Лекция 3. Временная сложность алгоритмов. Рекурсия и разделяй и властвуй

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Алгоритмы и структуры данных

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель и план лекции

Цель лекции

Научиться определять временную сложность циклов и рекурсивных функций и применять принцип «разделяй и властвуй» на примерах сортировки слиянием и умножения матриц.

План лекции

1. Временная сложность и порядок величины
2. Анализ циклов, фаз и нескольких переменных
3. Рекурсия и оценка её эффективности
4. Скорость роста функций на практике
5. Метод «разделяй и властвуй», рекуррентные соотношения
6. Сортировка слиянием, алгоритм Штрассена

Временная сложность

- Временная сложность алгоритма оценивает, сколько времени алгоритм будет использовать для некоторых входных данных. Идея состоит в том, чтобы представить эффективность как функцию, параметром которой является размер входных данных. Рассчитав временную сложность, мы можем узнать, достаточно ли быстр алгоритм без его реализации.

- Временная сложность алгоритма обозначается O(⋯), где три точки обозначают некоторую функцию. Обычно переменная n обозначает размер ввода. Например, если ввод представляет собой массив чисел, n будет размером массива, а если ввод представляет собой строку, n будет длиной строки.

Циклы

- Распространенная причина медленной работы алгоритма заключается в том, что он содержит много циклов, проходящих через входные данные. Чем больше вложенных циклов содержит алгоритм, тем он медленнее. Если имеется k вложенных циклов, временная сложность равна O(n^k). Например, временная сложность следующего кода равна O(n):

```
for (int i = 1; i <= n; i++) {
    // code
}
```

- временная сложность следующего кода равна O(n^2):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
```

Порядок величины

- Временная сложность не говорит нам точное количество раз выполнения кода внутри цикла, а только показывает порядок величины. В следующих примерах код внутри цикла выполняется 3n, n+5 и [n/2] раз, но временная сложность каждого кода равна O(n).

```
for (int i = 1; i <= 3*n; i++) {
    // code
}
```

```
for (int i = 1; i <= n+5; i++) {
    // code
}
```

```
for (int i = 1; i <= n; i += 2) {
    // code
}
```

- В качестве другого примера, временная сложность следующего кода равна O(n^2):

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // code
    }
}
```

Фазы

- Если алгоритм состоит из последовательных фаз, общая временная сложность равна наибольшей временной сложности одной фазы. Причина этого в том, что самая медленная фаза обычно является узким местом кода.
- Например, следующий код состоит из трех фаз с временными сложностями O(n), O(n^2) и O(n). Таким образом, общая временная сложность составляет O(n^2).

```
for (int i = 1; i <= n; i++) {
    // code
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
for (int i = 1; i <= n; i++) {
    // code
}
```

Несколько переменных

- Иногда временная сложность зависит от нескольких факторов. В этом случае формула временной сложности содержит несколько переменных.Например, временная сложность следующего кода равна O(nm):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // code
    }
}
```

Рекурсия

- Временная сложность рекурсивной функции зависит от количества вызовов функции и временной сложности одного вызова. Общая временная сложность является произведением этих значений.Например, рассмотрим следующую функцию:

```
void f(int n) {
    if (n == 1) return;
    f(n-1);
}
```

- Вызов f(n) вызывает n вызовов функций, и временная сложность каждого вызова составляет O(1). Таким образом, общая временная сложность равна O(n).

Рекурсия #2

- В качестве другого примера рассмотрим следующую функцию:

