Глава 1
ВВЕДЕНИЕ

> Это сложные вещи, но не нужно быть гением,
> чтобы ими пользоваться.
>
> *Джек Нунен, президент корпорации SPSS*

Эта книга посвящена *алгоритмам*. Чтобы глубоко почувствовать их красоту и осознать все сложности их разработки, первым делом следует понять, что же это такое. Оксфордский словарь английского языка сообщает, что алгоритмом называется *«обычно записываемый в алгебраической нотации процесс или набор правил, который в настоящее время используется, в частности, в вычислениях, машинном переводе и лингвистике»*. Современное значение этого термина сходно со значением слов *«рецепт»*, *«метод»*, *«процедура»* или *«функция»*, но в информатике ему дается более точное определение. За последние 200 лет множество авторитетных исследователей пытались закрепить значение этого термина, предлагая все более сложные и подробные определения, считая их более точными и элегантными[1]. Мы, как проектировщики алгоритмов и инженеры, будем пользоваться определением Дональда Кнута, сформулированным в конце 1960-х годов [7][2], согласно которому алгоритмом называется *«конечная, определенная, эффективная процедура, дающая какой-то результат»*. Эти интуитивно понятные характеристики широко принимаются в качестве требований к последовательности составляющих алгоритм шагов. Но они настолько глубоки, что следует рассмотреть каждую из них более подробно. Это наглядно продемонстрирует не только сценарии и проблемы, возникающие при проектировании и разработке алгоритмов, но и цель написания этой книги.

- **Конечность.** «Алгоритм всегда должен завершаться после конечного числа шагов... разумно конечного числа». Очевидно, что уточняющее наречие «разумно» связано с *эффективностью* алгоритма, ведь, по утверждению Кнута [7], «на практике требуются не просто алгоритмы, а годные алгоритмы». Годность зависит от того, как алгоритм поступает с такими ценными *вычислительными ресурсами*, как время работы процессора, используемая память, загрузка сети и подсистемы ввода-вывода, потребляемая энергия, а также от простоты и элегантности самого алгоритма. Последнее влияет на величину затрат на написание кода, его отладку и поддержку.

---

[1] См. статью «Характеристики алгоритмов»: https://en.wikipedia.org/wiki/Algorithm_characterizations.

[2] Список литературы, упоминаемой в тексте главы, приводится в ее конце. — *Примеч. ред.*

- **Определенность.** «Каждый шаг алгоритма должен быть строго определен путем досконального и однозначного указания всех действий, которые необходимо выполнить для каждого случая». Продемонстрировать этот принцип Кнут попытался, описав машинный язык для своего гипотетического компьютера MIX. Сегодня существует множество языков программирования, таких как C/C++, Java, Python и т. д. Для каждого из них предусмотрен набор команд, позволяющих программистам однозначно описывать лежащие в основе алгоритмов процедуры. Однозначность в данном случае обеспечивается формальной семантикой, закрепленной за каждой командой. Именно поэтому любой читающий алгоритм человек сможет правильно его интерпретировать.
- **Эффективность.** «Все составляющие алгоритм операции должны быть достаточно простыми, чтобы их в принципе мог реализовать человек с помощью бумаги и карандаша». Поэтому упомянутое ранее понятие «шаг» подразумевает необходимость полностью и глубоко понимать решаемую задачу и четко определять пошаговую структуру решения.
- **Процедура.** «Логически упорядоченная последовательность шагов».
- **Входные данные.** «Параметры, которые задаются перед началом работы алгоритма. Они берутся из заданных наборов объектов». Соответственно, поведение алгоритма не уникально, а зависит от предназначенных для обработки наборов объектов.
- **Выходные данные.** «Величины, определенным образом связанные с входными данными». По сути, это ответ, возвращаемый алгоритмом для заданных входных данных.

В этой книге я постарался избежать формального подхода к описанию алгоритмов, так как мне хотелось сосредоточиться на элегантных в теории и эффективных на практике идеях, составляющих основу алгоритмического решения некоторых интересных задач. При этом крайне важно было не завязнуть во множестве технических деталей программирования. Поэтому каждая глава посвящена разбору интересной задачи, порожденной каким-то практическим сценарием. Я буду предлагать решения все большей сложности и повышенной эффективности, стараясь, чтобы это не увеличивало сложность описания алгоритма. Я старался выбирать задачи, допускающие удивительно элегантные решения, которые можно записать несколькими строками кода. Поэтому предпочтение было отдано принятой в настоящее время практике проектирования алгоритмов, когда их поясняют на словах или с помощью имитирующего наиболее известные языки *псевдокода*. Но все описания будут достаточно строгими, чтобы соответствовать шести признакам Кнута.

Разумеется, *элегантность* будет не единственным критерием при проектировании алгоритмов — нашей целью станет еще и *эффективность*, как правило связанная с *временной и пространственной сложностью*. Традиционно временная сложность оценивается как функция от размера входных данных n путем подсчета максимального количества шагов $T(n)$, необходимого алгоритму для завершения работы. Поскольку при оценке учитывается максимальное по всем входным данным

размером n время работы алгоритма, говорят про *время в худшем случае*. Понятно, что чем больше n, тем больше будет $T(n)$, то есть это неубывающая положительная функция. Аналогично наихудшая пространственная сложность алгоритма определяется как максимальное количество ячеек памяти, используемых для вычислений над входными данными размером n.

