АЛГОРИТМЫ И АНАЛИЗ СЛОЖНОСТИ

Лекция 3

Оценка сложности

Необходимость оценок

- Высокая трудоемкость точного измерения
- Достаточность приближенного объема затрат

ФОРМИРОВАНИЕ ОЦЕНОК

- Асимптотические обозначения позволяют показать скорость роста функции трудоемкости, маскируя при этом конкретные коэффициенты
- Такая оценка позволяет определить предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных

Асимптотические оценки

- Θ (тетта)
- O (O большое)
- Ω (Омега)

- O - учебник Бахмана по теории простых чисел (Bachman, 1892)
- Θ, Ω введены Д. Кнутом(Donald Knuth)

Θ-оценка

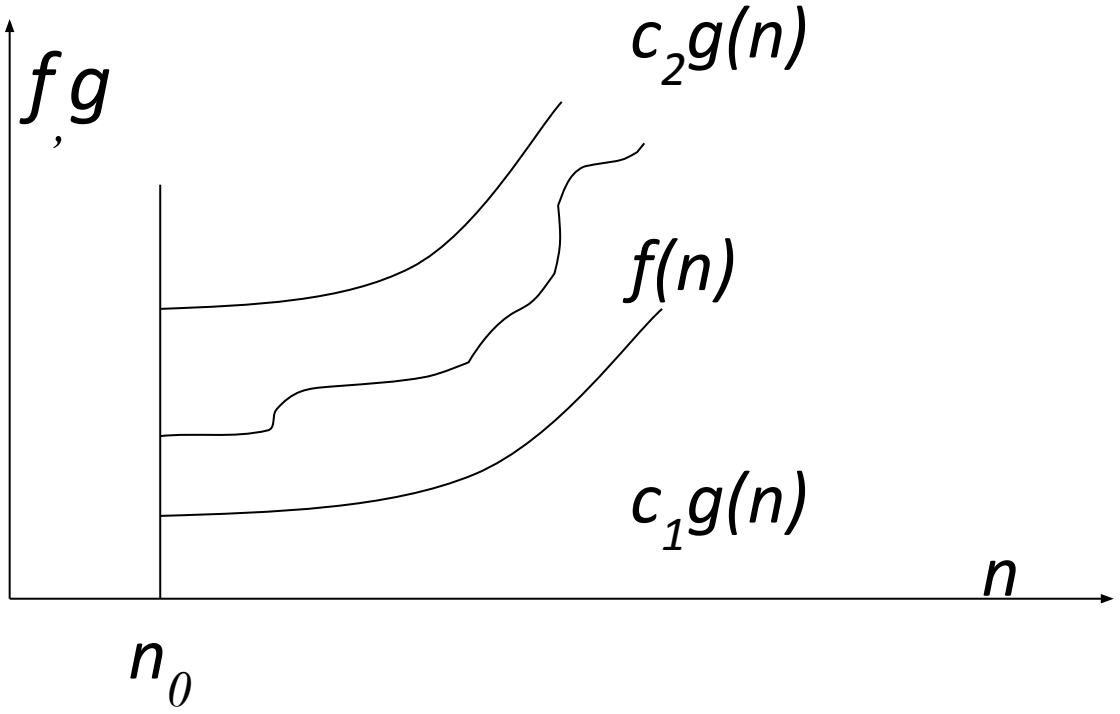
Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента $n \geq 1$, тогда:

$$f(n) = \Theta(g(n)),$$

если существуют положительные c1, c2, n0, такие, что:

$c1 * g(n) \leq f(n) \leq c2 * g(n)$, *при* $n > n0$

Θ-ОЦЕНКА



Θ-оценка

- функция $g(n)$ является *асимптотически точной оценкой* функции $f(n)$, т.к. по определению функция $f(n)$ не отличается от функции $g(n)$ с точностью до постоянного множителя

Θ-оценка

- Замечание:
  $f(n) = \Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$

Θ-ОЦЕНКА

Примеры:

- $f(n)=4n^2+n\ ln\ N+174$
  $f(n)= \Theta(n^2)$
- $f(n)=\Theta(1)$ означает, что $f(n)$ или равна $\neq 0$ константе,
  или $f(n)$ ограничена константой на ∞ : $f(n) = 7+1/n = \Theta(1)$

