Алгоритмы и структуры данных

Зариковская Наталья Вячеславовна

Лекция 2

Теоретические сведения

- **Сортировка -** это процесс расстановки элементов в некотором порядке. Сортировка двух записей состоит из сравнения их ключевых полей и определения, которое из них меньше. После этого записи переставляются так, что запись с меньшим ключом ставится перед записью с большим ключом. При рассмотрении алгоритмов сортировки для простоты будем предполагать, что записи состоят только из одного поля, что не ограничивает область применения этих алгоритмов.
- Методы сортировки делятся на ***внутренние*** и ***внешние.***
- Внутренние методы предполагают, что сортируемые данные целиком располагаются в оперативной памяти.
- Внешние методы используются для сортировки файлов данных, которые слишком велики, чтобы полностью поместиться в оперативной памяти.
-

Сортировка с помощью прямого обмена

- **Сортировка с помощью прямого обмена** (сортировка стандартным обменом, сортировка методом пузырька) перемещает один элемент массива S в соответствующую позицию при каждом просмотре. При первом просмотре каждый элемент S[i] сравнивается с элементом S[i+1] (1 <= i <= n-1) и при необходимости, если S[i]>S[i+1], меняется с ним местами. В результате наибольший элемент помещается в последнюю позицию. При втором просмотре выполняются те же действия, но уже не для n, а для (n-1) первых элементов массива. В результате следующий по величине элемент перемещается в предпоследнюю позицию. При третьем просмотре рассматриваются уже первые (n-2) элемента массива. Таким образом, для выполнения сортировки требуется максимально (n-1) просмотров. Во время каждого просмотра необходимо фиксировать наличие обменов. Если при очередном просмотре обменов не было, то массив уже упорядочен, сортировка заканчивается.
- **Алгоритм сортировки прямым обменом** состоит из следующих шагов.
- 1. t='Истина'; j=n-1.
- 2. Если t - 'Ложь', то 'Массив отсортирован'. Закончить.
- 3. t='Ложь'.
- 4. i=1.
- 5. Если s[i]>s[i+1], то t='Истина' и поменять местами s[i] и s[i+1]
- 6. i=i+1.
- 7. Если i<=j, то на шаг 5.
- 8. j=j-1. На шаг 2.
-

Сортировка с помощью прямого обмена

Анализ

В приведенном выше алгоритме переменные имеют следующие назначения:

- t - признак отсортированности массива (признак окончания сортировки);
- j - определяет количество (j+1) сортируемых в данном проходе элементов;
- i - определяет сравниваемые элементы (s[i] и s[i+1]).

Рассматриваемый алгоритм отличается от большинства алгоритмов сортировки тем, что он "замечает" отсортированность массива. Для этого используется логическая переменная (в рассмотренном алгоритме - переменная t), которая принимает значение "Истина", если при очередном проходе была хотя бы одна перестановка. Если же перестановок не было, то массив уже отсортирован, и переменная t будет иметь значение "Ложь".

- Число сравнений:
- минимальное - (n-1),
- среднее - $((n^2)/2) - (3n/4)$,
- максимальное - n*(n-1)/2.
- Число обменов:
- минимальное - 0,
- среднее - $(n^2/4)$,
- максимальное - $(n^2/2)$.

Следовательно, алгоритм сортировки методом прямого обмена имеет сложность $O(n^2)$.

-

Сортировка с помощью прямого выбора

Сортировка с помощью прямого выбора включает в себя следующие шаги.

- 1. Среди элементов массива S[1], ..., S[n] выбрать элемент с наибольшим значением.
- 2. Найденный элемент поменять местами с элементом S[n].
- 3. Выполнить шаги 1 и 2 для оставшихся n-1 элементов, n-2 элементов и т.д. до тех пор, пока не останется один, самый маленький элемент.
-

Алгоритм сортировки прямым выбором состоит из следующих шагов.

