ASD—1

1 Podstawowe zasady analizy algorytmów

Analiza algorytmów jest działem informatyki szukającym najlepszych algorytmów dla zadań komputerowych. Analiza algorytmów stara się między innymi znaleźć odpowiedzi na następujące pytania:

1. czy dany problem może być rozwiązany na komputerze w dostępnym czasie i pamięci;
2. który ze znanych algorytmów należy zastosować w danych okolicznościach;
3. czy istnieje lepszy algorytm od rozważanego algorytmu; czy też jest on optymalny;
4. jak uzasadnić, że podany algorytm rozwiązuje zamierzone zadanie.

Dokonując analizy algorytmu zwracamy uwagę na jego poprawność semantyczną, prostotę, czas działania, ilość zajmowanej pamięci, optymalność, okoliczności w jakich należy go używać, a w jakich nie.

1.1 Złożoność obliczeniowa

Złożoność obliczeniową algorytmu definiuje się jako ilość zasobów komputerowych potrzebnych do jego wykonania. Podstawowymi zasobami rozważanymi w analizie algorytmów są czas działania i ilość zajmowanej pamięci.

Zauważmy, że nie jest na ogół możliwe wyznaczenie złożoności obliczeniowej jako funkcji danych wejściowych (takich jak ciągi, tablice, drzewa czy grafy). Zwykle z zestawem danych wejściowych w sposób naturalny związany jest jego *rozmiar* rozumiany, mówiąc ogólnie, jako liczba pojedynczych danych wchodzących w jego skład.

Na przykład, w problemie sortowania za rozmiar przyjmuje się zwykle liczbę elementów w ciągu wejściowym; w problemie obchodzenia drzewa liczbę węzłów w drzewie; w problemie wyznaczenia wartości wielomianu stopień wielomianu. Rozmiar zestawu danych d będzie zwykle oznaczany przez $|d|$.

Aby móc wyznaczać złożoność obliczeniową algorytmu musimy się jeszcze umówić w jakich jednostkach będziemy ją liczyć. W przypadku *złożoności pamięciowej* za jednostkę przyjmuje się zwykle słowo pamięci maszyny. Sytuacja jest nieco bardziej skomplikowana w przypadku *złożoności czasowej*. Złożoność czasowa powinna być własnością samego tylko algorytmu, jako metody rozwiązania problemu, niezależnie od komputera, języka programowania, czy sposobu jego zakodowania. W tym celu wyróżnia się w algorytmie charakterystyczne dla niego operacje, nazywane operacjami dominującymi, takie, że ich łączna liczba jest proporcjonalna do liczby wykonań wszystkich operacji jednostkowych w dowolnej komputerowej realizacji algorytmu.

Na przykład, dla algorytmów sortujących, za operację dominującą przyjmuje się zwykle porównanie dwóch elementów w ciągu wejściowym, czasem także przestawienie elementów w ciągu; dla algorytmow przeglądających drzewo przejście krawędzi pomiędzy węzłami w drzewie; dla algorytmów wyznaczających wartość wielomianu operacje arytmetyczne $+$, $-$, $*$ i $/$.

Za *jednostkę złożoności* czasowej przyjmuje się wykonanie jednej operacji dominującej.

Zlożoność obliczeniową algorytmu traktuje się jako funkcję rozmiaru danych n. Wyróżnia się:

1. *złożoność pesymistyczną* - definiowaną jako ilość zasobów komputerowych potrzebnych przy "najgorszych" danych wejściowych rozmiaru n oraz

2. złożoność oczekiwaną - definiowaną jako ilość zasobów komputerowych potrzebnych przy "typowych" danych wejściowych rozmiaru n.

Żeby te pojęcia zdefiniować precyzyjnie wprowadźmy następujące oznaczenia:
D_n — zbiór zestawów wszystkich danych wejściowych rozmiaru n,
$t(d)$ — liczba operacji dominujących dla zestawu danych wejściowych d,
X_n — zmienna losowa, której wartością jest $t(d)$, dla $d \in D_n$,
p_{nk} — rozkład prawdopodobieństwa zmiennej losowej X_n, tj. prawdopodobieństwo, że dla danych rozmiaru n algorytm wykona k operacji dominujących ($k \geq 0$).

