ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ

1. Постановка задачи временной оценки трудоёмкости алгоритма
2. Пооперационный анализ
3. Понятие “элементарной” операции
4. Алгоритмические конструкции
5. Пример 1. Суммирование элементов прямоугольной матрицы
6. Пример 2. Поиск максимального элемента в массиве
7. Метод Гиббсона
8. Метод прямого определения среднего времени
9. Пример 3. Пооперационный анализ алгоритмов умножения

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ

1. Постановка задачи классификации алгоритмов
2. Классификация сложности

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ
1. Постановка задачи временной оценки трудоёмкости (1)

Временная оценка, фактически, зависит от:

- состава команд процессора, необходимых для отображения отдельных операций алгоритма;
- количестве этих команд;
- времени выполнения отдельных типов команд.

Задан: Алгоритм A, обладающий трудоёмкостью $F_A(D_A)$.

Необходимо: оценить время машинной реализации $T_A(D_A)$ алгоритма A.

Временная оценка определяется рядом причин.

1. Особенности формальной системы записи алгоритма на языке программирования (например, *MathLab*, *C++*, *Pascal* et c).

2.Особенности трансляции исходной программы в машинный код.

3. Различие во времени выполнения разнотипных команд процессора.

4. Различие во времени выполнения однотипных команд для разных форматов (типов) данных

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ
1. Постановка задачи временной оценки трудоёмкости (2)

5. Различия, обусловленные типами адресации.
6. Наличие у процессора особенностей архитектуры, способствующих ускорению вычислений:

конвейер,

кеширование памяти,

сопроцессоры и т.д.

МЕТОДЫ ПОСТРОЕНИЯ ОЦЕНОК

- Пооперационный анализ.
- Метод Гиббсона.
- Метод прямого определения среднего времени

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ
2. Метод пооперационного анализа

Пооперационный анализ включает

1. Построение функции трудоёмкости по каждой элементарной операции для заданного формата данных.
2. Экспериментальное либо потактовое определение времени выполнения соответствующей элементарной операции.

Оценка рассчитывается по формуле:

$$T_A(N) = \sum_{i \subset R} F_A^{(i)}(N) \cdot \bar{t}_i,$$

где N – объём входных данных;

i – множество “элементарных” операций, используемых в реализации алгоритма;

R – множество “элементарных” операций, предоставляемых вычислительным устройством;

$F_A^{(i)}(N)$ – число операций i–го типа;

$\bar{t}_i$ – среднее время выполнения i–ой “элементарной” операции.

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ

3. Понятие “элементарной” операции

“Элементарная” операция соотносится с укрупнённой командой языка высокого уровня.

К “элементарным” операциям относят:

- операция присваивания: *y* := *x*;
- операция одномерной индексации: *v*[*i*], предполагается вычисление адреса по формуле (алгоритму)

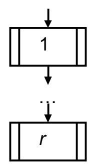
  Базовый_адрес_*v* + *i* × Размер _*v*;
- арифметические операции: +, –, × или (*), /;
- операции сравнения: <, >, =, ≥ и ≤;
- логические операции: *or*, *not*, *and*, *xor* и др.

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ

4. Алгоритмические конструкции (1)

1. Конструкция **“последовательность”** (“следование”)
Число элементарных операций:

$$F_{\Rightarrow} = f_1 + f_2 + \ldots + f_r.$$

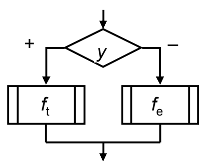


2. Конструкция **“ветвление”** (условный оператор)

if *y* ***then*** f_t **else** f_e

Число элементарных операций:

$$F_{if} = f_t \cdot P + f_e \cdot (1 - P) + 1,$$



где *P* – вероятность того, что результат проверки условия – значение “истина”;

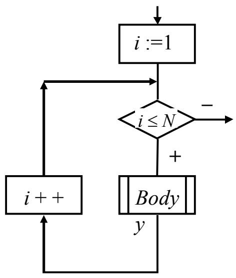
f_t и f_e – число операции по результату проверки;

1 – операция сравнения в ходе проверки.

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ
4. Алгоритмические конструкции (2)

3. Конструкция **"цикл"** с заданным числом повторений

for *i* := 1 ***to*** *N*
/ Body */*
end for



Число элементарных операций заголовка цикла:

1. инициализация управляющей переменной;
2. проверка условия окончания цикла;
3. инкремент управляющей переменной цикла;
4. присвоение нового значения управляющей переменной в теле цикла.

$$1 + 3 \cdot N$$

Число элементарных операций в целом

$$F_{for} = 1 + 3 \cdot N + N \cdot f_{Body}$$

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ
5. Пример 1. Сумма элементов прямоугольной матрицы

Алгоритм работает одинаково при зафиксированных размерах матрицы, поэтому является количественно-зависимым.

