

Лекции 1-2. Основные понятия вычислительной геометрии

1. Основные понятия

Вычислительная задача - это задача, которая может быть решена путем выполнения согласованного набора команд. Вычислительная задача формулируется в виде постановки задачи с описанием всей необходимой входной информации и желаемого результата как функции этой входной информации.

Алгоритмом называется способ решения вычислительной задачи. Алгоритм абстрактен в том смысле, что может быть реализован различными путями, например, на разных компьютерах.

Алгоритмическая парадигма - идея (общий метод), на основе которого реализуется алгоритм.

Алгоритмы используют *структуры данных*. Алгоритмы мотивируют разработку и изучение структур данных, а структуры данных служат материалом, из которого формируются алгоритмы.

Абстрактные типы данных (АТД) дают общие представления структуры данных, независимо от ее применения. АТД включает набор операций, которые поддерживаются структурой данных. Эти операции служат интерфейсом между алгоритмом и его структурами данных: алгоритм выполняет операции, поддерживаемые АТД, а структура данных реализует эти операции.

Анализ алгоритмов - это измерение и описание их характеристик. Обычно существует несколько конкурирующих алгоритмов для решения задачи, а для АТД есть несколько видов структур данных. Анализ алгоритмов позволяет выбрать способ описания и измерения свойств алгоритмов и структур данных. Это является основой для выбора наилучшего решения задачи.

Основными характеристиками, используемыми при анализе алгоритмов являются *затраты времени* и *объем памяти* на выполнение алгоритма. Существуют три подхода, упрощающие задачу анализа алгоритмов без снижения полезности результата. Первый состоит в использовании *абстрактной модели вычислений*, второй - в определении зависимости времени работы от объема входных данных, третий - в выражении времени работы в виде простой функции отношения *память/время*.

2. Модель вычислений

Для анализа алгоритма нужно идентифицировать *операции*, которые в нем используются, и оценить стоимость каждой из них. Модель вычислений должна включать наиболее существенные операции алгоритма. Числовые алгоритмы обычно анализируются путем подсчета арифметических операций, алгоритмы сортировки и поиска - подсчетом количества сравнений пар элементов.

Стоимость каждой операции, учитываемой моделью вычислений, устанавливается в абстрактных единицах времени, называемых шагами. *Время работы* алгоритма равно общему числу выполняемых шагов.

Определенное таким образом время вычислений зависит от модели вычислений. Поэтому естественно принять одинаковую модель вычислений для различных решений одной и той же задачи для того, чтобы их можно было сравнивать и искать наиболее эффективное решение. По этой причине модель вычислений (часто неявно) является частью постановки задачи.

Пример - задача ПОИСК:

Для заданного целого числа x и отсортированного массива a различных целых чисел вернуть индекс числа x внутри массива a , а если такого числа в массиве нет, то вернуть значение -1 .

Модель вычислений для этой задачи содержит только операцию *проба*: сравнение ключа поиска x с некоторым целым числом в массиве a . Поскольку одиночная проба может быть выполнена за постоянное время, стоимость пробы составляет один шаг.

Используем *алгоритмическую парадигму* последовательного поиска: пошаговый просмотр массива a с пробой каждого целого числа.

Возможная реализация алгоритма:

```
FUNCTION SequentialSearch(x: INTEGER;
    CONST a: SearchArray; n: INTEGER): INTEGER;
(* последовательный поиск ключа x в массиве a из
   n элементов с возвращением индекса найденного
   элемента либо -1, если такового нет *)
VAR i: INTEGER;
BEGIN
    Result:=-1;
    FOR i:=1 TO n DO
        IF a[i]=x THEN
            BEGIN Result:=i; Break; END;
    END;
```

Обозначим время работы программы SequentialSearch в виде функции от данных через $f(x, a, n)$. Тогда очевидно

$$f(x, a, n) = \begin{array}{ll} k+1 & \text{- если число } x \text{ обнаружено в позиции } k \\ n & \text{- в противном случае.} \end{array}$$

3. Мера сложности

Для упрощения анализа время работы и требуемая память выражаются обычно как функции только размера массива, а не всех допустимых входимых значений. Соответствующие значения этих функций определяют *временную сложность* и *пространственную сложность* алгоритма. Мера сложности определяется для *наихудшего случая* или *в среднем*.

Время работы в *наихудшем случае* определяется для самого длительного выполнения алгоритма для любых входных данных при каждом размере входа. Например, программа SequentialSearch в *наихудшем случае* имеет время работы $T(n)=n$. Вычисление времени работы в *наихудшем случае* предполагает конструирование этого *наихудшего случая*. Несмотря на то, что соответствующий анализ является часто весьма сложным, он, как правило, более прост и реалистичен, чем для оценок в *среднем*.