Przez *pesymistyczną* złożoność czasową algorytmu rozumie się funkcję:

$$W(n) = \sup\{t(d) : d \in D_n\}$$

a przez *oczekiwaną złożoność czasową* algorytmu funkcję:

$$A(n) = \sum_{k \geq 0} k p_{nk}$$

tj. wartość oczekiwaną $\mathrm{ave}(X_n)$ zmiennej losowej X_n.

Aby stwierdzić, na ile funkcje $W(n)$ i $A(n)$ są reprezentatywne dla wszystkich danych wejściowych rozmiaru n rozważa się miary wrażliwości algorytmu:

miarę pesymistycznej wrażliwości

$$\Delta(n) = \sup\{t(d_1) - t(d_2) : d_1, d_2 \in D_n\}$$

oraz

miarę oczekiwanej wrażliwości

$$\delta(n) = \mathrm{dev}(X_n)$$

gdzie $\mathrm{dev}(X_n)$ jest standardowym odchyleniem zmiennej losowej X_n tj. $\mathrm{dev}(X_n) = \sqrt{\mathrm{var}(X_n)}$ i $\mathrm{var}(X_n) = \sum_{k \geq 0}(k - \mathrm{ave}(X_n))^2 p_{nk}$ jest wariancją zmiennej losowej X_n.

Przykład 1 Przeszukiwanie sekwencyjne ciągu

Dane wejściowe: L, N, a – gdzie N jest liczbą naturalną, $N \geq 0$, a jest poszukiwanym elementem, $L[0..N]$ jest tablicą, w której na miejscach od 0 do $N-1$ znajdują się elementy ciągu.

Wynik: Jeśli $a \in L[0..N-1]$ to $szukaj = i$ takie, że $0 \leq i \leq N-1$ i $L[i] = a$. W przeciwnym razie $szukaj = N$.

```
int szukaj(int *L, int N, int a)
{
```

```
  int i;

  L[N] = a; i = 0;
  while (L[i] != a)
    i++;

  return i;
}
```

- Rozmiar danych wejściowych: $n = N$
- Operacja dominująca: porównanie $L[i] \neq a$
- Pesymistyczna złożoność czasowa: $W(n) = n + 1$
- Pesymistyczna wrażliwość czasowa: $\Delta(n) = n$
- Oczekiwana złożoność czasowa: $A(n) = (n + 1)/2$
- Oczekiwana wrażliwość czasowa" $\delta(n) \approx .29n$

Faktyczna złożoność czasowa algorytmu (czas działania) w chwili jego użycia jako programu różni się od wyliczonej teoretycznie współczynnikiem proporcjonalności, który zależy od konkretnej jego realizacji. Zatem istotną częścią informacji, która jest zawarta w funkcjach złożoności $W(n)$ i $A(n)$ jest ich rząd wielkości, czyli ich zachowanie asymptotyczne, gdy n dąży do nieskończoności. Zwykle staramy się podać jak najprostszą funkcję charakteryzującą rząd wielkości $W(n)$ i $A(n)$, np. n, $n \log n$, n^2 , n^3.

Używamy w tym celu następujących oznaczeń dla rzędów wielkości funkcji. Niech $f, g, h : N \to R_+ \cup \{0\}$.

Mówimy, że f jest co najwyżej rzędu g, oznaczenie $f(n) = O(g(n))$, jeśli istnieją stałe rzeczywista $c > 0$ i naturalna n_0 takie, że nierówność $f(n) \leq cg(n)$ zachodzi dla każdego $n \geq n_0$.

Na przyklad, $n^2 + 2n = O(n^2)$, bo $n^2 + 2n \leq 3n^2$, dla każdego naturalnego n.