- 1. j=n.
- 2. M=S[1]; k=1; i=2.
- 3. Если M<S[i], то M=S[i]; k=i.
- 4. i=i+1.
- 5. Если i<=j, то на шаг 3.
- 6. Иначе, поменять местами S[i] и S[k].
- 7. j=j-1.
- 8. Если j>1, то на шаг 2.
- 9. Иначе, "Массив отсортирован".
-

Сортировка с помощью прямого выбора

- В приведенном выше алгоритме переменные имеют следующие назначения:
- j - определяет количество сортируемых в данном проходе элементов;
- M - максимальный элемент;
- k - индекс максимального элемента.
-
- Число сравнений:
- среднее - $(n^2 - n)/2$
- Число обменов:
- минимальное – $3*(n - 1)$
- среднее – $(n - 1)$
- максимальное – $n^2/4+3*(n – 1)$
-
- **Анализ**
- Алгоритм сортировки прямым выбором "не замечает" отсортированности массива. Поэтому количество просмотров всегда постоянно и равно (n - 1). Если сравнивать быстродействие алгоритмов прямого обмена и прямого выбора, то последний в среднем работает быстрее. Это объясняется тем, что количество сравнений в обоих алгоритмах одинаково. Среднее количество перестановок в алгоритме прямого обмена - $(n^2)/4$. Количество перестановок в алгоритме прямого выбора равно (n - 1), что значительно меньше. Алгоритм сортировки методом прямого выбора имеет сложность $O(n^2)$.
-

Сортировка с помощью прямого включения

- **Сортировка с помощью прямого включения** (сортировка вставками) основана на последовательной вставке элементов в уже упорядоченную последовательность. Алгоритм сортировки заключается в следующем. Сначала упорядоченным считается один, первый элемент. Второй элемент либо меняется местами с первым, либо остается на своем месте. Далее 3-й элемент включается в нужное место уже упорядоченной последовательности из 2-х элементов, за ним 4-й и т.д. до n-го. Пусть необходимо вставить i-й (i>=2) элемент в уже упорядоченную последовательность из i-1 элементов. Элемент S[i] последовательно сравнивается с каждым элементом S[k] (0<=k<=i-1) и, либо вставляется на свободное место, если S[i]>=S[k], либо элемент S[k] сдвигается на одну позицию вправо и процесс выполняется для элемента S[k-1]. Весь процесс вставки элемента заканчивается либо когда найден первый элемент S[k] такой, что S[k]<=S[i], либо достигнута левая граница упорядоченной последовательности (элемент S[i] меньше всех элементов упорядоченной последовательности). Очевидным улучшением описанного выше процесса является установка барьера в нулевом элементе массива S (S[0]=S[i]), что позволит избежать проверки на выход за левую границу массива.

Сортировка с помощью прямого включения

- **Алгоритм сортировки прямым включением** состоит из следующих шагов.
1. i=2.
2. S[0]=S[i].
3. j=i-1.
4. Если S[0]>=S[j], то на шаг 7.
5. S[j+1]=S[j].
6. j=j-1. На шаг 4.
7. S[j+1]=S[0].
8. i=i+1. Если i<=n, то на шаг 2.
9. Иначе, "Массив отсортирован".

В приведенном выше алгоритме переменные имеют следующие назначения:

S[i] - вставляемый элемент;

S[0] - "барьер".

Сортировка с помощью прямого включения

- Число сравнений:
- минимальное - (n – 1)
- среднее - $(n^2 + n - 2)/4$
- максимальное – $(n^2 + n - 4)/4$
- Число обменов:
- минимальное – 3*(n-1)
- среднее – $(n^2 + 9n - 10)/4$
- максимальное – $(n^2 + 3n - 4)/2$
-
- **Анализ**
- Алгоритм сортировки прямым включением "не замечает" отсортированности массива. Поэтому количество просмотров всегда постоянно и равно (n - 1). Если сравнивать быстродействие алгоритмов прямого включения и прямого выбора, то первый в среднем работает быстрее. Это объясняется тем, что при примерно равном количестве перестановок количество сравнений в алгоритме сортировки прямым
- включением в среднем в два раза меньше. Алгоритм сортировки методом прямого выбора имеет сложность $O(n^2)$.
-
-

