Анализ алгоритмов

Алтайский государственный университет
Факультет математики и ИТ
Кафедра информатики

Барнаул 2014

Лекция 16

- План
  - Эффективность алгоритмов
    - Эффективность алгоритмов и ее измерение
    - Временная сложность алгоритма в зависимости от размера задачи
    - Худший, средний и лучший случаи
    - Что ускорять: компьютер или алгоритм?
  - Асимптотический анализ алгоритмов
    - Цели асимптотического анализа
    - O-символика
    - Примеры асимптотического анализа алгоритмов
    - Асимптотическая сложность задач
    - Временная и пространственная сложность
  - Бинарный поиск
    - Идея алгоритма
    - Временная сложность алгоритма

Эффективность алгоритмов

- Эффективность алгоритмов и ее измерение
- Временная сложность алгоритма в зависимости от размера задачи
- Худший, средний и лучший случаи
- Что ускорять: компьютер или алгоритм?

Эффективность алгоритмов

- Часто для решения одной и той же задачи могут быть использованы различные алгоритмы
- Как выбрать алгоритм?
- При разработке программ преследуется две (часто конфликтующие) цели
  1. Разработать алгоритм, простой для понимания, кодирования и отладки
     - Предмет изучения дисциплины «Software Engineering»
  2. Разработать алгоритм, **эффективно** использующий ресурсы компьютера
     - Предмет изучения дисциплины «Структуры данных и анализ алгоритмов»

Эффективность алгоритмов

- Основные ресурсы
  - Время выполнения алгоритма
    - Определяется количеством тривиальных шагов, необходимых для решения задачи
  - Пространство, используемое алгоритмом
    - Определяется объёмом оперативной памяти или памяти на носителе данных

Остается «за скобками» :

  - Трудоемкость кодирования алгоритма
    - Определяется временными затратами программиста на кодирование и отладку алгоритма

Как измерять эффективность алгоритмов?

- Возможные пути

1. Экспериментальное (эмпирическое) сравнение алгоритмов
   - Сравнение затрат времени (памяти) при непосредственном запуске программ

2. Асимптотический анализ алгоритмов
   - Построение теоретических оценок затрат времени (памяти) в зависимости от различных факторов

От чего зависит время выполнения?

- От загрузки машины
- От операционной системы
- От компилятора
- От специфики значений входных данных
- От размера задачи
  - Зависимость времени выполнения **T** от размера задачи *n* выражается некоторой функцией **T**(*n*)

Зависимость времени от размера

- Пример 1. Поиск максимума

```
int largest(int array[], int n) {
  int currlarge = 0;
  for (int i=1; i<n; i++)
    if (array[currlarge] < array[i])
      currlarge = i;
  return currlarge;
}
```

$\mathbf{T}(n) = c_1 n + c_2$

- Пример 2. Подсчет

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; i<n; j++)
      sum++;
```

$\mathbf{T}(n) = c_1 n^2 + c_2$

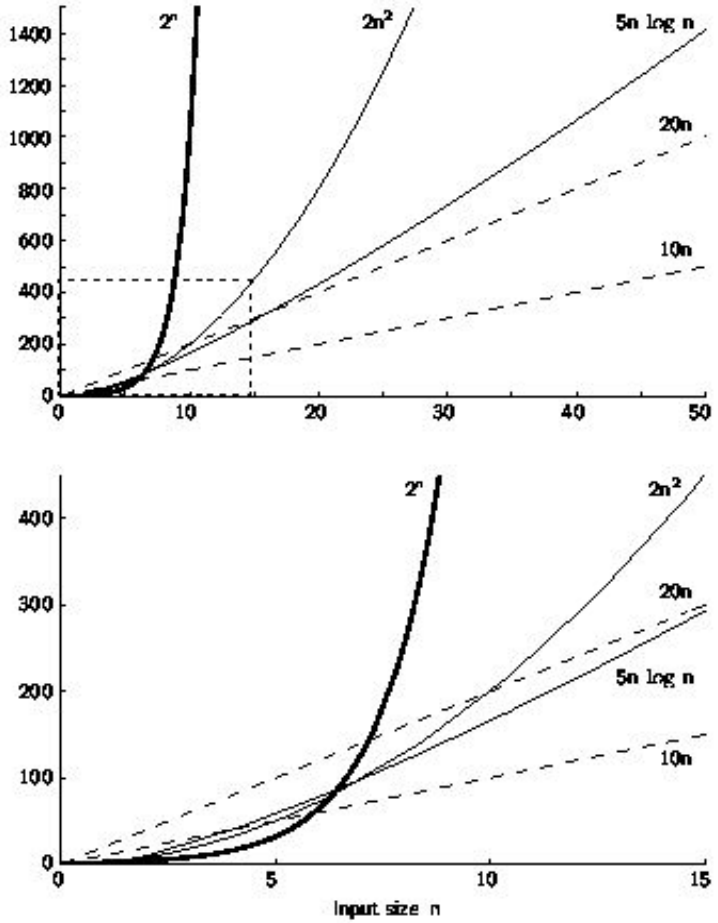
- Пример 3. Присваивание