Время работы в *среднем* определяет усредненное значение времени работы для всех видов входных данных для каждого размера входа. Для программы в предположении, что поиск завершился успешно и что значение ключа равновероятно равно каждому элементу массива, время работы в *среднем* равно $T(n)=(n+1)/2$. Если же поиск может завершиться неуспешно, то это значение лежит в диапазоне $[(n+1)/2, n]$.

4. Асимптотический анализ

Асимптотическая эффективность алгоритмов показывает, как изменяется время работы алгоритма при стремлении к бесконечности размера входного массива. Пусть время работы алгоритма выражается функцией $f(n)$ от размера входа n . Нас будет интересовать скорость роста этой функции. При этом мы будем сравнивать эту скорость роста со скоростями роста следующих простейших функций $T(n)$:

$T(n)=1$ (постоянное время) - алгоритм выполняется за время, независимое от n ;

$T(n)=\log n$ (логарифмическое время) - алгоритм основан на парадигме многократного разбиения задачи на подзадачи фиксированного размера;

$T(n)=n$ (линейное время) - алгоритм затрачивает фиксированное время на обработку каждого элемента входного массива независимо от n ;

$T(n)=n \log n$ («почти линейное время») - такое время имеют многие алгоритмы, основанные на парадигмах «разделяй и властвуй», плоского заметания, балансировки;

$T(n)=n^2$ (квадратичное время) - алгоритм затрачивает постоянное время на обработку всех пар элементов входного массива;

$T(n)=n^3$ (кубическое время) - на обработку каждой пары элементов затрачивается линейное время.

5. Асимптотическая нотация

Асимптотическая нотация является удобным языком для описания скорости роста функций.

Пусть задана функция $f(n)$.

Запись $O(f(n))$ обозначает множество всех функций, которые растут не быстрее, чем $f(n)$. Функция $g(n)$ принадлежит множеству $O(f(n))$, если $g(n)$ не более, чем в постоянное число раз превышает $f(n)$ при достаточно большом n . Более формально, $g(n) \in O(f(n))$, если существуют вещественное число $c > 0$ и целое число $n_0 \geq 1$ такие, что $g(n) \leq c f(n)$ для всех $n \geq n_0$.

Например, $an+b \in O(n)$ для всех констант a и b . Для доказательства положим $c = |a+1|$ и $n_0 = |b|$.

Запись $\Omega(f(n))$ обозначает множество всех функций, которые растут не более медленно, чем $f(n)$. Функция $g(n)$ принадлежит множеству $\Omega(f(n))$, если существуют вещественное число $c > 0$ и целое число $n_0 \geq 1$ такие, что $g(n) \geq c f(n)$ для всех $n \geq n_0$.

Запись $\Theta(f(n))$ обозначает множество всех функций, имеющих скорость роста такую же, как и функция $f(n)$. Формально, $g(n) \in \Theta(f(n))$, если существуют вещественные числа $c_1 > 0$ и $c_2 > 0$ и целое число $n_0 \geq 1$ такие, что для всех $n \geq n_0$ имеет место $c_1 f(n) \leq g(n) \leq c_2 f(n)$.

Таким образом, $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

6. Рекурсивные алгоритмы

Рекурсивное решение задачи состоит в декомпозиции большой задачи на более мелкие подзадачи того же вида, решении этих подзадач и в последующем объединении полученных решений для формирования решения исходной задачи. Подзадачи решаются рекурсивно тем же самым алгоритмом. В результате декомпозиции должны быть в конечном счете получены подзадачи

столь простые (малой размерности), что их решение может быть получено непосредственно.

Такая алгоритмическая парадигма называется *рекурсивной декомпозицией*. Алгоритмы, основанные на рекурсивной декомпозиции, анализируются с помощью *рекуррентных отношений*. Пример рекуррентного отношения - определение функции факториала:

$$n! = n * (n-1)! \quad \text{при } n \geq 1$$

$$n! = 1 \quad \text{при } n = 0.$$

Рассмотрим алгоритм *сортировки путем выборки*. Для сортировки n целых чисел из набора сначала выделяется наибольшее число, а затем этот же алгоритм применяется к оставшимся $n-1$ числам.

```
PROCEDURE SelectionSort (VAR a: SortArray; n: INTEGER);
  (* рекурсивная сортировка массива a из n элементов
  путем выборки максимального *)
  VAR i: INTEGER;
  BEGIN
    IF n > 0 THEN
      BEGIN
        i := PositionOfMax(a, n); (* индекс наибольшего числа *)
        Swap(a[i], a[n]); (* перестановка элементов массива *)
        SelectionSort(a, n-1);
      END;
    END;
  END;
```

Процедура PositionOfMax имеет время работы $\Theta(n)$, а процедура Swap - постоянное время $\Theta(1)$. Время работы процедуры SelectionSort $T(n)$ выражается рекуррентным соотношением

$$T(n) = T(n-1) + an \quad \text{при } n \geq 1$$

$$T(n) = b \quad \text{при } n = 0$$

где a и b - константы. Преобразуем это выражение в замкнутую форму.

$$\begin{aligned} T(n) &= T(n-1) + an = T(n-2) + a(n-1) + an = T(n-3) + a(n-2) + a(n-1) + an = \\ &= T(0) + a + 2a + 3a + \dots + a(n-2) + a(n-1) + an = b + an(n-1)/2 = \\ &= a/2 \cdot n^2 + a/2 \cdot n + b. \end{aligned}$$

Следовательно $T(n) \in \Theta(n^2)$.