Быстрая сортировка

- Если количество элементов в массиве не многим меньше максимального их значения, то в данном случае наиболее эффективным и по быстродействию, и по простоте Основные достоинства этого алгоритма состоят в том, что он точечный (использует лишь небольшой дополнительный стек), в среднем требует только около N log N операций для того, чтобы отсортировать N элементов, и имеет экстремально короткий внутренний цикл. Недостатки алгоритма состоят в том, что он рекурсивен (реализация очень затруднена когда рекурсия недоступна), в худшем случае он требует N2 операций, кроме того он очень "хрупок": небольшая ошибка в реализации, которая легко может пройти незамеченной, может привести к тому, что алгоритм будет работать очень плохо на некоторых файлах.
-
-
-

Быстрая сортировка

- **Алгоритм реализуется при помощи рекурсивных вызов**, поэтому
- зададим процедуру Сортировка(iLo, iHi).
- Она реализует следующие шаги:
-
- 1. Lo=iLo, Hi=iHi, Mid=A[(Lo+Hi) div 2];
- 2. Если A[Lo] < Mid, то Lo=Lo+1, на шаг 2
- 3. Если A[Hi] > Mid, то Hi=Hi-1, на шаг 3
- 4. Если Lo > Hi, на шаг 8
- 5. Поменять местами A[Lo] и A[Hi]
- 6. Lo=Lo+1, Hi=Hi-1
- 7. Если Lo <= Hi, на шаг 3
- 8. Если Hi > iLo, Вывов Сортировка(A,iLo,Hi)
- 9. Если Lo < iHi, Вывов Сортировка(A,Lo,iHi)
- 10. Возврат (на предыдущий уровень рекурсии)
-
- **Анализ**
- Данный алгоритм является наиболее быстрым из известных и имеет несколько модификаций. Однако средняя скорость его работы определяется выражением O(N * lgN).

Сортировка Шелла

- Сортировка вставками не относится к категории быстродействующих, поскольку единственный вид операции обмена, который она использует, выполняется над двумя соседними элементами, в связи с чем элемент может передвигаться вдоль массив лишь на одно место за один раз. Например, если элемент с наименьшим значением ключа оказывается в конце массива, потребуется сделать *N* шагов, чтобы поместить его в надлежащее место. Сортировка методом Шелла представляет собой простейшее расширение метода вставок, быстродействие которого выше за счет обеспечения возможности обмена местами элементов, которые находятся далеко один от другого.
- Возникает вопрос: какую последовательность шагов следует использовать? В общем случае на этот вопрос трудно найти правильный ответ. В литературе опубликованы результаты исследований различных последовательностей шагов; некоторые из них хорошо зарекомендовали себя на практике, однако наилучшую последовательность, по-видимому, отыскать не удалось. В общем случае на практике используются убывающие последовательности шагов, близкие к геометрической прогрессии в результате чего число шагов находится в логарифмической зависимости от размеров файлов. Например, если размер следующего шага равен примерно половине предыдущего, то для сортировки файла, состоящего из 1 миллиона элементов, потребуется примерно 20 шагов, если же такое соотношение примерно равно одной четвертой, то достаточно будет 10 шагов. Использование как можно меньшего числа шагов — это весьма важное требование.
- Практический результат от обнаружения хорошей последовательности шагов, по-видимому, ограничен повышением быстродействия алгоритма на 25%, в то время сама проблема представляет собой довольно таки увлекательную головоломку
-

Сортировка Шелла

- Последовательность шагов 1 4 13 40 121 364 093 3280 9841 ... .
- Она просто вычисляется (начав с 1, получить значение следующего шага, множив предыдущее значение на 3 и добавив 1) и обеспечивает реализацию сравнительно эффективной сортировки даже в случае относительно больших файлов.
- Многие другие последовательности шагов позволяет получить еще более эффективную сортировку, однако довольно трудно превзойти эффективность более чем на 20% даже в случае сравнительно больших значений N.
- Одной из таких последовательностей является 1 8 23 77 281 1073 193 16577 ..., т.е. последовательность 4i+1 +3*2i+ для i > 0. Можно доказать, что приведенная последовательность обеспечивает повышенное быстродействие для самых трудных случаев сортировки.
- С другой стороны, существуют и плохие последовательности шагов: например,

  1 2 4 8 16 32 64 128 256 512 1024 2048 ...

(первая последовательность шагов, предложенная Шеллом еще в 1959 г. скорее всего, служит причиной низкой эффективности сортировки, поскольку элементы на нечетных позициях не сравниваются с элементами на четных позициях вплоть до последнего прохода.