```
void g(int n) {
    if (n == 1) return;
    g(n-1);
    g(n-1);
}
```

- В этом случае каждый вызов функции порождает два других вызова, за исключением n = 1. Давайте посмотрим, что происходит, когда g вызывается с параметром n. В следующей таблице показаны вызовы функций, созданные этим единственным вызовом:

function call	number of calls
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
$\cdots$	$\cdots$
$g(1)$	2^{n-1}



$$1+2+4+\cdots+2^{n-1}=2^n-1=O(2^n).$$

Оценка эффективности

- Вычисляя временную сложность алгоритма, можно проверить перед его реализацией, что он достаточно эффективен для решения задачи. Отправной точкой для оценок является тот факт, что современный компьютер может выполнять несколько сотен миллионов операций в секунду.
- Например, предположим, что ограничение по времени для задачи составляет одну секунду, а размер входных данных равен n = 10^5 . Если временная сложность O(n^2), алгоритм выполнит примерно $(10^5)^2$ = (10^{10}) операций. Это должно занять как минимум несколько десятков секунд, поэтому алгоритм кажется слишком медленным для решения задачи.

Оценка эффективности #2

- С другой стороны, учитывая размер входных данных, мы можем попытаться угадать требуемую временную сложность алгоритма, решающего задачу. В следующей таблице приведены некоторые полезные оценки с учетом временного ограничения в одну секунду.

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

Заключение. Скорость роста

$n/f(n)$	$\lg n$	n	$n\lg n$	n^2	2^n	$n!$
10	0,003 мкс	0,01 мкс	0,033 мкс	0,1 мкс	1 мкс	3,63 мс
20	0,004 мкс	0,02 мкс	0,086 мкс	0,4 мкс	1 мс	77,1 лет
30	0,005 мкс	0,03 мкс	0,147 мкс	0,9 мкс	1 с	$8,4\times10^{15}$ лет
40	0,005 мкс	0,04 мкс	0,213 мкс	1,6 мкс	18,3 мин	
50	0,006 мкс	0,05 мкс	0,282 мкс	2,5 мкс	13 дней	
100	0,007 мкс	0,1 мкс	0,644 мкс	10 мкс	4×10^{13} лет	
1 000	0,010 мкс	1,00 мкс	9,966 мкс	1 мс		
10 000	0,013 мкс	10 мкс	130 мкс	100 мс		
100 000	0,017 мкс	0,10 мс	1,67 мс	10 с		
1 000 000	0,020 мкс	1 мс	19,93 мс	16,7 мин		
10 000 000	0,023 мкс	0,01 с	0,23 с	1,16 дней		
100 000 000	0,027 мкс	0,10 с	2,66 с	115,7 дней		
1 000 000 000	0,030 мкс	1 с	29,90 с	31,7 лет		

Алгоритм “Разделяй и властвуй”

1. **Разделение:** предполагает разделение проблемы на более мелкие подзадачи.
2. **Властвование:** решать подзадачи, вызывая их рекурсивно до тех пор, пока они не будут решены.
3. **Комбинирование:** объединить подзадачи, чтобы получить окончательное решение всей проблемы.

Умножение матриц

- Даны две квадратные матрицы A и B размера **NxN** каждая, найдите их матрицу умножения.
- ***Наивный метод*** : Ниже приведен простой способ умножения двух матриц.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Алгоритмы

- Быстрая сортировка
- Сортировка слиянием
- Ближайшая пара точек
- Алгоритм Штрассена
- Алгоритм быстрого преобразования Фурье (БПФ) Кули – Тьюки
- Алгоритм Карацубы для быстрого умножения

Что не относится к принципу «Разделяй и властвуй»:

- Двоичный поиск — это алгоритм поиска. На каждом этапе алгоритм сравнивает входной элемент x со значением среднего элемента массива. Если значения совпадают, верните индекс середины. В противном случае, если x меньше среднего элемента, алгоритм повторяется для левой стороны среднего элемента, в противном случае — для правой стороны среднего элемента. **Вопреки распространенному мнению, это не пример принципа «разделяй и властвуй», поскольку на каждом этапе существует только одна подзадача (разделяй и властвуй требует, чтобы было две или более подзадач**), и, следовательно, это случай «Decrease and Conquer».

- .

•

Divide And Conquer algorithm :

```
DAC(a, i, j)
{
    if(small(a, i, j))
      return(Solution(a, i, j))
    else
      mid = divide(a, i, j)               // f1(n)
      b = DAC(a, i, mid)                 // T(n/2)
      c = DAC(a, mid+1, j)              // T(n/2)
      d = combine(b, c)                   // f2(n)
   return(d)
}
```

Рекурентное соотношение

Recurrence Relation for DAC algorithm :

This is a recurrence relation for the above program.