7. Алгоритмы «разделяй и властвуй»

Алгоритмическая парадигма «разделяй и властвуй» состоит в декомпозиции исходной задачи на две подзадачи примерно равного размера. В качестве примера рассмотрим алгоритм *сортировки слиянием*.

```

PROCEDURE MergeSort (VAR a: SortArray; n1,n2:INTEGER);
  (* рекурсивная сортировка части массива a из
элементов
      с n1 до n2 включительно методом слияния *)
VAR m: INTEGER;
BEGIN
  IF (n2-n1>0) THEN
    BEGIN
      m:=(n1+n2) DIV 2;
      MergeSort(a,n1,m); (* сортировка первого подмассива
*)
      MergeSort(a,m+1,n2); (* сортировка второго
подмассива *)
      Merge(a,n1,m,n2); (* слияние отсортиров.
подмассивов *)
    END;
  END;
END;

```

Процедура слияния в подобных алгоритмах играет основную роль.

```

PROCEDURE Merge (VAR a: SortArray; n1,m,n2:INTEGER);
  (* слияние двух отсортированных подмассивов массива
a,
      состоящих из элементов с n1 до m и с m+1 до n2
      *)
VAR aind,bind,cind: INTEGER;
    c: SortArray;
BEGIN
  aind:=n1; bind:=m+1; cind:=0;
  WHILE (aind<=m) AND (bind<=n2) DO
    BEGIN
      IF (a[aind]<a[bind]) THEN
        BEGIN c[cind]:= a[aind];
          INC(aind);
        END
      ELSE
        BEGIN c[cind]:= a[bind];
          INC(bind);
        END;
      INC(cind);
    END;
  WHILE (aind<=m) DO
    BEGIN c[cind]:= a[aind];
      INC(aind); INC(cind);
    END;
  WHILE (bind<=n2) DO
    BEGIN c[cind]:= a[bind];
      INC(bind); INC(cind);
    END;
  FOR aind:=n1 TO n2 DO a[aind]:=c[aind-n1];
END;

```

Предположим сначала, что количество n элементов в массиве является степенью числа 2, т.е. $n=2^m$. Время работы алгоритма сортировки слиянием $T(n)$ описывается рекуррентным соотношением

$$T(n) = 2T(n/2) + an \quad \text{при } n > 1$$

$$T(n) = b \quad \text{при } n = 1$$

Оценку для $T(n)$ в замкнутой форме дает следующая

Теорема. Существуют положительные константы c_1 и c_2 такие, что для всех n имеет место $T(n) \leq c_1 n \log n + c_2$.

Доказательство по индукции по степени m . Положим $c_1 = a + b$, $c_2 = b$.

При $n = 1$, т.е. $m = 0$ справедливость оценки очевидна.

Пусть при некотором $n = 2^m$ оценка верна. Рассмотрим случай $n = 2^{m+1}$.

Тогда

$$T(n) = 2T(n/2) + an \leq 2(c_1 n/2 \log n/2 + c_2) + an \leq c_1 n (\log n - 1) + c_2 + an = c_1 n \log n + c_2$$

Таким образом, мы получили верхнюю оценку времени работы алгоритма сортировки слиянием $T(n) \in O(n \log n)$.

Эта оценка легко обобщается на общий случай, когда число n не равно степени двойки. Пусть m - минимальное целое такое, что $n \leq n_m = 2^m$. Тогда