Sum_Matr(*Matr*, *N_str*, *N_stl*; *Res*)
    Res := 0; { 1 – (:=)}
    for i := 1 *to N_str* { $1 + 3 \cdot N\_str$}
        for j := 1 *to N_stl* {$1 + 3 \cdot N\_stl$}
            Res := *Res* + *Matr* [*i, j*] {1 (:=) + 1 (+) + 1 ([*i*]) + 1 ([*j*]) = 4}
        end for
    end for

$$F_A(N\_str, N\_stl) = 1+1+ 3 \cdot N\_str + 3 \cdot N\_str(1+ 3 \cdot N\_stl + 4 \cdot N\_stl) =$$
$$= 7 \cdot N\_str \cdot N\_stl + 4 \cdot N\_str + 2 = \Theta(N^2).$$

Пусть $N\_str = N\_stl = N$; тогда $F_A(N, N) = 7 \cdot N^2 + 4 \cdot N + 2 = \Theta(N^2)$.

$$F_A(N\_stl, N\_str) = 1+1+ 3 \cdot N\_stl + 3 \cdot N\_stl(1+ 3 \cdot N\_str + 4 \cdot N\_str) =$$
$$= 7 \cdot N\_stl \cdot N\_str + 4 \cdot N\_stl + 2 = \Theta(N^2).$$

$$F_A(N\_str, N\_stl) – F_A(N\_stl, N\_str) = 4 \cdot (N\_str – N\_stl)$$

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ. 6. Пример 2. Поиск максимального элемента в массиве (1)

Пример 2. Поиск максимального элемента в массиве

big = list [1] { 1(:=) + 1 ([i]) = 2 }

for i = 2 to N { 1 + 3· (N – 1) }

if (*list* [*i*] > *big*) then *big = list* [*i*] { 1(:=) + 2 ([i]) + 1 (>) = 4 }

end for

Число операций определяется как N, так и содержимым массива *list*. Алгоритм – количественно-параметрический

1. Лучший случай $F_A^{\vee}(n)$:

$$2 + 1 + 3\cdot (N-1) + (N-1) \cdot [1([i]) + 1 (>)] = 3 + (N-1) \cdot (3 + 2) = 5\cdot N - 2 = \Theta(n).$$

2. Худший случай $F_A^{\wedge}(n)$:

$$2 + 1 + 3\cdot (N-1) + (N-1) \cdot 4 = 7\cdot N - 4 = \Theta(n).$$

3. Средний случай $\overline{F}_A(n)$:

Пусть максимальный элемент находится среди очередных r элементов подмассива, что вызовет присваивание, а данные распределены равномерно

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ. 6. Пример 2. Поиск максимального элемента в массиве (2)

$$\sum_{r=1}^{N} r^{-1} = H_n = \ln N + \gamma, \qquad \gamma = 0{,}57$$

H_n – n-е гармоническое число. Имеем:

$$3 + (N-1) \cdot (3 + 2) + H_n \cdot [1([i]) + 1 (>)] = 5\cdot N + 2\cdot(\ln N + 2\cdot\gamma) - 2 = \Theta(n).$$

Вычисление оценки среднего случая как среднеарифметического из оценок лучшего и худшего из случаев:

$$0{,}5\times (5\cdot N - 2 + 7\cdot N - 4) = 6\cdot N - 3 = \Theta(n).$$

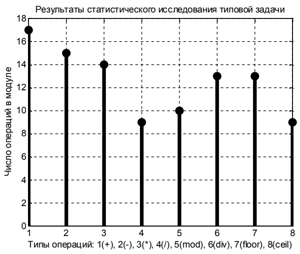
ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ.
7. Метод Гиббсона

Включает этапы

1. Классификация отдельных задач по типам
   - задачи научно-технического и расчётного характера с преобладанием вещественных типов данных;
   - задачи целочисленной арифметики;
   - задачи обработки текстовой информации.
2. Экспериментальное исследование реальных программ с учётом типов задач на частоту встречи тех или иных операций.
3. Определяется время типовой задачи на тестовых прогонах
4. Рассчитывается оценка времени работы алгоритма

$$T_A(N) = T_A(N) \cdot t_t,$$

где t_t – тип задачи.



ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ.
8. Метод прямого определения среднего времени

Метод включает этапы

1. Определить $F_A(N)$ в ходе анализа трудоёмкости.
2. Задавая различные значения длины входа N_0, определить среднее время выполнения программы T_0, на фиксированной длине входа:

$$\bar{t}_A = \frac{\sum_{N_0} T_0(N_0)}{\sum_{N_0} F_A(N_0)}.$$

3. Приняв предположение о статистической устойчивости оценки среднего времени по N, на произвольное значение , определить среднее время:

$$T(N) = \bar{t}_A \times F_A(N).$$

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ.
9. Пооперационный анализ алгоритмов умножения (1)

Mult_A1(a, b ,c, d; Re, Im)	*Mult_A2(a, b ,c, d; Re, Im)*
Re := $a \times c + b \times d$	*r1* := $c \times (a + b)$
Im := $a \times d + b \times c$	*r2* := $b \times (d + c)$
Return (*Re*, *Im*)	*r3* := $a \times (d - c)$
End	*Re* := *r1* – *r2*
	Im := *r1* – *r3*
	Return (*Re*, *Im*)
	End

ВРЕМЕННЫЕ ОЦЕНКИ ТРУДОЁМКОСТИ АЛГОРИТМОВ.
9. Пооперационный анализ алгоритмов умножения (2)

Операция умножения комплексных чисел определяется следующим образом
$(a + j \cdot b) \times (c + j \cdot d) = (a \cdot c - b \cdot d) + j \cdot (a \cdot d - b \cdot c) = Re + j \cdot Im$
В *Mult_A1* операций: 8 = 4 (×) + 2 (+/ –) + 2 (:=).
В *Mult_A2* операций: 13 = 3 (×) + 5 (+/ –) + 5 (:=).

Пересчитаем операции в “попугаях” – операциях (+/ –) . Пусть:
- $k_u > 1$ для пересчёта операции умножения $t\,(\times) = k_u \times t\,(+/-)$;
- $k_p > 1$ для пересчёта операции присваивания $t\,(:=) = k_p \times t\,(+/-)$.

Пересчитанные времена выполнения:
- $T_1 = [4\cdot k_u + 4\cdot k_p + 2] \times t\,(+/-)$,
- $T_2 = [3\cdot k_u + 5\cdot k_p + 5] \times t\,(+/-)$,
- $\Delta T = T_1 - T_2 = [\,k_u - 3\cdot k_p - 3] \times t\,(+/-)$.

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
1. Постановка задачи классификации алгоритмов (1)

Пусть существует алгоритмически разрешимая задача и известен алгоритм её решения A с оценкой трудоёмкости $F_A^{\wedge}(D_A) = O[g(D_A)]$ (по "худшему" случаю)

Ставятся вопросы:

- о *существовании* функции $g(D_A)$, являющейся нижним предельным значением (функциональным пределом) оценок трудоёмкости;
- о *возможности* создания алгоритма с худшим значением трудоёмкости, равным функциональному пределу.

Ответ на эти вопросы даёт *ТЕОРИЯ СЛОЖНОСТИ ВЫЧИСЛЕНИЙ*. Её цель:

определение оценки сложности "гипотетически наилучшего" алгоритма для "худшего" набора данных.

Функциональный теоретический нижний предел трудоёмкости $F_{\Theta\lim} = \min\{\Theta[F_A^{\wedge}(D)]\}$

Если его удаётся найти, то можно утверждать, что любой алгоритм решения задачи будет работать не быстрее $F_{\Theta\lim}$, в худшем случае, то есть

$$F_A^{\wedge}(D) = \Omega(F_{\Theta\lim})$$

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
1. Постановка задачи классификации алгоритмов (2)

Примеры.

1. При выборе максимального либо минимального элемента массива чисел, должен быть рассмотрен каждый его компонент. Поэтому

$$F_{\Theta\lim} = \Theta(n).$$

2. Выполняя умножение матриц, ввиду необходимости обрабатывать все элементы данных, представляется справедливой оценка

$$F_{\Theta\lim} = \Theta(n^2).$$

Экспериментально выяснилось, что "быстрые"алгоритмы умножения дают оценку $\Theta(n^{2,34})$. Одно из двух:

- существует, но, до сих пор, не построен скоростной алгоритм умножения матриц;
- оценку следует рассматривать как теоретически предельную и доказать это.

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (1)

Alan Cobham & Jack Edmonds – авторы классификации задач по сложности.

1. Класс P – задачи с полиномиальной сложностью

Задача является *полиномиальной*, если существует константа k и алгоритм, решающий задачу с оценкой трудоёмкости $F_A(n) = O(n^k)$, где n – длина входа алгоритма, $n = |D|$.

Класс P обладает рядом особенностей:

- для большинства задач этого класса $k < 6$;
- класс P инвариантен по модели вычислений (подходит для приложений в широком классе моделей);
- класс P обладает свойством естественной замкнутости, ибо сумма либо произведение полиномов является полиномом;
- класс P эквивалентен понятию “практически решаемая задача”.