```
            O(1) if n is small
T(n) =      f1(n) + 2T(n/2) + f2(n)
```

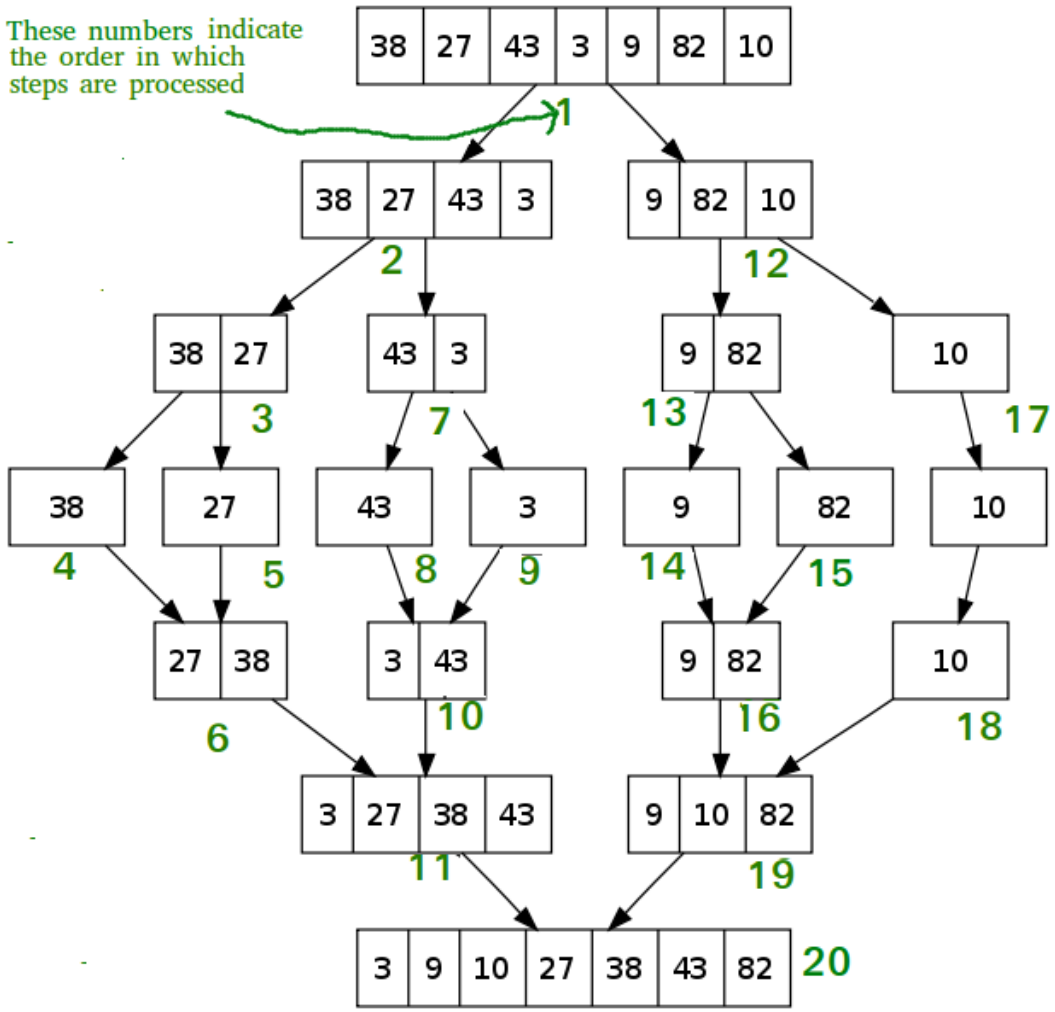
Пример

найти максимальный и минимальный элемент в заданном массиве.

