposób implementowane są jako procedury rekurencyjne:

```
function DiZ(x)
    if x jest małe lub proste then return AdHoc(x)
    przekształć x na x_1, ..., x_k o mniejszym rozmiarze niż x
    for i ← 1 to k do y_i ← DiZ(x_i)
    na podstawie y_1 ... y_k oblicz rozwiązanie y dla x
    return y
```

gdzie $AdHoc$ jest algorytmem używanym do rozwiązania problemu dla "łatwych" danych.

2 Ważne równanie rekurencyjne.

Twierdzenie 1 *Niech $a, b, c \in \mathcal{N}$. Rozwiązaniem równania rekurencyjnego*

$$T(n) = \begin{cases} b & \text{dla } n = 1 \\ aT(n/c) + bn & \text{dla } n > 1 \end{cases}$$

dla n będących potęgą liczby c jest

$$T(n) = \begin{cases} \Theta(n) & \text{jeżeli } a < c, \\ \Theta(n \log n) & \text{jeżeli } a = c, \\ \Theta(n^{\log_c a}) & \text{jeżeli } a > c \end{cases}$$

Dowód. Niech: $n = c^k$, czyli $k = \log_c n$. Stosując metodę podstawiania otrzymujemy:

$$T(n) = a^k bn/c^k + a^{k-1} bn/c^{k-1} + \ldots + abn/c + bn = bn \sum_{i=0}^{k} \left(\frac{a}{c}\right)^i.$$

Rozważamy 3 przypadki:

1. $a < c$
   Wówczas $\frac{a}{c} < 1$, więc szereg $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i$ jest zbieżny do pewnego $m \in \mathcal{R}^+$. Stąd $T(n) = bmn$.
2. $a = c$
   Wówczas $\frac{a}{c} = 1$, więc $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i = k + 1 = \Theta(\log_c n)$. Stąd $T(n) = \Theta(n \log_c n)$.

3. $a > c$
   Wówczas $\frac{a}{c} > 1$, więc:

$$T(n) = bc^k \sum_{i=0}^{k} \left(\frac{a}{c}\right) = bc^k \frac{\left(\frac{a}{c}\right)^{k+1} - 1}{\left(\frac{a}{c}\right) - 1} =$$

$$b\frac{a^{k+1} - c^{k+1}}{a - c} = \frac{ba}{a-c} a^{\log_c n} - \frac{cb}{a-c} n = \frac{ba}{a-c} a^{\log_c n} - O(n).$$

   Ponieważ n jest $O(a^{\log_c n}) = O(n^{\log_c a})$, więc $T(n) = \Theta(a^{\log_c n})$.

□

Interpretacja: Twierdzenie określa złożoność algorytmów opartych na strategii dziel i zwyciężaj, które:

- redukują problem dla danych o rozmiarze n do rozwiązania tego problemu dla a zestawów danych, każdy o rozmiarze n/c.
- wykonują kroki 1 i 3 schematu ogólnego w czasie liniowym.

3 Przykłady.

3.1 Sortowanie

Nasze przykłady rozpoczniemy od zaprezentowania dwóch strategii Dziel i Zwyciężaj dla problemu sortowania.

Problem:

Dane: tablica $T[1..n]$ elementów z uporządkowanego liniowo uniwersum

Zadanie: uporządkować T

3.1.1 Strategia 1: Sortowanie przez scalanie.

Strategia ta oparta jest na tym, że dwa uporządkowane ciągi potrafimy szybko (w czasie liniowym) scalić w jeden ciąg. Aby posortować tablicę wystarczy więc podzielić ją na dwie części, niezależnie posortować każdą z części a następnie scalić je.

```
procedure mergesort(T[1..n])
  if n jest małe then insert(T)
  else
       X[1 .. ⌈n/2⌉] ← T[1 .. ⌈n/2⌉]
       Y[1 .. ⌊n/2⌋] ← T[1 + ⌈n/2⌉ .. n]
       mergesort(X); mergesort(Y)
       T ← merge(X, Y)
```

Czas działania algorytmu wyraża się równaniem $t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \Theta(n)$, którego rozwiązaniem jest $t(n) = \Theta(n \log n)$. Jak później pokażemy jest to asymptotycznie optymalny czas działania algorytmów sortowania.

Mankamentem tego algorytmu jest fakt wykorzystywania dodatkowych tablic (poza tablicą wejściową) podczas scalania ciągów. Niestety nie jest znany sposób usunięcia tej wady. Co prawda znane są metody "scalania w miejscu" w czasie liniowym, lecz są one bardzo skomplikowane, co sprawia, że stałe w funkcjach liniowych ograniczających czas działania są nieakceptowalnie wielkie.