```
count = 0;
```

$\mathbf{T}(n) = c$

Характер роста функций

- В зависимости от вида функции **T**(*n*) характер ее роста может быть разным



Наилучший, наихудший и средний случаи

- При том же размере входных данных *n* время выполнения алгоритма может быть разным

- **Пример.** Последовательный поиск элемента *K* в массиве размера *n*
  - Элементы массива, начиная с первого, поочередно просматриваются до тех пор пока не найден *K*
  - *Наилучший случай:* *K* найден на 1-й позиции
  - *Наихудший случай:* *K* найден на *n*-й позиции
  - *Средний случай:* в среднем *K* обнаружится после $(n+1)/2$ сравнений

Какой из случаев оценивать?

- Анализировать поведение алгоритма в среднем – наиболее разумно, но наиболее трудно
  - Требуется знание распределения значений (частоты возникновения тех или иных данных)
- Поведение алгоритма в худшем случае важно анализировать в алгоритмах реального времени
  - Пример – системы диспетчеризации транспорта

Ускорять компьютер или алгоритм?

- Что произойдет, если использовать компьютер в 10 раз производительнее?

T(n)	n	n'	Изменение	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}\, n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10n}$	3.16
2^n	13	16	$n' = n + 3$	---

Ускорять компьютер или алгоритм?

- Абсолютные временные затраты алгоритмов разной сложности

T(n)	n=10	n=10^3	n=10^6
log n	0.2 сек	0.6 сек	1.2 сек
n	0.6 сек	1 час	16.6 час
n^2	6 сек	16.6 час	1902 года
2^n	1 час	10^{295} лет	10^{300000} лет

Асимптотический анализ алгоритмов

- Цели асимптотического анализа
- О-символика
- Примеры асимптотического анализа алгоритмов
- Асимптотическая сложность задач
- Временная и пространственная сложность

Асимптотический анализ

- Экспериментальное сравнение алгоритмов трудоемко
- На практике чаще используется более простой асимптотический анализ алгоритмов
- Асимптотический анализ алгоритмов направлен на получение и сравнение теоретических оценок сложности алгоритмов (вида **T**(*n*)) при достаточно больших *n*

Асимптотический анализ

- В асимптотическом анализе алгоритмов используются обозначения, принятые в математическом асимптотическом анализе

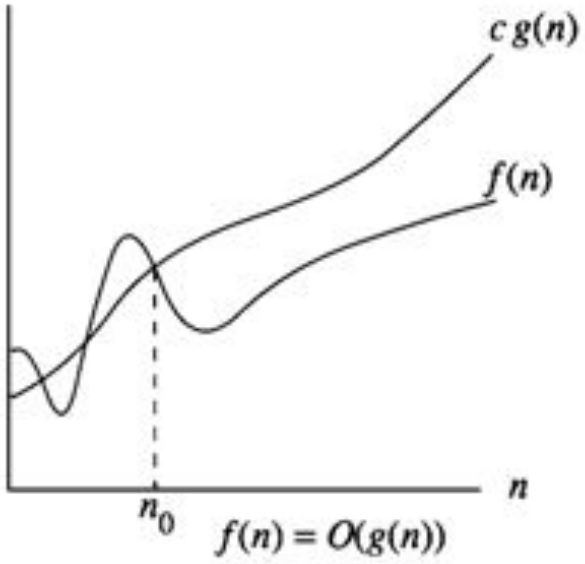
- *O*-символика
  - $o(f(n))$ оценка порядка малости
  - $O(f(n))$ оценка верхней границы
  - $\Omega(f(n))$ оценка нижней границы
  - $\Theta(f(n))$ оценка верхней и нижней границы

Асимптотический анализ: O()

- Определение
  - Говорят, что неотрицательная функция $f(n)$ есть $O(g(n))$, если существуют константы $c>0$ и $n_0>0$, такие что $f(n) \le cg(n)$ для любых $n > n_0$.
- Пример использования
  - Временная сложность алгоритма есть $O(n^2)$ в [лучшем, среднем, худшем] случае.
- Смысл
  - Для всех достаточно больших входных данных ($n>n_0$), алгоритм всегда выполняется менее, чем за $cg(n)$ шагов в [лучшем, среднем, худшем] случае

Асимптотический анализ: O()

- Другими словами
  - Запись $f(n)=O(g(n))$ означает, что $f(n)$ принадлежит классу функций, которые растут не быстрее, чем функция $g(n)$ с точностью до постоянного множителя.



- Пример. Если $\mathbf{T}(n) = 3n^2$, то $\mathbf{T}(n)$ есть $O(n^2)$

Асимптотический анализ: O()

- **O()** указывает верхнюю границу
- При выборе верхней границы наиболее интересна наименьшая из возможных
- Пример.