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (2)

2. Класс NP – полиномиально проверяемые задачи

Содержательно задача относится к классу NP, если её решение может быть полиномиально проверено.

Пусть для всех $D \in D_A$, $|D| = n$ поставлены в соответствие

- сертификат $S \in S_A$, такой что $|S| = O(n^r)$;
- алгоритм $A_S = A_S(D, S)$,такой, что выдаёт 1, если решение правильно, и 0 в противном случае.

Если $F(A_S) = O(n^m)$, то задача принадлежит к классу NP.

Пример. Задача о сумме.

Дано: N чисел в виде массива $A = \{a_1, a_2, \dots , a_n\}$ и число V.

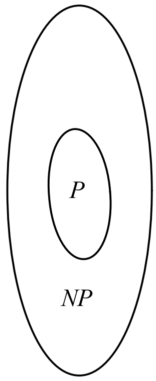
Найти: массив $X = \{x_1, x_2, \dots , x_n\}$, $x_i \in \{0, 1)$, такой что $V = \sum a_i \times x_i$.

Проверка результата потребует не более $\Theta(N)$ операций (сложность проверки)

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (3)

Вопрос: “Можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить за полиномиальное время?”

$NP \setminus P \neq \{\ \}$



Его сформулировал Jack Edmonds в форме $P = NP$?

Если задача принадлежит классу P, то она, может быть проверена “перерешиванием” (повторением хода решения).

Доказательства совпадения или несовпадения классов P и NP отсутствуют. Существует предположение, что P является собственным подмножеством класса NP.

СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (4)

3. Класс NP-полных задач (NPC: NP-complete)

Понятие NPC было введено С. Куком (Stephen Cook) и исходит из идеи сводимости одной задачи к другой.

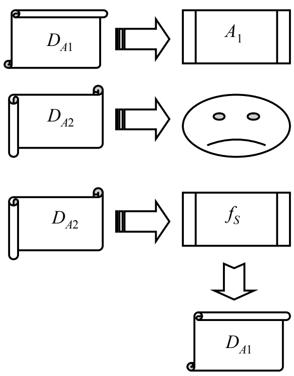
Пусть

- D_{A1} – множество входных проблем задачи 1;
- D_{A2} – множество входных проблем задачи 2.

Если существует функция f_S, которая сводит конкретную постановку 2-й задачи d_{A2} к конкретной постановке 1-й задачи d_{A1}, то есть

$$f_S(d_{A2} \in D_{A2}) = d_{A1} \in D_{A1},$$

то задача 2 сводима к задаче 1.



СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (5)

Если $F_A(f_S) = O(n^k)$, то алгоритм сведения принадлежит к классу P, а задача 2 полиномиально сводится к задаче 1.

Считается, что задачи представлены (заданы) на разных языках, 1 – на L_1, а 2 – на L_2, то полиномиальную сводимость обозначают как $L_2 \leq_P L_1$.

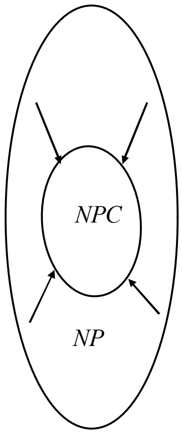
Существуют условия, определяющие класс *NPC* :

- задача должна принадлежать к классу *NP* ($L \in NP$);
- к данной задаче должны полиномиально сводиться все задачи из класса *NP*.

$$L_X \leq_{\mathrm{P}} L,\ L_X \in NP.$$

Теорема.

Если существует задача, принадлежащая классу *NPC*, для которой существует полиномиальный алгоритм решения с оценкой $F = O(n^k)$, то класс P совпадает с классом *NP*, то есть, $P = NP$.



СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ
2. Классификация сложности (5)

Доказано существование множества *NPC*-задач, но не удалось найти полиномиальных алгоритмов их решения.

Примеры *NPC*-задач

1. Задача о выполнимости комбинационной схемы

   Имеется схема, составленная из не более чем $O(n^k)$ элементов И, ИЛИ, НЕ с n входами и одним выходом каждый.

   Существует ли входная комбинация $\{x_1, x_2, \ldots , x_n\}$ на множестве $\{0, 1\}$, при котором на выходе схемы будет 1?

   Решение можно получит перебором 2^n входных комбинаций, то есть $F^{\wedge}(n) = O(n^k \times 2^n)$, n^k – число операций затрачиваемые на проверку. Полиномиальный алгоритм не известен

2. Задача о сумме

$$F_A(N, V) = O(N \times 2^N).$$