Przy okazji prezentacji tego algorytmu chcemy zwrócić uwagę na niezwykle ważny, a często zaniedbywany, aspekt implementacji algorymów typu Dziel i Zwyciężaj: staranne dobranie progu na rozmiar danych, poniżej którego nie opłaca się stosować algorytmu rekurencyjnie. Przykładowo powyżej zastosowaliśmy dla małych danych algorytm *insert*. Może on wymagać czascu kwadratowego, ale jest bardzo prosty i łatwy w implementacji, dzięki czemu w praktyce jest on dla małych danych szybszy od rekurencyjnej implementacji sortowania przez scalanie. Teoretyczne wyliczenie wartości progu jest zwykle trudne i zawodne. Jego wartość zależy bowiem także od efektywności implementacji a nawet od typu maszyny, na którym program będzie wykonywany. Dlatego, jeśli zależy nam na optymalnym "dostrojeniu" programu (na przykład z tego powodu, że jest on bardzo często wykonywaną procedurą, ważącą na efektywności całego systemu), powinniśmy wyznaczyć ten próg poprzez starannie dobrane eksperymenty.

3.1.2 Strategia 2: Quicksort

Najistotniejszym krokiem algorytmu sortowania przez scalanie jest krok 3 (łączenie wyników podproblemów). Natomiast krok 1 jest trywialny i sprowadza się do wyliczenia indeksu środka tablicy (ze względów technicznych połączyliśmy go powyżej z kopiowaniem elementów do tablic roboczych).

W algorytmie *Quicksort* sytuacja jest odwrotna: istotnym krokiem jest krok 1. Polega on na podziale elementów tablicy na dwa ciągi, takie, że każdy element pierwszego z nich jest nie mniejszy od każdego elementu drugiego z nich. Jeśli teraz każdy z tych ciągów zostanie niezależnie posortowany, to krok 3 staje się zbyteczny.

procedure Quicksort($T[1..n]$)
  if n jest małe **then** insert(T)
  else
    wybierz element dzielący x
    (* niech k równa się liczbie elementów tablicy T nie większych od x*)
    przestaw elementy tablicy T tak, że $\forall_{i\leq k} T[i] \leq x$
    Quicksort($T[1..k]$); Quicksort($T[(k+1)..n]$);

Analizie złożoności algorytmu *Quicksort* poświęcimy oddzielny wykład. Wówczas omówimy także sposoby implementacji kroku 1.

3.2 Mnożenie bardzo dużych liczb.

Problem:
Dane: liczby naturalne a i b
komentarz: liczby a i b są długie.
Wynik: iloczyn $a \cdot b$

Dla prostoty opisu przyjmijmy, że obydwie liczby mają tę samą długość (równą $n = 2^k$). Narzucający się algorytm oparty na strategii Dziel i Zwyciężaj polega na podziale n-bitowych czynników na części $n/2$-bitowe, a następnie odpowiednim wymnożeniu tych części.

Niech $a = a_1 \cdot 2^s + a_0$ i $b = b_1 \cdot 2^s + b_0$, gdzie $s = n/2$; $0 \leq a_1, a_0, b_1, b_0 < 2^s$. Iloczyn $a \cdot b$ możemy teraz zapisać jako

$$ab = c_2 \cdot 2^{2s} + c_1 \cdot 2^s + c_0,$$

gdzie $c_2 = a_1 b_1$; $c_1 = a_0 b_1 + a_1 b_0$; $c_0 = a_0 b_0$.

Jak widać jedno mnożenie liczb n-bitowych można zredukować do czterech mnożeń liczb $n/2$-bitowych, dwóch mnożeń przez potęgę liczby 2 i trzech dodawań. Zarówno dodawania jak i mnożenia przez potęgę liczby 2 można wykonać w czasie liniowym. Taka redukcja nie prowadzi jednak do szybszego algorytmu - czas działania wyraża się wzorem $T(n) = 4T(n/2) + \Theta(n)$, którego rozwiązaniem jest $T(n) = \Theta(n^2)$. Aby uzyskać szybszy algorytm musimy potrafić obliczać współczynniki c_2, c_1, c_0

przy użyciu trzech mnożeń liczb $n/2$-bitowych. Uzyskujemy to przez zastąpienie dwóch mnożeń podczas obliczania c_1 jednym mnożeniem i dwoma odejmowaniami:

$$c_1 = (a_1 + a_0)(b_1 + b_0) - c_0 - c_2.$$

```
multiply(a, b)
    n ← max(|a|, |b|)   (* |x| oznacza długość liczby x *)
    if n jest małe then pomnóż a i b klasycznym algorytmem
                        return obliczony iloczyn
    p ← ⌊n/2⌋
    a1 ← ⌊a/2^p⌋; a0 ← a mod 2^p
    b1 ← ⌊b/2^p⌋; b0 ← b mod 2^p
    z ← multiply(a0, b0)
    y ← multiply(a1 + a0, b1 + b0)
    x ← multiply(a1, b1);
    return 2^(2p) x + 2^p (y − x − z) + z
```