```
Ввод: { 70, 250, 50, 80, 140, 12, 14 }
 Вывод: Минимальное число в данном массиве: 12
Максимальное число в данном массиве: 250
```

Подход: найти максимальный и минимальный элемент из заданного массива — это приложение по принципу «разделяй и властвуй». В этой задаче мы найдем максимальный и минимальный элементы в заданном массиве. В этой задаче мы используем подход «разделяй и властвуй» (DAC), который состоит из трех этапов: разделяй, властвуй и объединяй.

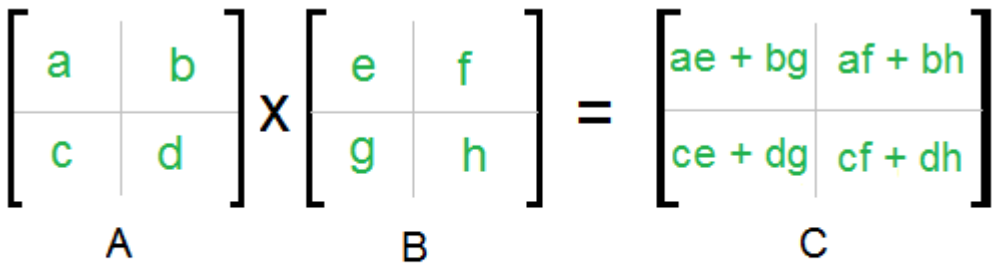
Сортировка слиянием (Merge sort)



Алгоритм Штрассена

- Алгоритм Штрассена — это эффективный алгоритм умножения двух матриц. Простой метод умножения двух матриц требует трех вложенных циклов и равен O(n^3). Алгоритм Штрассена умножает две матрицы за время O(n^2,8974).
- Ниже приведен простой метод «Разделяй и властвуй» для умножения двух квадратных матриц.

1. Разделите матрицы A и B на 4 подматрицы размера N/2 x N/2, как показано на диаграмме ниже.
2. Вычислите следующие значения рекурсивно. ae + bg, af + bh, ce + dg и cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Рекурентные соотношения

Рука об руку с парадигмой "разделяй и властвуй" идут рекуррентные соотношения, поскольку они предоставляют естественный способ описания времени работы соответствующих алгоритмов. ***Рекуррентное соотношение*** представляет собой уравнение или неравенство, которое описывает функцию через ее значения для меньших аргументов. Например, в разделе 2.3.2 мы описывали время $T(n)$ работы процедуры MERGE-SORT в наихудшем случае с помощью рекуррентного соотношения

$$T(n) = \begin{cases} \Theta(1) , & \text{если } n = 1 , \\ 2T(n/2) + \Theta(n) , & \text{если } n > 1 , \end{cases} \tag{4.1}$$

решением которого является функция $T(n) = \Theta(n \lg n)$.

Рекуррентные соотношения могут принимать множество разных форм. Например, рекурсивный алгоритм может делить задачу на подзадачи разного размера, например, разбивая на части, представляющие собой $2/3$ и $1/3$ исходной задачи. Если при этом разделение и комбинирование выполняются за линейное время, время работы такого алгоритма определяется рекуррентным соотношением $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Выводы

1. Временная сложность выражает время работы как функцию размера входа и позволяет оценить алгоритм без его реализации.
2. Сложность цикла определяется числом итераций; сложности вложенных циклов перемножаются, а при нескольких параметрах формула содержит несколько переменных.
3. Сложность рекурсии равна числу вызовов, умноженному на сложность одного вызова, и описывается рекуррентным соотношением.
4. Алгоритмы со временем $n!$ и 2^n практичны лишь для малых n, а $O(n)$ и $O(\log n)$ применимы к огромным объёмам данных.
5. Метод «разделяй и властвуй» разбивает задачу на подзадачи и объединяет их решения (сортировка слиянием, алгоритм Штрассена).

Контрольные вопросы

1. Что такое временная сложность алгоритма и как она обозначается?
2. Как определить сложность вложенных циклов? Приведите примеры $O(n^2)$ и $O(nm)$.
3. Как оценивается сложность рекурсивной функции? Зачем нужно рекуррентное соотношение?
4. При каких значениях n алгоритмы со сложностью $n!$ и 2^n становятся непрактичными?
5. Назовите этапы метода «разделяй и властвуй».
6. Какова сложность сортировки слиянием? Чем алгоритм Штрассена отличается от обычного умножения матриц?

Список литературы

Основная литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 3-е изд. — М.: Вильямс, 2013.
2. Скиена С. Алгоритмы. Руководство по разработке. 2-е изд. — СПб.: БХВ-Петербург, 2011.
3. Левитин А. Алгоритмы: введение в разработку и анализ. — М.: Вильямс, 2006.

Дополнительная литература

4. Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы. — М.: МЦНМО, 2014.
5. Aho A.V., Hopcroft J.E., Ullman J.D. Data Structures and Algorithms. — Addison-Wesley, 1983.
6. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. 4th ed. — MIT Press, 2022.

Электронные ресурсы

7. MIT OpenCourseWare, курс 6.006 «Introduction to Algorithms»: ocw.mit.edu