  - Хотя **T**(n) = $3n^2$ есть O(n^3), мы выбираем O(n^2) как более информативный вариант

Асимптотический анализ: O()

- **Пример 1.** Линейный поиск в массиве (средний случай)

  $\mathbf{T}(n) = c_s n/2$

  Для всех $n > 1$, $c_s n/2 \leq c_s n$

  Таким образом, по определению,
  $\mathbf{T}(n)$ есть $O(n)$ при $n_0 = 1$ и $c = c_s$

Асимптотический анализ: O()

- **Пример 2.** Пусть $\mathbf{T}(n) = c_1 n^2 + c_2 n$ в среднем случае

  $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$
  для всех $n > 1$

  $\mathbf{T}(n) \leq cn^2$ при $c = c_1 + c_2$ и $n_0 = 1$.

  Тогда $\mathbf{T}(n)$ есть $O(n^2)$ по определению

- **Пример 3.** $\mathbf{T}(n) = c$. Говорят: $\mathbf{T}(n)$ есть $O(1)$.

Асимптотический анализ: O()

- Распространенная ошибка
  - “Лучшим для моего алгоритма является случай n=1, т.к. при этом алгоритм наиболее быстр” – НЕКОРРЕКТНО!
- O() описывает характер роста по мере стремления n к ∞
- Лучшим случаем называют такой, при котором на обработку входных данных размера n тратится наименьшее время по сравнению с прочими данными того же размера

Асимптотический анализ: O()

- Распространенная ошибка
  - Часто худший случай путают с верхней границей
- Верхняя граница описывает характер роста по мере стремления n к ∞
- Худшим случаем называют такой, при котором на обработку входных данных размера n тратится наибольшее время по сравнению с прочими данными того же размера

Асимптотический анализ: Ω()

- Определение
  - Говорят, что неотрицательная функция $f(n)$ есть $\Omega(g(n))$, если существуют константы $c>0$ и $n_0>0$, такие что $f(n) \geq cg(n)$ для любых $n > n_0$.

- Смысл
  - Для всех достаточно больших входных данных ($n>n_0$), алгоритм всегда выполняется более, чем за $cg(n)$ шагов
  - Нижняя граница
  - Оценка $\Omega(g(n))$ задает нижнюю асимптотическую оценку роста функции $f(n)$ и определяет класс функций, которые растут не медленнее, чем $g(n)$ с точностью до постоянного множителя

Асимптотический анализ: Ω()

- **Пример.** $\mathbf{T}(n) = c_1 n^2 + c_2 n$.

$c_1 n^2 + c_2 n \geq c_1 n^2$ для любых $n > 1$

$\mathbf{T}(n) \geq cn^2$ для $c = c_1$ и $n_0 = 1$

Таким образом, $\mathbf{T}(n)$ есть $\Omega(n^2)$ по определению

Из всех нижних границ интересна наибольшая

Асимптотический анализ: Θ()

- Определение
  - Говорят, что неотрицательная функция $f(n)$ есть $\Theta(g(n))$, если существуют константы $c_1>0$, $c_2>0$ и $n_0>0$, такие что $c_1 g(n) \le f(n) \le c_2 g(n)$ для любых $n > n_0$.
  - Функция $f(n)$ есть $\Theta(g(n))$, если она одновременно есть $\Omega(g(n))$ и $O(g(n))$

- Смысл
  - Асимптотическое равенство (с точностью до константы)
  - Полностью описывает характер роста функции

Асимптотический анализ

- Правила упрощения
  1. **Транзитивность**
     Если $f(n)$ есть $O(g(n))$ и $g(n)$ есть $O(h(n))$,
     то $f(n)$ есть $O(h(n))$
  2. **Игнорирование констант**
     Если $f(n)$ есть $O(kg(n))$ для любой константы $k > 0$,
     то $f(n)$ есть $O(g(n))$
  3. **Отбрасывание членов низких порядков**
     Если $f_1(n)$ есть $O(g_1(n))$ и $f_2(n)$ есть $O(g_2(n))$,
     то $(f_1 + f_2)(n)$ есть $O(\max(g_1(n), g_2(n)))$
  4. **Мултипликативность**
     Если $f_1(n)$ есть $O(g_1(n))$ и $f_2(n)$ есть $O(g_2(n))$,
     то $f_1(n)f_2(n)$ есть $O(g_1(n)g_2(n))$

Асимптотический анализ

- Пример 1