Fakt 1 *Złożoność czasowa powyższego algorytmu wynosi $O(n^{\log 3})$.*

Dowód: (Zakładamy, że a i b są liczbami n - bitowymi i n jest potęgą liczby 2.)
Wystarczy pokazać, że czas działania algorytmu wyraża się wzorem:

$$T(n) = \begin{cases} k & \text{dla } n = 1 \\ 3T(n/2) + \Theta(n) & \text{dla } n > 1 \end{cases}$$

Jedyną wątpliwość może budzić fakt, że, wskutek przeniesienia, liczby $a_1 + a_0$ i $b_1 + b_0$ mogą być $(n/2) + 1$-bitowe. W takiej sytuacji $a_1 + a_0$ zapisujemy w postaci $a'2^{n/2} + a''$, a $b_1 + b_0$ w postaci $b'2^{n/2} + b''$, gdzie a' i b' są bitami z pozycji $(n/2) + 1$ liczb a i b, a a'' i b'' są złożone z pozostałych bitów. Obliczenie y możemy teraz przedstawić jako:

$$(a_1 + a_0)(b_1 + b_0) = a'b'2^n + (a'b'' + a''b')2^{n/2} + a''b''$$

Jedynym składnikiem wymagającym rekurencyjnego wywołania jest $a''b''$ (obydwa czynniki są $n/2$-bitowe). Pozostałe mnożenia wykonujemy w czasie $O(n)$, ponieważ jednym z czynników jest pojedynczy bit (a' lub b') lub potęga liczby 2.
Tak więc przypadek, gdy podczas obliczania y występuje przeniesienie przy dodawaniu, zwiększa złożoność jedynie o pewną stałą, nie zmieniając klasy złożoności algorytmu. $\square$

3.2.1 Podział na więcej części

Pokażemy teraz, że powyższą metodę można uogólnić. Niech $k \in \mathcal{N}$ będzie dowolną stałą. Liczby a i b przedstawiamy jako

$$a = \sum_{i=0}^{k-1} a_i \cdot 2^{in/k} \qquad b = \sum_{i=0}^{k-1} b_i \cdot 2^{in/k}$$

gdzie wszystkie a_i oraz b_i są liczbami co najwyżej n/k-bitowymi. Naszym zadaniem jest policzenie liczb $c_0, c_1, \ldots, c_{2k}$ takich, że dla $j = 0, \ldots, 2k-2$:

$$c_j = \sum_{r=0}^{j} a_r b_{j-r}.$$

Niech liczby $w_1, \ldots, w_{2k-1}$ będą zdefiniowane w następujący sposób:

$$w_t = (\sum_{i=0}^{k-1} a_i t^i) \cdot (\sum_{i=0}^{k-1} b_i t^i).$$

Łatwo sprawdzić, że $w_t = \sum_{j=0}^{2k-2} c_j t^j$. Otrzymaliśmy więc układ $2k-1$ równań z $2k-1$ niewiadomymi $c_0, c_1, \ldots, c_{2k-2}$. Fakt ten możemy zapisać jako

$$U \cdot [c_0, c_1, \ldots, c_{2k-3}, c_{2k-2}]^T = [w_1, w_2, \ldots, w_{2k-2}, w_{2k-1}]^T,$$

gdzie elementy macierzy $U = \{u_{ij}\}$ są równe $u_{ij} = i^j$ (dla $i = 1, \ldots, 2k-1$ oraz $j = 0, \ldots, 2k-2$). Ponieważ U jest macierzą nieosobliwą (jest macierzą Vandermonde'a), istnieje rozwiązanie powyższego układu. Jeśli $w_1, \ldots, w_{2k-1}$ potraktujemy jako wartości symboliczne, to rozwiązując ten układ wyrazimy c_i jako kombinacje liniowe tych wartości. To stanowi podstawę dla następującego algorytmu:

1. Oblicz rekurencyjnie wartości $w_1, \ldots, w_{2k-1}$.
2. Oblicz wartości $c_0, \ldots, c_{2k-2}$.
3. **return** $\sum_{i=0}^{2k-2} c_i 2^{in/k}$.

Fakt 2 *Powyższy algorytm działa w czasie* $\Theta(n^{\log_k(2k-1)})$.

Pomijamy formalny dowód tego faktu. Wynika on z tego, że w kroku 1 wywołujemy $2k-1$ razy rekurencyjnie funkcję dla danych o rozmiarze n/k oraz z tego, że kroki 2 i 3 wykonują się w czasie liniowym.