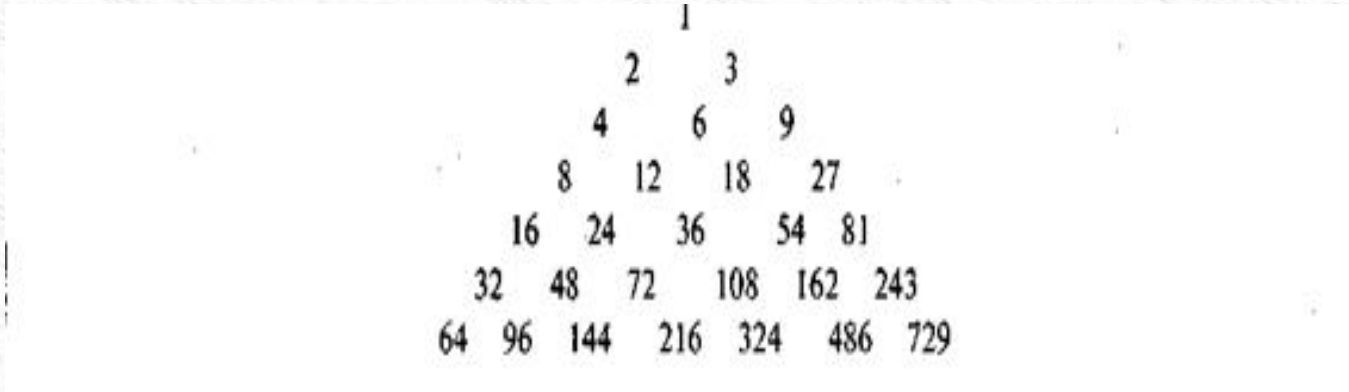
- Этот эффект заметен на файлах с произвольной организацией, и он становится катастрофическим в наихудших случаях: эффективность метода резко снижается и время выполнения сортировки становится пропорциональным квадрату N, если, например, половина элементов файла с меньшими значениями находится в четных позициях, а другая половина элементов (с большими значениями) — в нечетных позициях

Сортировка Шелла

- **Свойство 1.** *Сортировка методом Шелла выполняет менее N(h — l)(k — l)/g операций сравнения при g-сортировке h- и k-упорядоченного файла при условии, что h и k взаимно просты.*
- **Свойство 2.** *Сортировка методом Шелла выполняет менее O ($N^{3/2}$) операций сравнения для последовательности шагов* **1 4 13 40 121 364 1093 3280 9841...**
- Для больших шагов, когда имеются *h* подфайлов размером *N/h,* в наихудшем случае расходы составляют примерно N^2/h. При малых шагах из свойства 1 следует, что стоимость составляет приблизительно *Nh.* Все зависит от того, насколько успешно удается вписаться в эти границы на каждом шаге. Это справедливо для каждой относительно простой последовательности, возрастающей экспоненциально.
- **Свойство 3.** *Сортировка методом Шелла выполняет менее O ($N^{4/3}$) операций сравнения для последовательности шагов* **1 8 23 77 281 1073 4193 16577...**
- Последовательности шагов, которые рассматривались до сих пор, эффективны в силу того, что следующие один за другим элементы последовательности взаимно просты. Другое семейство последовательностей шагов эффективно именно благодаря тому, что такие элементы *не являются* взаимно простыми.

Сортировка Шелла

- **<u>Свойство 4</u>.** *Сортировка методом Шелла выполняет менее 0(N(* $\log N)^2$) *операций сравнения для последовательности* шагов **1 2 3 4 6 9 8 12 18 27 16 24 36 54 81...**
- Рассмотрим треугольник, составленный из шагов, в котором каждое число в два раза больше, чем число выше и правее, и в три раза больше, чем число выше и, если мы используем эти числа снизу вверх и справа налево как последовательность шагов в рамках сортировки методом Шелла, то каждому шагу *x* в нижнем ряду предшествуют значения *2x* и *3x,* так что каждый подфайл оказывается 2-упорядочен и 3-упорядочен, при этом ни один элемент не передвигается больше, чем на одну позицию в процессе всей сортировки!
- Число шагов из треугольника, которое меньше N по величине, и подавно будет меньше $(\text{Log}_2 N)^2$