```
a = b;
```

Присвоение требует константного времени, поэтому имеет сложность Θ(1)

- Пример 2

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

Цикл имеет линейную сложность, т.е. Θ(*n*)

Асимптотический анализ

- Пример 3

```
sum = 0;
for (j=1; j<=n; j++)
  for (i=1; i<=j; i++)
    sum++;
for (k=0; k<n; k++)
  A[k] = k;
```

Первая строка есть $\Theta(1)$

Вложенный цикл есть $\Sigma i = \Theta(n^2)$

Последний цикл есть $\Theta(n)$

Итог: $\Theta(n^2)$

Асимптотический анализ

- Пример 4

```
sum1 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    sum2++;
```

Первая пара вложенных циклов – n^2 шагов

Вторая пара вложенных циклов – $(n+1)(n)/2$ шагов

Обе пары есть $\Theta(n^2)$

Итог: $\Theta(n^2)$

Асимптотический анализ

- Пример 5

```
sum1 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
    sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=k; j++)
    sum2++;
```

Первая пара циклов: Σn для k=1…log n есть $\Theta(n \log n)$

Вторая пара циклов: $\Sigma 2^k$ для k=0…log n–1 есть $\Theta(n)$

Итог: $\Theta(n \log n)$

Бинарный поиск

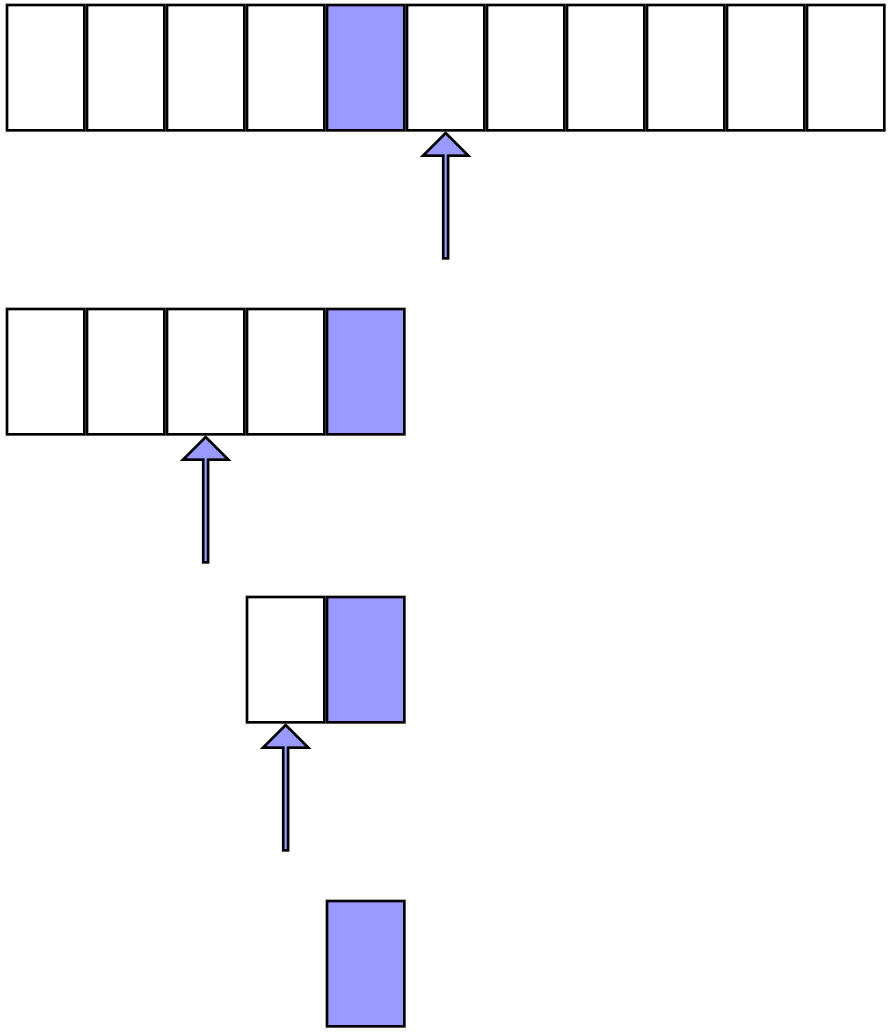
- Идея алгоритма
- Временная сложность алгоритма

Бинарный поиск

- Возможно ли ускорить линейный алгоритм поиска элемента в массиве? ($\Theta(n)$)
- Да, если массив упорядочен
- Наличие порядка позволяет использовать принцип «разделяй и властвуй»: на каждом шаге уменьшая вдвое размер решаемой задачи
- Бинарный поиск – реализация стратегии «разделяй и властвуй» в чистом виде

Бинарный поиск

- На каждом шаге центральный элемент $A[m]$ проверяется на совпадение с искомым K
- Если $A[m] = K$, конец алгоритма
- Если $A[m] < K$, то далее решается задача поиска в подмассиве $A[m]…A[n]$?
- иначе – в подмассиве $A[1]…A[m]$



Бинарный поиск

- Код