Jakkolwiek zwiększając k możemy z wykładnikiem nad n dowolnie blisko przybliżyć się do 1, to jednak zauważmy, że otrzymane algorytmy mają znaczenie czysto teoretyczne. Stałe występujące w kombinacjach obliczanych w punkcie 2 już dla niewielkich wartości k są bardzo duże i sprawiają, że algorytm działa szybciej od klasycznego mnożenia pisemnego dopiero dla danych o astronomicznie wielkich rozmiarach.

3.3 Równoczesne znajdowanie minimum i maksimum w zbiorze.

PROBLEM: Minmax
Dane: zbiór $S = \{a_1, a_2, \ldots, a_n\}$ [1]
Wynik: $min\{a_i \mid i = 1, \ldots, n\}$ $max\{a_i \mid i = 1, \ldots, n\}$

KOMENTARZ: Ograniczamy się do klasy algorytmów, które danych wejściowych używają jedynie w operacjach porównania. Interesują nas dwa zagadnienia:

- znalezienie algorytmu z tej klasy, który rozwiązuje problem Minmax, używając jak najmniejszej liczby porównań.
- dokładne wyznaczenie dolnej granicy na liczbę porównań (tym problemem zajmiemy się na wykładzie poświęconym dolnym granicom).

Proste podejście do tego problemu polega na tym, by szukane liczby znaleźć niezależnie, np. najpierw minimum a potem maksimum. Takie rozwiązanie wymaga $2n-2$ porównań. Jego nieoptymalność wynika z tego, że algorytm podczas szukania maksimum nie wykorzystuje w żaden sposób informacji jakie nabył o elementach zbioru S w czasie wyszukiwania minimum. W szczególności w trakcie szukania maksimum będą brały udział w porównaniach te elementy S-a, które podczas szukania minimum były porównywane z większymi od siebie elementami, a więc nie mogą być maksimum. To spostrzeżenie prowadzi do następującego algorytmu:

[1] Termin "zbiór" jest użyty tu w dość swobodnym znaczeniu. W zasadzie S jest multizbiorem - elementy mogą się w nim powtarzać. O ile jednak nie będzie to źródłem dwuznaczności, w przyszłości termin "zbiór" będziemy stosować także w odniesieniu do multizbiorów.

```
MinMax1(S)
    S_m ← S_M ← ∅
    for i = 1 to n div 2 do
        porównaj a_i z a_{n-i+1}; mniejszą z tych liczb wstaw do zbioru S_m, a większą - do zbioru S_M
    m ← min{a | a ∈ S_m}
    M ← max{a | a ∈ S_M}
    if n parzyste then return (m, M)
                  else return (min(m, a_{⌈n/2⌉}), max(M, a_{⌈n/2⌉}))
```

Fakt 3 *Algorytm MinMax1 wykonuje* $\lceil \frac{3}{2}n - 2 \rceil$ *porównań na elementach zbioru* S.

DOWÓD. Niech $n = 2m$. W pętli **for** wykonywanych jest m porównań, a w dwóch następnych wierszach po $m-1$ porównań. W sumie mamy $3m - 2 = \lceil \frac{3}{2}n - 2 \rceil$ porównań.

Gdy $n = 2m+1$, algorytm najpierw znajduje minimum i maksimum w zbiorze $S \setminus \{a_{\lceil n/2 \rceil}\}$. Zbiór ten ma $2m$ elementów, a więc liczba wykonanych porównań wynosi $3m-2$. Ostatnia instrukcja wymaga dwóch porównań, co w sumie daje $3m$ porównań. Jak łatwo sprawdzić $3m = 3(n-1)/2 = \lceil 3(n-1)/2 - 1/2 \rceil = \lceil \frac{3}{2}n - 2 \rceil$. □

Inne podejście polega na zastosowaniu strategii Dziel i Zwyciężaj: zbiór S dzielimy na dwie części, w każdej części znajdujemy minimum i maksimum, jako wynik dajemy mniejsze z minimów i większe z maksimów. Poniższa, niestaranna implementacja tego pomysłu nie osiąga jednak liczby porównań algorytmu $MinMax1$ i powinna stanowić ostrzeżenie przed niefrasobliwym implementowaniem niedopracowanych algorytmów. Poprawienie tej implementacji pozostawiamy jako zadanie na ćwiczenia.

```
Procedure MinMax2(S)
    if |S|= 1 then return (a_1, a_1)
    else
        if |S|= 2 then return (max(a_1, a_2), min(a_1, a_2))
        else
            podziel S na dwa równoliczne (z dokładnością do jednego elementu) podzbiory S_1, S_2
            (max1, min1) ← MinMax2(S_1)
            (max2, min2) ← MinMax2(S_2)
            return (max(max1, max2), min(min1, min2))
```