Сложеност алгоритама

Проучавање (временске) сложености алгоритама састоји се од бројања рачунских корака (операција) које треба извршити. Неформално, сложеност алгоритама можемо дефинисати као максималан број операција потребних за извршавање алгоритама.

Почетна претпоставка је да су све (основне) операције исте сложености. Прецизније, претпоставка је да се основне операције извршавају за јединично време. Најчешће основне операције су: додела вредности променљивој, поређење две променљиве, аритметичко логичке операције, улазно-излазне операције... Нпр. степеновање не представља основну операцију.

Прецизно понашање алгоритма немогуће је предвидети у већини случајева. На понашање утиче много фактора, па се у обзир узимају само главне карактеристике, а занемарују се детаљи везани за тачну реализацију. Према томе, анализи брзине извршавања алгоритма приступа се уз занемаривање константних фактора (видети примере). Другим речима, занима нас асимптотско понашање сложености. Иако је израчунавање сложености приближно, на основу сложености добијају се значајне информације о алгоритму.

Број операција приликом извршавања алгоритма је у директној зависности од величине (димензије) улаза n. Дакле, анализа сложености алгоритма као резултат треба да да утрошено време у зависности од n. Поставља се питање које n узети као најрепрезентативније. Приступ је да се анализа врши за "најгори" могући случај.

пример 1:

```
s=0
for i in range(1,n):
    s=s+a[i]
avg= s/n
```

Наведеним кôдом се рачуна средња вредност низа чија је дужина n.

Број операција који се извршава у току овог алгоритма је 2n+2 (n пута се повећава вредност променљиве i, n пута се повећава вредност суме s, једна операција за иницијализацију суме s и једна операција за рачунање средње вредности.

Као што смо већ поменули, константе занемарујемо. (оне су обично доста мање од улазних величина)
Примера ради, ако је n довољно велико, да ли ће се извршити n+1 или n операција, неће бити пуно важно. Ако је n мало, сложеност алгоритма је свакако мала.
Слично се и мноэнилац може занемарити уколико је константан.

Коначно, сложеност горњег алгоритма је: n операција. Ову чињеницу записујемо као O(n) и кажемо да је у овом случају алгоритам линеарне сложености.

пример 2:

```
for i in range(0,n-1):
    for j in range(i+1,n):
        if (a[i]<a[j]):
            pom=a[i]
            a[i]=a[j]
            a[j]=pom.
```

Наведени алгоритам врши сортирање низа дужине n.
Тачан број операција приликом извршавања овог алгоритма је

$$\Big(\underbrace{n}_{\text{увећавање } i}+\underbrace{(n-1)+(n-2)+(n-3)+\dots+1}_{\text{увећавање } j \text{ за } i=1,2,\dots,n-1}\Big) * \underbrace{3}_{\text{3 операције размене вредности}} = 3\,\frac{n(n-1)}{2}$$

Занемарујући константе, добијамо да је број операција n^2-n. Како је n много мање од n^2 за велике вредности n, можемо занемарити линеарни фактор. Дакле, у овом случају је сложеност квадратна, тј. $O(n^2)$.

У наставку нешто више о асимптотској ознаци $O(f)$ (велико O од f)

Асимптотска ознака O

деф: Нека $f,g : \mathbb{N}^+ \to \mathbb{N}$. Каже се да је $g(n)=O(f(n))$ ако постоје позитивне константе c и N, такве да за свако $n>N$ важи $g(n) \leq c\,f(n)$.

Ознака $O(f(n))$ се, заправо, односи на класу функција, једнакост $g(n)=O(f(n))$ је уобичајна ознака за $g(n)\in O(f(n))$. Интуитивно, f је "горња граница" за функцију f.
На пример, $5n^2+15=O(n^2)$ (јер је $5n^2+15 \leq 6n^2$ за $n\geq 4$).

У изразу $O(\log n)$ основа логаритма није битна, јер се логаритми за различите основе разликују за мултитпикативну константу:

$$\log_a n = \log_a (b^{\log_b n}) = \log_b n \cdot \underbrace{\log_a b}_{\text{константа}}$$

О-изрази се могу сабирати и множити:

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$
$$O(f(n))\, O(g(n)) = O(f(n) \cdot g(n))$$

Није увек једноставно утврдити сложеност алгоритма. У наставку дате су програмске конструкције и процена временске сложености тих конструкција:

програмска конструкција	сложеност
секвенца наредби S: P Q	$O(S) = O(P) + O(Q)$
условна наредба S: if (uslov) then P; else Q;	$O(S) = \max\{O(P), O(Q)\}$
For петља S: for i=1 to n do P;	$O(S) = n\,O(P)$
While/Repeat петља S: while (uslov) do P;	$O(S) = m\,O(P)$ m-број итерација петље у најгорем могућем случају

У наставку дајемо још неке примере одређивања сложености алгоритама.