O-ОЦЕНКА

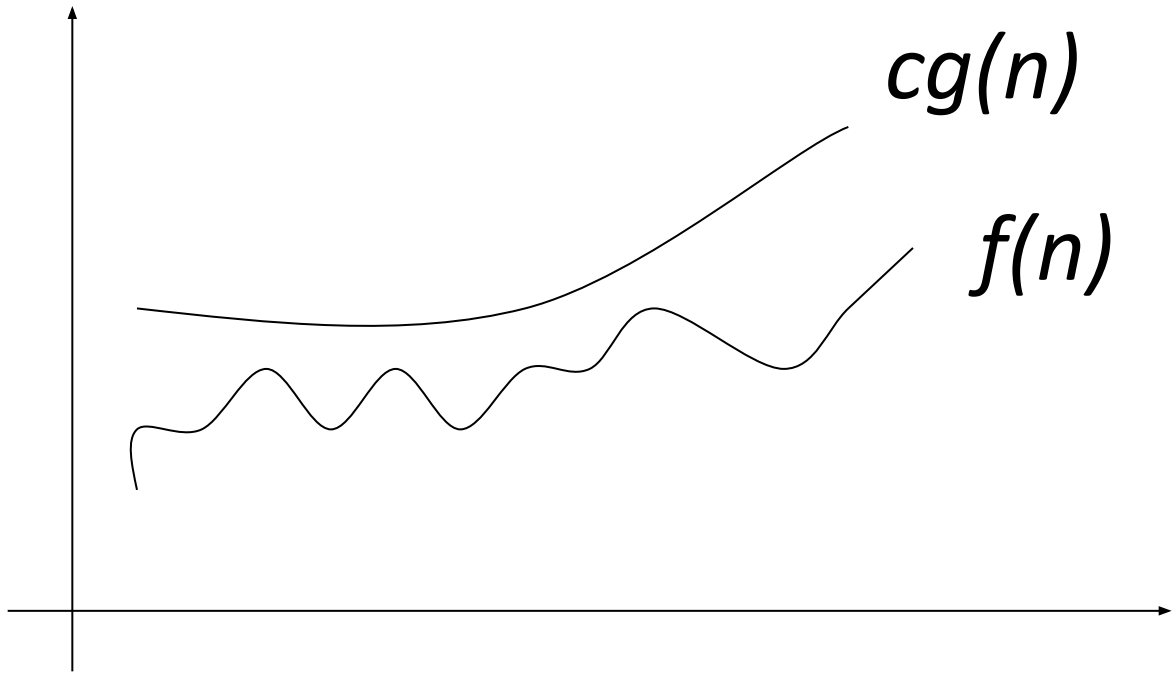
Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента $n \geq 1$, тогда:

$$f(n) = O(g(n)),$$

если $\exists\ c > 0, n0 > 0$ *такие, что*

$0 \leq f(n) \leq c * g(n), \quad \forall n > n0$

O-ОЦЕНКА



O-ОЦЕНКА

- $O(g(n))$ - класс функций, таких, что все они растут не быстрее, чем функция $g(n)$ с точностью до постоянного множителя
- иногда говорят, что $g(n)$ мажорирует функцию $f(n)$

О-ОЦЕНКА

Примеры:

- $f(n)=1/n$,
- $f(n)= 12$,
- $f(n)=3*n+17$,
- $f(n)=n*Ln(n)$,
- $f(n)=6*n^2+24*n+77$

О-ОЦЕНКА

- Замечание:
  указывают наиболее «близкую» мажорирующую функцию, например для $f(n)= n^2$ оценка $O(2^n)$ не имеет практического смысла