```
// Возвращает позицию элемента K
  // в упорядоченном массиве размера n
int binary(int array[], int n, int K) {
  int l = -1;
  int r = n;          // l, r за границами массива
  while (l+1 != r) {        // Стоп, если l, r сошлись
    int i = (l+r)/2;        // Центральный элемент
    if (K < array[i]) r = i;  // Идем налево
    if (K == array[i]) return i;  // K найден
    if (K > array[i]) l = i;  // Идем направо
  }
  return n; // K не встречается в массиве
}
```

- Сколько сравнений производится в худшем случае?

Асимптотический анализ различных управляющих конструкций

- Цикл `while` анализируется так же как и `for`
- Условный оператор `if` оценивается по наиболее сложной ветви `then`/`else`
  - Вероятность срабатывания ветвей не должна зависеть от *n*
- Оператор ветвления `switch` оценивается по наиболее сложной ветви `case`
  - Вероятность срабатывания ветвей не должна зависеть от *n*
- Сложность вызова подпрограммы равна сложности подпрограммы

Асимптотический анализ сложности задач

- Анализ задачи = анализ классов алгоритмов
- Верхняя граница: верхняя граница сложности наилучшего из известных алгоритмов решения задачи
- Нижняя граница: нижняя граница для всех известных алгоритмов решения задачи

Асимптотический анализ сложности задач

- Пример

- Нет смысла говорить о верхних/нижних границах, если известно точное количество операций

- Пример неточных знаний о задаче: **сортировка**

  1. Сложность ввода/вывода: $\Omega(n)$.
  2. Пузырьковая сортировка или вставками: $O(n^2)$.
  3. Более эффективные методы (Quicksort, Mergesort, Heapsort, и т.д.): $O(n \log n)$.
  4. Для некоторых типов данных существуют методы со сложностью $O(n)$.

Асимптотический анализ: несколько параметров

- Упорядочить по популярности C значений пикселов на изображении, содержащем P пикселов

```
for (i=0; i<C; i++)   // Инициализировать счетчики
  count[i] = 0;
for (i=0; i<P; i++)   // Просмотреть все пикселы
  count[value(i)]++; // Увеличить счетчик значения
sort(count);          // Сортировать счетчики
```

- Если в качестве размера данных использовать P, то время работы есть $\Theta(P \log P)$
- Более точно: $\Theta(P + C \log C)$

Асимптотический анализ: затраты памяти

- Асимптотический анализ затрат памяти проводится совершенно аналогично анализу временных затрат
- Обычно такая потребность возникает при построении структур данных

Баланс «затраты памяти»/«затраты времени»

- Временные затраты алгоритма могут быть понижены, если повысить расход памяти и наоборот:
- Жертвуя временем, можно сэкономить память

- Принцип баланса «время»/«дисковое пространство»:
  - Чем меньше затраты дискового пространства, тем быстрее программа

Вопросы?

- Эффективность алгоритмов
  - Эффективность алгоритмов и ее измерение
  - Временная сложность алгоритма в зависимости от размера задачи
  - Худший, средний и лучший случаи
  - Что ускорять: компьютер или алгоритм?
- Асимптотический анализ алгоритмов
  - Цели асимптотического анализа
  - О-символика
  - Примеры асимптотического анализа алгоритмов
  - Асимптотическая сложность задач
  - Временная и пространственная сложность
- Бинарный поиск
  - Идея алгоритма
  - Временная сложность алгоритма