пример 3:

```
                        циклус
                        са кораком 2
                        ↓  тј   i=i+2
for i in Range(1:n:2):
        P
```

где је $O(P)=1$ (тј неки програм (или код) сложености 1)

Сложеност претходне петље је $O(n)$.
Пошто се петља извршава за парне вредности бројачке променљиве, тело петље се извршава око $\frac{n}{2}$ пута.
Како се константе занемарују, сложеност је и даље линеарна.

пример 4:

```
for i in Range(1,10):
        P
```

где је $O(P)=1$

Сложеност претходне петље је $O(1)$.
Укупан број извршавања петље је 10, не зависи ни од једног параметра, па се може сматрати малом константом.

пример 5:

```
i=1
while (i<n):
    i = i*2
    print(i)
```

Сложеност претходне петље је $O(\log n)$, јер се вредност променљиве i дуплира у сваком кораку док не пређе n.

пример 6:

```
for i in range(1,n):
    j=1
    while (j<i):
        print(j)
        j=2*j
```

Сложеност овог кода је $O(n \log n)$.
Сложеност унутрашње петље за фиксирано i је $O(\log i)$, па се укупна сложеност може проценити на $\log 1 + \log 2 + \dots + \log n$, што је $O(n \log n)$

пример 7:

```
i=1
while (i*i<n):
    print(i)
    i=i+1
```

Сложеност претходног кода је $O(\sqrt{n})$, јер се петља извршава све док је $i<\sqrt{n}$.

Рекурентне једначине

Сложеност рекурзивних функција често се описује рекурентним једначинама. Уместо прецизног решавања рекурентних једначина, довољно је знати њихово асимптотско понашање. У наставку ће бити дато асимптотско понашање рекурентних једначина које се често појављују као сложеност рекурзивних функција.

1° Проблеми који се своде на проблем чија је димензија за један мања од димензије полазног.

$$T(n) = T(n-1) + f(n), \quad T(0)=c$$

број операција полазног проблема

број операција проблема који је за димензију мањи

додатне операције

$$T(n) = T(n-1) + f(n) = T(n-2) + f(n-1) + f(n) = \dots$$
$$= T(0) + f(1) + f(2) + \dots + f(n)$$
$$= c + \sum_{i=1}^{n} f(i)$$

нпр. - ако је $f(n)=n$

$$\sum_{i=1}^{n} f(i) = 1+2+\dots+n = \frac{n(n-1)}{2}$$

па је сложеност полазног проблема $O(n^2)$

- ако је $f(n)=1$

$$\sum_{i=1}^{n} f(i) = n$$

па је сложеност полазног проблема $O(n)$

- ако је $f(n)=\log n$

$$\sum_{i=1}^{n} f(i) = \log 1 + \log 2 + \dots + \log n < n \log n$$

па је сложеност полазног проблема $O(n \log n)$

2° Проблеми који се своде на два (или више) проблема чија је димензија за један мања од полазног

$$T(n) = 2T(n-1) + f(n), \quad T(0) = c$$

$$\begin{aligned}
T(n) &= 2T(n-1) + f(n) = 2\left(T(n-2) + f(n-1)\right) + f(n) \\
&= 2^2\left(T(n-3) + f(n-2)\right) + 2f(n-1) + f(n) \\
&= \dots = \\
&= 2^n \cdot T(0) + \sum_{k=1}^{n} 2^k f(n-k)
\end{aligned}$$

нпр. ако је $f(n) = n$

$$\begin{aligned}
T(n) &= 2^n + \sum_{k=1}^{n} 2^k (n-k) \\
&= 2^n + n\sum_{k=1}^{n} 2^k - \sum_{k=1}^{n} 2^k k \\
&= 2^n + n(2^{n+1} - 1) - ((n-1)2^{n+1} + 2) \\
&= 2^n + 2^{n+1} - n - 2
\end{aligned}$$

па је у овом случају сложеност полазног проблема $O(2^n)$, односно експоненцијална

3° Проблеми који се своде на један (или више) потпроблема који су знатно мање димензије од полазног (неколико пута мање димензије)

$$T(n) = cT\left(\frac{n}{b}\right) + f(n), \quad T(0) = \text{const.}$$

$$\begin{aligned}
T(n) &= cT\left(\frac{n}{b}\right) + f(n) = c\left(c\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \\
&= \dots \\
&= c^k T\Big(\underbrace{\frac{n}{b^k}}_{=1}\Big) + c^k f\left(\frac{n}{b^k}\right) + \dots + cf\left(\frac{n}{b}\right) + f(n)
\end{aligned}$$

где је $k = \log_b n$

Дакле,

$$T(n) = c^{\log_b n} + \sum_{i=0}^{\log_b n} c^i f\left(\frac{n}{b^i}\right).$$

Нпр. ако је $c=2$, $b=2$ и $f(n)=n$

$$T(n) = 2^{\log_2 n} + \sum_{i=0}^{\log_2 n} 2^i \cdot \frac{n}{2^i}$$

$$= n + \log_2 n \cdot n$$

па је сложеност полазног програма $O(n \log n)$.