Mówimy, że f jest co najmniej rzędu g, oznaczenie $f(n) = \Omega(g(n))$, jeśli $g(n) = O(f(n))$.

Mówimy, że f jest dokładnie rzędu g, oznaczenie $f(n) = \Theta(g(n))$, jeśli zarówno $f(n) = O(g(n))$, jak i $f(n) = \Omega(g(n))$.

Większość rozważanych algorytmów ma złożoność czasową rzędu jednej z następujących funkcji:

1. $\log n$ — złożoność logarytmiczna
2. n — złożoność liniowa
3. $n \log n$ - złożoność $n \log n$
4. n^2 — zlożoność kwadratowa
5. n^3, n^4, ... — następne złożoności wielomianowe
6. $n^{\log n}$ — złożoność podwykładnicza
7. 2^n — złżoność wykładnicza 2^n
8. $n!$ — złżoność wykładnicza $n!$

Zauważmy, że algorytm o złożoności wykładniczej może być zrealizowany jedynie dla małych rozmiarów danych. Istnieje próg, od którego funkcja wykładnicza zaczyna rosnąć tak szybko, że realizacja algorytmu na komputerze staje się niemożliwa. Mianowicie załóżmy przykładowo, że dla danych rozmiaru n algorytm wykonuje 2^n operacji jednostkowych oraz że każda operacja jednostkowa zajmuje czas odpowiednio 10^{-6} i 10^{-9} sekund na dwóch różnych komputerach. Czas działania potrzebny do realizacji algorytmu przedstawia tabelka.

Rozmiar n	20	50	100	200
Czas działania ($2^n/10^6$)	1.04 sek	35.7 lat	$4 * 10^{14}$ wieków	$5 * 10^{44}$ wieków
Czas działania ($2^n/10^9$)	0.001 sek	13 dni	$4 * 10^{11}$ wieków	$5 * 10^{41}$ wieków

Widać, że nawet 1000-krotne przyspieszenie szybkości działania komputera niewiele pomaga algorytmowi wykładniczemu. Nierealizowalność uważa się za wewnętrzną cechę algorytmu o złożoności wykładniczej. Aby jednak mieć pełny obraz sytuacji powinniśmy jeszcze rozważyć wrażliwość algorytmu na dane wejściowe. Może się zdarzyć, że dla danego algorytmu $W(n) = 2^n + O(1)$, ale także $D(n) = 2^n + O(1)$. Wówczas nie możemy twierdzić, że algorytm jest nierealizowalny dla reprezentatywnych danych. Dane wejściowe, dla których czas działania jest wykładniczy mogą się nigdy nie pojawić w rzeczywistych okolicznościach!

Przy korzystaniu z wyników analizy złożoności algorytmu należy zatem brać pod uwagę następujące uwarunkowania:

1. algorytm i jego realizacja przeznaczona do wykonania są zwykle wyrażone w dwóch całkowicie róńych językach;
2. wrażliwość algorytmu na dane wejściowe może spowodować, że faktyczne zachowanie się algorytmu na

używanych danych będzie odbiegać od zachowania opisanego funkcjami złożoności $W(n)$ i $A(n)$;

3. może być trudno przewidzieć rzeczywisty rozkład prawdopodobieństwa zmiennej losowej X_n;

4. dla niektórych algorytmów nie są znane matematyczne oszacowania wielkości $W(n)$ i $A(n)$; szczególnie wyznaczenie $A(n)$ dla rzeczywistego rozkładu prawdopodobieństwa może stanowić bardzo trudny problem matematyczny;

5. czasami działanie dwóch algorytmów trudno jest jednoznacznie porównać; jeden działa lepiej dla pewnej klasy zestawów danych, drugi dla innych.

Ważną cechą algorytmu jest jego prostota, z której zwykle wynika mniejszy współczynnik proporcjonalności przy złożoności obliczeniowej oraz łatwość realizacji (zaprogramowania). Szczególnie więc w dwóch przypadkach:

1. program używający nasz algorytm ma być wykonany raz lub tylko kilka razy;

2. algorytm ma być stosowany tylko dla małych rozmiarów danych,

należy wybierać algorytm raczej pod kątem jego prostoty niż małej złożoności obliczeniowej.