Эмпирические исследования последовательностей шагов сортировки методом Шелла

- Сортировка методом Шелла выполняется в несколько раз быстрее по сравнению с другими элементарными методами сортировки даже в тех случаях, когда шаги являются степенями 2, в то же время некоторые специальные виды последовательностей шагов позволяют увеличить ее быстродействие в 5 и более раз. Три лучших последовательности, приведенные в данной таблице, существенно различаются по положенным в их основу принципам. Сортировка методом Шелла вполне пригодна для практических приложений даже в случае файлов больших размеров. По эффективности она намного превосходит методы выбора и вставок, равно как и пузырьковую сортировку

Рост функций

- Для большинства алгоритмов *главным параметром (primary parameter)* является N, который оказывает существенное влияние на время их выполнения. Параметр N может быть степенью полинома, размером файла при сортировке или поиске, количеством символов в строке или некоторой другой абстрактной мерой размера рассматриваемой задачи: чаще всего, он прямо пропорционален величине обрабатываемого набора данных. Когда таких параметров существует более одного, мы часто сводим анализ к одному параметру, задавая его как функцию от других параметров или рассматривая одновременно только один параметр (считая остальные постоянными). Таким образом, мы ограничиваем себя рассмотрением только одного параметра *N* без потери общности. Нашей целью является выражение требований к ресурсам, предъявляемых разрабатываемыми нами программами (как правило, это время выполнения) в зависимости от *N* с использованием максимально простых математических формул, которые обеспечивают точность расчетов для больших значений параметров. Алгоритмы, изучаемые нами, обычно имеют время выполнения, пропорциональное одной из следующих функций:

- **1** Большинство инструкций большинства программ выполняется один или максимум несколько раз. Если все инструкции программы обладают этим свойством, мы говорим, что время выполнения программы *постоянно (constant).*

Рост функций

- **log N** Когда время выполнения программы описывается *логарифмической (logarithmic)* зависимостью, программа немного утрачивает быстродействие с ростом *N.* Такое время выполнения обычно характерно для программ, которые сводят крупную задачу к некоторой последовательности задач меньшего размера, уменьшая на каждом шаге размер задачи на некоторую небольшую часть. В интересующем нас диапазоне мы будем рассматривать время выполнения как величину, не превосходящую некоторое большое постоянное значение. Основание логарифма изменяет это значение, но ненамного:
- когда *N* — тысяча, log *N* равно 3, если основание равно 10, либо примерно 10, если основание равно 2; когда *N* равно миллиону, значения log N всего лишь удвоится. При удвоении N значение *logN* возрастет на постоянную величину, а удваивается лишь, когда *N* увеличится до N^2.

- **N** Когда время выполнения программы *линейно (linear),* это обычно означает, что каждый элемент ввода подвергается небольшой обработке. Когда *N* равно миллиону, такого же порядка и время выполнения алгоритма. Когда *N* удваивается, то же происходит и со временем выполнения. Эта ситуация оптимальна для алгоритма, который должен обработать *N* вводов (или произвести *N* выводов).

- ***NlogN*** Время выполнения, пропорциональное .N log N имеет место, когда алгоритм решает задачу, разбивая ее на подзадачи меньших размеров, решая их независимо и затем объединяя решения. Из-за отсутствия подходящего прилагательного *("линерифмический" "linerithmic")* мы просто говорим, что время выполнения такого алгоритма равно *N* log N. Когда *N* равно 1 миллион, N log N возрастает примерно до 20 миллионов. Когда *N* удваивается, то время выполнения возрастает более чем вдвое (но не намного более).

Рост функций

- **N^2** Когда время выполнения алгоритма является *квадратичным (quadratic),* он полезен для практического использования применительно к небольшим задачам. Квадратичное время выполнения обычно характерно для алгоритмов, которые обрабатывают все элементы данных парами (возможно, в цикле двойного уровня вложения). Когда N равно одной тысяче, время выполнения равно одному миллиону. Когда *N* удваивается, время выполнения увеличивается в четыре раза.
-
- **N^3** Аналогичный алгоритм, обрабатывающий элементы данных тройками (возможно, в цикле тройного уровня вложения), имеет *кубическое (cubic)* время выполнения и практически применим лишь для решения малых задач. Когда *N* равно 100, время выполнения равно 1 миллиону. Когда *N* удваивается, время выполнения увеличивается в восемь раз.
-
- ***2^N*** Лишь немногие алгоритмы с *экспоненциальным (exponential)* временем выполнения имеют практическое применение, хотя такие алгоритмы возникают естественным образом при попытках решения задачи "в лоб". Когда *N* равно 20, время выполнения равно 1 миллиону.
- Когда *N* удваивается, время выполнения увеличивается в четыре раза!