Ω -ОЦЕНКА

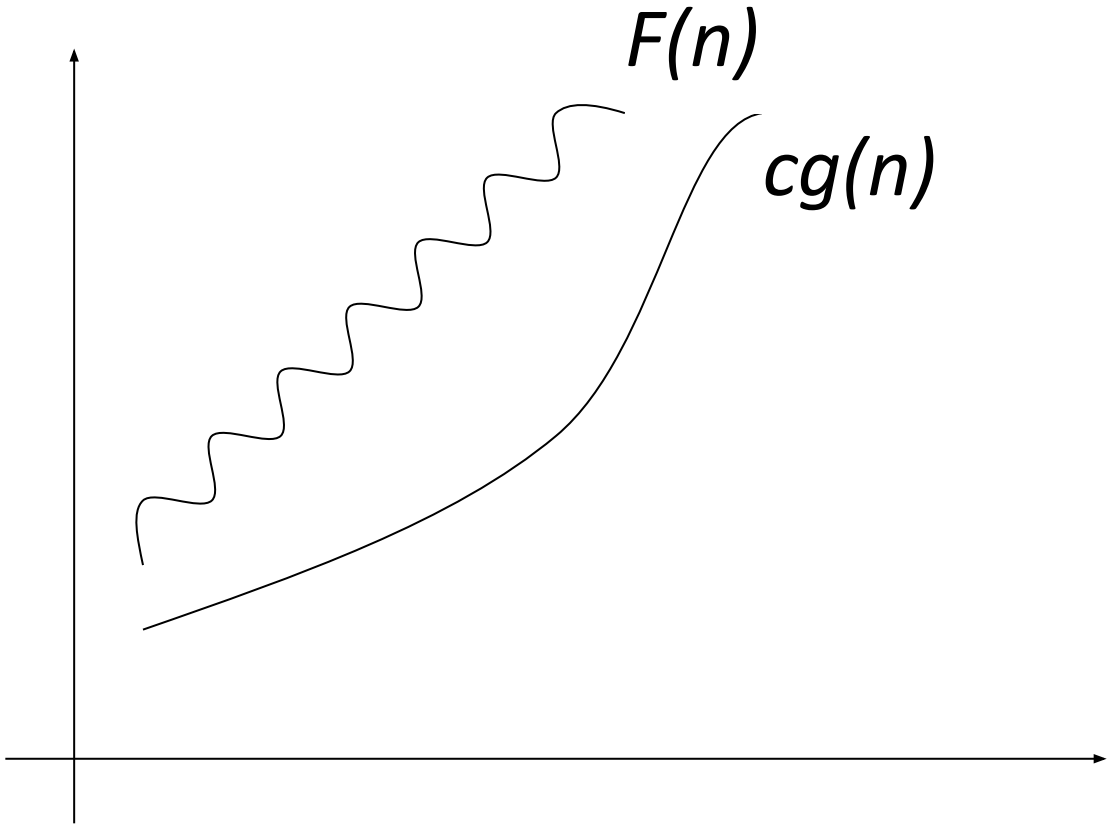
Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента $n \geq 1$, тогда:

$$f(n) = \Omega(g(n)),$$

если $\exists\ c > 0, n0 > 0$ *такие, что*

$0 \leq c * g(n) \leq f(n), \ \forall n > n0$

Ω-оценка



Ω -ОЦЕНКА

- определяет класс функций, которые растут не медленнее, чем $g(n)$ с точностью до постоянного множителя

Ω -ОЦЕНКА

Пример:

- *Ω(n*Ln(n))* обозначает класс функций, которые растут не медленнее, чем *g(n) = n*Ln(n)*
- в этот класс попадают все полиномы со степенью большей единицы
- а также все степенные функции с основанием большим единицы

Асимптотические оценки

- Замечание:
  не всегда для пары функций справедливо одно из асимптотических соотношений, например для $f(n)=n^{1+\sin(n)}$ и $g(n)=n$ не выполняется ни одно из асимптотических соотношений

Асимптотические оценки

- Резюме:
  Знание асимптотики поведения функции трудоемкости алгоритма - его сложности, дает возможность делать прогнозы по выбору более рационального с точки зрения трудоемкости алгоритма для больших размерностей исходных данных

Асимптотические оценки

- Резюме:
  Θ-оценка является более предпочтительной,
  чем оценки O и Ω

ПРАВИЛА ВЫЧИСЛЕНИЙ

- Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух программных фрагментов P_1 и P_2. $T_1(n)$ имеет степень роста $O(f(n))$, а $T_2(n)$ – $O(g(n))$.
- Правило сумм:
  - $T_1(n) + T_2(n)$, то есть время последовательного выполнения фрагментов P_1 и P_2, имеет степень роста $O(max\ (f(n),g(n)))$
  - Если $f(n) \leq g(n)$, то $O(f(n) + g(n)) \equiv O(f(n))$
  - Если c – >0 константа, то $O(f(n)+c) \equiv O(f(n))$
- Правило произведений
  - $T_1(n) \cdot T_2(n)$, то есть время вложенного выполнения фрагментов P_1 и P_2, имеет степень роста $O((f(n) \cdot g(n))$
  - Если c – >0 константа, то $O(c \cdot f(n)) \equiv O(f(n))$