2 Sortowanie

Sortowanie jest problemem, który jest bardzo często rozwiązywany na komputerach. Jego popularność wiąże się z faktem, że łatwiej jest korzystać ze zbiorów uporządkowanych niż nieuporządkowanych.

Sortowanie definiuje się następująco: Dany jest ciąg $q = [a_0, a_1, ..., a_{n-1}]$ elementów zbioru liniowo uporządkowanego. Dokonać permutacji ustawiającej elementy w porządku niemalejącym

$$a_{i_0} \leq a_{i_1} \leq \ldots \leq a_{i_{n-1}}.$$

Za operację dominującą przyjmować będziemy porównania elementów w ciągu. Za złożoność pamięciową $S(n)$ przyjmować będziemy ilość dodatkowej pamięci (oprócz n miejsc pamięci dla elementów w ciągu) potrzebnej do wykonania algorytmu.

Dla zapisu algorytmów przyjmować będziemy założenie, że elementy ciągu q są liczbami całkowitymi (typu long) i że znajdują się w tablicy a, $a[i] = a_i$, dla $0 \leq i < n$. Przy analizie probabilistycznej (wielkości $A(n)$ i $\mathrm{dev}(n)$) będziemy przyjmować, że danymi wejściowymi są permutacje liczb $1, 2, \ldots, n$ oraz, że każda taka permutacja jest jednakowo prawdopodobna.

2.1 SelectionSort - sortowanie przez selekcję

```
void SelectionSort(long *a, int n)
{
  int i, j, min;
```

```
  long x;

  for ( i = 0; i < n; i++){
    min = i;
    for (j = i+1; j < n; j++)
      if (a[j] < a[min] )
        min = j;

     x = a[min];
     a[min] = a[i];
     a[i] = x;
   }
}
```

Analiza złożoności algorytmu SelectionSort jest bezpośrednia: $W(n) = A(n) = (n-1)+(n-2)+\ldots+1 = n(n-1)/2 = \frac{1}{2}n^2 + O(1)$.
$\Delta(n) = \delta(n) = 0$.
$S(n) = O(1)$ (mówimy, że algorytm sortuje w miejscu).

Do głównych zalet przedstawionego algorytmu należą:

1. optymalność, jeśli chodzi o liczbę przestawień – tylko $n-1$;
2. prosty do uruchomienia;
3. zadowalająco szybki dla małych wartości n.

2.2 InsertionSort - sortowanie przez wstawianie

```
void InsertionSort(long *a, int n)
{
```

```
  int i, j;
  long x;

  for ( i = 1; i < n; i++){
    j = i;
    x = a[i];
    while ( (j > 0) && (a[j-1] > x) ){
      a[j] = a[j-1];
      j--;
    }
    a[j] = x;
  }
}
```

Oto wyniki analizy algorytmu InsertionSort:
$W(n) = \frac{1}{2}n^2 + O(n)$,
$\Delta(n) = \frac{1}{2}n^2 + O(1)$,
$S(n) = O(1)$,
$A(n) = \frac{1}{4}n^2 + O(n)$,
$\delta(n) = \frac{1}{6}n^{3/2} + O(n)$.
Oczekiwana zlożoność czasowa InsertionSort jest więc nieco lepsza niż SelectionSort, chociaż tego samego rzędu wielkości.

2.3 QuickSort - szybkie sortowanie (Hoare 1960)

QuickSort jest jednym z najczęściej używanych algorytmów sortowania. Jest uważany za najszybszy algorytm sortowania dla "losowych" danych wejściowych. Jego konstrukcja jest oparta na zasadzie "dziel-i-zwyciężaj". Wejściowy ciąg q zostaje w specjalny sposób podzielony na dwie części, po czym obie części są niezależnie sortowane.