Рост функций

- Время выполнения конкретной программы, скорее всего, будет некоторой константой, умноженной на одно из перечисленных выше выражений *(главный член — leading term)* плюс некоторые слагаемые меньшего порядка. Значения постоянного коэффициента и остальных слагаемых зависят от результатов анализа и деталей реализации. В грубом приближении коэффициент при главном члене связан с количеством инструкций во внутреннем цикле: на любом уровне разработки алгоритма разумно сократить количество таких инструкций. Для больших *N* доминирует главный член, для малых N или в случае тщательно разработанных алгоритмов свой вклад вносят и другие слагаемые, поэтому сравнение алгоритмов становятся более сложным. В большинстве случаев мы будем называть время выполнения программ просто "линейным", "кубическим" и т.д.
- В итоге, чтобы уменьшить общее время выполнения программы, мы минимизируем количество инструкций во внутреннем цикле. Каждую инструкцию, необходимо подвергнуть исследованию: нужна ли она вообще? Существует ли более эффективный способ выполнить ту же задачу? Некоторые программисты считают, что автоматические инструменты, содержащиеся в современных компиляторах, могут создавать наилучший машинный код; другие утверждают, что наилучшим способом является написание внутренних циклов вручную на машинном языке, или ассемблере. Как правило, мы будем воздерживаться от рассмотрения вопросов оптимизации на таком уровне, хотя время от времени будем указывать, сколько машинных инструкций требуется для выполнения определенных операций, чтобы показать, почему на практике одни алгоритмы могут оказаться быстрее других.

Рост функций

- **секунды**
- 10^2 1,7 минуты
- 10^4 2,8 часа
- 10^5 1,1 дня
- 10^6 1,6 недели
- 10^7 3,8 месяца
- 10^8 3,1 года
- 10^9 3,1 десятилетия
- 10^{10} 3,1 столетия
- 10^{11} никогда
- Перевод секунд: Огромная разница между такими числами, как 10^4 и 10^8, становится более очевидной, когда мы применяем их для измерения промежутков времени, а затем переводим в привычные единицы измерения времени;(2^{10} примерно равно 10^3 то этой таблицей можно воспользоваться и для перевода степеней 2 в привычные единицы времени; н-р 2^{32} секунд составляет примерно 124 года).
- Для многих приложений нашим единственным шансом решить крупную задачу остается использование эффективного алгоритма. В этой таблице показано минимальное количество времени, необходимое для решения задач размером 1 миллион и 1 миллиард с использованием линейных алгоритмов, алгоритмов с зависимостью $N \log N$ и квадратичных алгоритмов на компьютерах с быстродействием 1 миллион, ! миллиард и 1 триллион операций в секунду. Быстрый алгоритм помогает существенно ускорить решение задачи на медленной машине, однако быстрая машина не сможет выручить, когда используется медленный алгоритм.

Рост функций

Операций в секунду	Размер задач 1 миллион			Размер задачи 1 миллиард		
	N	N log N	N^2	N	N log N	N^2
10^6	секунд	секунд	недель	часов	часов	никогда
10^9	мгновенно	мгновенно	часов	секунд	секунд	десятилетий
10^{12}	мгновенно	мгновенно	секунд	мгновенно	мгновенно	недель

- В этой таблице сравниваются значения, принимаемые рядом функций, с которыми нам придется часто сталкиваться при анализе алгоритмов. Очевидно, доминирующей является квадратичная функция, особенно на больших значениях N, а на малых значениях N различие между функциями оказываются не такими, как можно было бы ожидать. Например, $N^{3/2}$ больше, чем $N \lg^2 N$, на очень больших значениях N, однако на небольших N наблюдается обратная картина. Точное время выполнения алгоритма может быть выражено в виде линейной комбинации этих функций. Мы можем легко отделить быстрые алгоритмы от медленных из-за огромной разницы, например, междуlg N и *N* или N и N^2, тем не менее, различия между двумя быстрыми алгоритмами может потребовать более тщательных исследований.