Общие правила вычисления *O*-оценки

- Время выполнения присваивания, чтения и записи имеет порядок *O(1)*.
- Время выполнения последовательности операторов по правилу сумм оценивается наибольшим временем выполнения оператора в данной последовательности.
- Время выполнения условий состоит из времени вычисления логического выражения (имеет порядок роста *O(1)*) и времени выполнения условно исполняемых операторов => имеет порядок роста *O(max (операторы then, операторы else))*
- Время выполнения цикла состоит из времени инициализации, вычисления условия, модификации (имеют порядок роста *O(1)*) и времени выполнения операторов тела цикла => имеет порядок роста <кол-во итераций>**O(max (операторы тела цикла))*

ПРИМЕР 1

Задача: поиск максимума в массиве

```
MaxS (S,n; Max)
Max ← S[1]
For i ← 2 to n
  if Max < S[i]
    then  Max ← S[i]
end for
return Max
```

ПРИМЕР

Худший случай

- Элементы массива отсортированы по возрастанию.
- Трудоемкость :

  $F\hat{}(n)=1+1+1+ (n-1) (3+2+2)=7 n - 4 = \Theta(n)$

ПРИМЕР

Лучший случай

- Максимальный элемент расположен на первом месте
- Трудоемкость :

  $F^{\vee}(n)=1+1+1+ (n-1) (3+2)=5 n - 2 = \Theta(n)$

ПРИМЕР

Средний случай

- При просмотре *к-го* элемента массива переприсваивание максимума произойдет, если в подмассиве из первых *к* элементов максимальным элементом является последний.
- В случае равномерного распределения исходных данных, вероятность того, что максимальный из *к* элементов расположен в последней позиции равна *1/к*.
- Тогда в массиве из *n* элементов общее количество операций переприсваивания максимума определяется :

$$\sum_{i=1}^{N} 1/i = Hn \approx Ln(N) + \gamma, \quad \gamma \approx 0,57$$

ПРИМЕР

Средний случай

- Трудоемкость :

  $F(n)=1 + (n-1)(3+2) + 2(Ln(n) + \gamma)=5n +2 Ln(n) - 4 +2\gamma = \Theta(n)$

ПРИМЕР 2

```
int count_match=0;
int count_change=0;
for(int i=0;i<N;i++)
      for(int j=0;j<N-1;j++)
      {
         if (Ar[j]>Ar[j+1])
         {
            int buf=Ar[j];
            Ar[j]=Ar[j+1];
            Ar[j+1]=buf;
        count_change++;
         }
         count_match++;
      };
```

ПРИМЕР 3

```
int count_match=0;
int count_change=0;
 int i=0, inv=1;
 while(inv)
 {
       inv=0;
       for(int j=0;j<N-1-i;j++)
       {
          if (Ar[j]>Ar[j+1])
          {
              int buf=Ar[j];
              Ar[j]=Ar[j+1];
              Ar[j+1]=buf;
        count_change++;
              inv=1;
          }
          count_match++;
       };
       i++;
 };
```

ПРИМЕР 4

```
void Dragging(int *S, int i, int j)
{
/* i, j задают область элементов массива S, обладающего свойством
сортирующего дерева. Корень строящегося дерева помещается в i */
   int k;
   count_call++;
   if (2*i+1<=j) // если i не лист
   {
      if (2*i+1<j) // если сыновей - 2
         if (S[2*i+1]<S[2*i+2]) k=2*i+2; // находим наибольшего из сыновей
         else k=2*i+1;
      if ((2*i+1)==j) k=j; // если сын один
      if (S[i]<S[k]) // если наибольший сын больше отца, то меняем их местами
      {
         int r=S[k]; S[k]=S[i]; S[i]=r;
         Dragging(S,k,j);
      };
   };
};
```

ПРИМЕР 4

```
for (int i=(n-1)/2; i>=0; i--)
        Dragging(S,i,n-1);

for (int i=N-1; i>=0; i--)
{
   int r=a[0]; a[0]=a[i]; a[i]=r;
   Dragging(a,0,i-1);
};
```