Oto ogólny schemat algorytmu QuickSort:

```
void QuickSort(long *a, int l, int r)
/* sortujemy a[l..r]; l<r */
{
  int j;

  j = partition(l,r);
  if (j-1 > l)
    QuickSort(l,j-1);
  if (r > j+1)
    QuickSort(j+1,r);
}
```

Decydujące znaczenie dla poprawności i efektywności algorytmu ma funkcja `partition`, która przekształca tablicę `a[l..r]` w ten sposób, że:

1. element $v = a[j]$ znajduje się na swoim właściwym, ostatecznym miejscu w tablicy;
2. $a[l], ..., a[j-1] \leq v$;
3. $v \geq a[j+1], ..., a[r]$.

Jako element v rozdzielający ciag $a[l..r]$ wybieramy $v = a[l]$. Następnie przeglądamy ciąg $a[l+1], ..., a[r]$ od lewej strony, dopóki nie znajdziemy elementu nie mniejszego od $a[l]$, po czym przeglądamy $a[l+1], ..., a[r]$, tym razem od strony prawej, dopóki nie napotkamy na element nie większy niż $a[l]$. Dwa elementy, na których przeglądania się zatrzymały są w niewłaściwej kolejności, zamieniamy je miejscami. Następnie powtarzając opisane postępowanie zapewniamy, że elementy na lewo od wskaźnika wędrującego ze strony lewej na prawo, są nie większe niż $a[l]$, elementy na prawo od wskaźnika wędrującego ze strony prawej na lewo, są nie mniejsze niż $a[l]$. Kiedy oba wskaźniki się spotkają, proces dzielenia jest zakończony - wystarczy element rozdzielający $v = a[l]$ zamienić z ostatnim elementem lewej części.

Poniżej podajemy kompletną postać funkcji QuickSort.

```
void QuickSort(long *a, int l, int r)
{
   int i,j;
   long v, x;

   /* partition; r>l */

   v = a[l];
   i = l;
   j = r+1;

   do{
     do
       i++;
     while ( (i <= r) && ( a[i] < v) );
     do
       j--;
     while ( a[j] > v );
```

```
    if (i < j){
      x = a[i]; a[i] = a[j]; a[j] = x;
    }
  }while (i < j);
  a[l] = a[j]; a[j] = v;
  /* koniec partition */

  if (j-1 > l)
    QuickSort(a,l,j-1);
  if (r > j+1)
    QuickSort(a,j+1,r);
}
```

W wyniku analizy złożoności algorytmu QuickSort otrzymujemy:
$W(n) = \frac{1}{2}n^2 + O(n)$,
$\Delta(n) = O(n^2)$,
$A(n) \approx 1.4n \log n + O(logn)$,
$\delta(n) \approx 0.68n + O(logn)$.

Zauważmy, że:

1. asymptotyczna oczekiwana złożoność czasowa QuickSort jest $n \log n$, a więc mniej niż dla poprzednich algorytmów;
2. współczynnik przy $n \log n$ jest nieduży, około 1.4;
3. oczekiwana wrażliwość jest niewielka w stosunku do oczekiwanej złożoności, co mówi o silnym skupieniu rzeczywistej liczby porównań wokół wartosci oczekiwanej;

4. liczba innych działań algorytmu, poza porównaniami, nie jest znacząco większa od liczby porównań.

Powyższe fakty tłumaczą, dlaczego wyniki testowań kreują QuickSort na najszybszy algorytm sortowania.

Poważną wadą algorytmu QuickSort jest jego niestabilność.

Pewną wadę QuickSort stanowi też użycie rekursji, która przy wykonywaniu algorytmu tłumaczona jest na operację na stosie. W złożoności pamięciowej musimy uwzględnić pamięć potrzebną na implementację stosu. W przypadku pesymistycznym głębokość rekursji jest $n-1$, zatem $S(n) = O(n)$, czyli istotnie więcej niż poprzednie algorytmy. Na następnym wykładzie pokażemy jak zmniejszyć rozmiar stosu do $O(\log n)$.