lg N	√N	N	N lg N	$N (\lg N)^2$	$N^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	100000	100000000
17	316	100000	1990964	27588016	31622777	10000000000
20	1000	1000000	19931569	397267426	1000000000	1000000000000

Рост функций

- При анализе алгоритмов можно воспользоваться еще несколькими функциями. Например, алгоритм с N^2 входными данными, имеющий время выполнения N^3, лучше рассматривать как алгоритм с зависимостью $N^{3/2}$. Кроме того, алгоритмы, допускающие разбиение на две подзадачи, имеют время выполнения, пропорциональное $N \log^2 N$ Из первой таблицы очевидно, что обе эти функции ближе к $N \log N$, нежели к N^2.
- Логарифмическая функция играет особую роль при разработке и анализе алгоритмов, поэтому ее стоит рассмотреть подробнее. Поскольку нам часто приходится давать оценку аналитическим результатам, в которых опущен постоянный множитель, мы будем пользоваться записью *"logN",* опуская основание. Изменение основания логарифма с одной константы на другую меняет значение логарифма лишь на постоянный множитель, однако в определенных контекстах мы используем конкретные значения оснований логарифмов. В математике настолько важным является понятие *натуральный логарифм (natural logarithm)* с основанием $e = 2.71828...$, что широкое распространение получило следующее сокращение: $\log_e N = \ln N$. В вычислительной технике очень важен *двоичный логарифм (binary logarithm)* (т.е. по основанию 2), поэтому используется сокращение $\log_2 N = \lg N$.
- Иногда нам приходится вычислять логарифмы, особенно в отношении больших чисел. Например, lg lg 2^{256} = lg 256 = 8. Из этого примера должно быть понятно, что в большинстве практических случаев lg *lgN* рассматривается как константа, поскольку значение этого выражения достаточно мало даже для очень больших *N.*
- .

Рекомендации для выбора алгоритма сортировки

- Если сортируется небольшой объем данных (N < 100), то рекомендуется выбирать простой метод сортировки, так как сложные алгоритмы занимают больший размер кода и не эффективны при малом количестве сортируемых элементов.
- Если сортируются элементы большого размера, то рекомендуется использовать специальные таблицы с помощью которых осуществляется доступ к самим элементам (например, это могут быть указатели на элементы или их порядковые номера), причем ключ по которому сортируются данные может входить или не входить в данные такой таблицы. После сортировки таблицы можно или переставить исходные данные по таблице, или оставить все как есть и осуществлять дальнейший доступ к данным по уже отсортированной таблице.
- Если сортируются данные в файле, то необходимо учитывать, что большая часть времени будет тратиться на чтение, запись элемента и перемещение по файлу. В такой ситуации методы вставок являются не эффективными, так как требуют большое число перестановок. (Кстати, при внутренней сортировке время сравнения и перестановки двух элементов практически одинаковое.) Так же возможно считывать небольшие участки данных в память, там их сортировать и записывать обратно в память, после чего файл будет уже частично отсортирован и останется довести дело до конца каким-либо методом, предпочитающим частично отсортированные данные.
- Если используются структуры данных, отличные организацией доступа от массива, то необходимо выбирать метод сортировки наиболее подходящий для данной структуры. Например, если сортируется связный список, то для него будет эффективен метод вставок. Надо преобразовать соответствующий алгоритм таким образом, чтобы вначале находилось место для вставки очередного элемента, а затем уже вставлять его (в наших примерах на базе массива приходилось многократно переставлять элементы, что на самом деле не эффективно).
- Рекомендуется выбирать алгоритм сортировки с учетом предполагаемого входного состояния данных (является таблица частично отсортированной, отсортированной в обратном порядке и т.д.) и рекомендаций приведенных почти к каждому алгоритму.
- Если после сортировки надо получить новую таблицу содержащую отсортированные данные (имеется ввиду, что в памяти хранится две таблица: исходная и отсортированная), то неплохие результаты могут дать алгоритмы основанные на выборках элементов.
- .