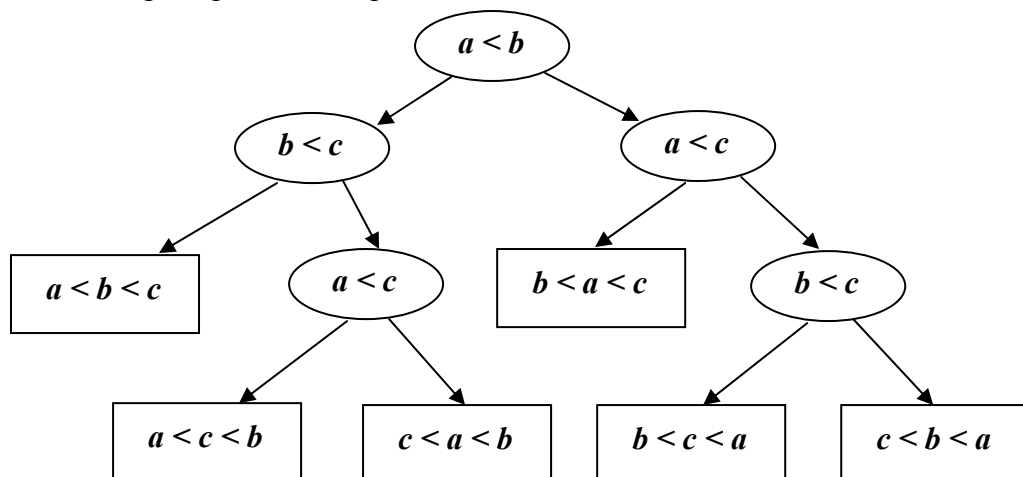
$$T(n) \leq T(n_m) \leq c_1 n_m \log n_m + c_2 \leq c_1 2n \log 2n + c_2 = 2c_1 n (\log n + 1) + c_2 \leq 2c_1 n (2 \log n) + c_2 = 4c_1 n \log n + c_2.$$

Положив новое значение $c_1 = 4(a + b)$, получаем требуемый результат.

8. Оценка сложности задачи

Сложность задачи определяется величиной необходимых и достаточных затрат времени на ее решение. При этом оценка сложности задачи осуществляется по всему множеству алгоритмов ее решения. Необходимое время решения определяется как нижняя оценка типа $\Omega(f(n))$, а достаточное время - как верхняя оценка типа $O(f(n))$. Соответственно, для того, чтобы получить оценку необходимого времени, нужно взять минимум времени по всем алгоритмам, а для того, чтобы получить оценку достаточного времени, нужно найти хотя бы один алгоритм, реализующий эту оценку. Получение верхних оценок показано в предыдущем разделе. Рассмотрим получение нижней оценки на примере задачи сортировки.

Алгоритм сортировки может рассматриваться как двоичное дерево решений. В дереве решений отражаются все возможные вычисления для всех



входов данного размера. Для каждого размера входа - свое дерево решений. Каждому внутреннему узлу соответствует одно сравнение. Две ветви, выходящие из узла, соответствуют ветвлению алгоритма в зависимости от результата сравнения. Листья дерева соответствуют окончательному результату.

Пример дерева решений для массива из трех чисел a, b, c представлен на рисунке.

В дереве решения каждый путь от корня к листу отображает возможные вычисления и длина пути равна числу сравнений при этом выполняемых. Следовательно, длина самого длинного пути из корня к листьям представляет собой количество сравнений в худшем случае.

Лемма. Двоичное дерево высоты h содержит не более 2^h листьев.

Доказательство. По индукции. При $h=0$ - один лист - утверждение справедливо.

Пусть при $h=k$ число листьев не превосходит 2^k . Рассмотрим дерево высоты $k+1$. У его корня имеются не более двух поддеревьев высоты k , в каждом из которых не более 2^k листьев (по индуктивному предположению). Следовательно, в дереве высоты $k+1$ не более, чем $2 \cdot 2^k = 2^{k+1}$ листьев, что и требовалось доказать.

Теорема. Высота любого дерева решений, упорядочивающего массив из n различных элементов, не меньше $\log n!$

Доказательство. Результатом упорядочивания массива из n различных элементов может быть любая из $n!$ перестановок. Следовательно, в дереве решений не менее $n!$ листьев. Но если бы высота h этого дерева была меньше чем $\log n!$, то согласно лемме число листьев было бы меньше, чем $2^h = 2^{\log n!} = n!$

Следствие. Любой алгоритм, упорядочивающий n чисел путем попарных сравнений, выполняет не менее $c \cdot n \cdot \log n$ сравнений при некотором $c > 0$ и всех n больше некоторого n_0 .

Доказательство.

Поскольку $n! \geq n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (\lceil n/2 \rceil) \geq (n/2)^{n/2}$,

а также $\log(n/2) = \log n - 1 \geq (\log n)/2$ при $n \geq 4$,

то $\log n! \geq \log(n/2)^{n/2} = n/2 \cdot \log(n/2) \geq n/4 \cdot \log n$ при $n \geq 4$.

Положив $c=1/4$, $n_0=4$, получаем требуемое утверждение.

Таким образом, мы получили нижнюю оценку сложности для задачи сортировки $T(n) \in \Omega(n \log n)$. Поскольку ранее в разделе 6 была получена верхняя оценка сложности $T(n) \in O(n \log n)$, тем самым доказано, что алгоритм сортировки слиянием является оптимальным и точная асимптотическая оценка сложности задачи сортировки $T(n) \in \Theta(n \log n)$.