Такой подход к *проектированию* и *анализу* алгоритмов базируется на очень простой вычислительной модели, известной как *модель фон Неймана* (она же машина с произвольным доступом к памяти, или RAM-машина). Она состоит из процессора и памяти бесконечного размера с постоянным временем доступа к каждой из ячеек. Это означает, что каждый шаг занимает фиксированное время, одинаковое для любой операции — арифметической, логической или просто операции доступа к памяти (чтение/запись). Соответственно, для точной оценки времени выполнения алгоритма на ПК достаточно *подсчитать* количество его шагов. После чего сравнивается асимптотическое поведение функций временной сложности разных алгоритмов при $n \to +\infty$: чем быстрее с ростом размера входных данных растет временная сложность, тем хуже алгоритм. Надежность этого подхода долго была предметом дискуссий, тем не менее RAM-машина доминировала на алгоритмической сцене десятилетиями (и до сих пор не утратила своих позиций), ведь она достаточно проста, что влияет на разработку и оценку алгоритмов. Кроме того, на старых ПК и при малых размерах входных данных она дает довольно точную оценку производительности. Поэтому неудивительно, что в большинстве книг, знакомящих с алгоритмами, для оценки их производительности используется именно модель фон Неймана [6].

Но последние десятилетия стали временем существенных перемен, что вызвало необходимость изменений в разработке и анализе алгоритмов. Во-первых, усложнилась архитектура современных ПК (это давно уже не один ЦП с однородной оперативной памятью). Во-вторых, резко возрос размер входных данных и теперь случай $n \to +\infty$ перестал быть только теорией, ведь в наше время данные в изобилии генерируются множеством источников, таких как расшифровка последовательностей ДНК, банковские трансакции, мобильная связь, навигация и поиск в Интернете, аукционы и пр. Для современных ПК RAM-машина стала абстракцией, которую нельзя применить, зато рост количества данных так повлиял на бизнес [2], общество [1] и науку в целом [3], что разработка алгоритмов, подходящих для предельных случаев, стала интересовать не только теоретиков, но и гораздо более широкую аудиторию профессионалов. В результате мы наблюдаем не только заново вспыхнувший интерес к этой теме, но и появление термина «алгоритм» даже в обычной речи.

В новых условиях потребовались новые вычислительные модели, способные лучше абстрагироваться от особенностей современных компьютеров и приложений и точнее оценивать производительность алгоритмов. Современный ПК состоит из одного или нескольких ЦП (многоядерных, графических, тензорных и т. п.) и имеет сложную иерархию уровней памяти. У каждого уровня свои технологические особенности (рис. 1.1). Это кэши L1 и L2, RAM, один или несколько HDD или SSD;

возможно, данные распределены по целому набору вычислительных узлов в сети (хостов) со своей иерархией (которая зависит и от их географического положения) — так называемое облако. Каждый из этих уровней памяти имеет собственные стоимость, емкость, задержку, пропускную способность и метод доступа. Чем ближе к ЦП, тем он меньше, быстрее и дороже. Для доступа к кэшам достаточно наносекунд, тогда как извлечение данных с дисков (ввод-вывод на внешнем устройстве) занимает миллисекунды. Это так называемое *узкое место в подсистеме ввода-вывода* с поразительным коэффициентом замедления 10^5–10^6, который прекрасно иллюстрирует цитата, приписываемая Томасу Кормену: «Разницу в скоростях доступа к оперативной и дисковой памяти можно сравнить со скоростью заточки карандаша в случаях, когда точилка лежит на вашем столе и когда за ней требуется лететь на другой конец света».

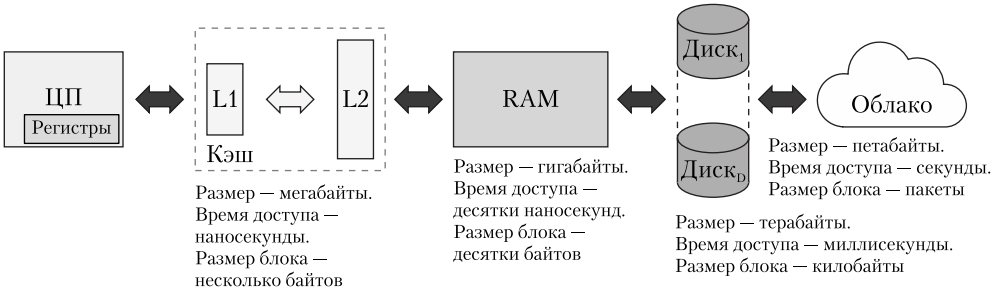

Рис. 1.1. Пример иерархии запоминающих устройств современного ПК

Разумеется, в ходе инженерных исследований ищут способы снизить влияние узкого места в подсистеме ввода-вывода на эффективность приложений, работающих с большими наборами данных. Но хорошие разработка и проектирование алгоритмов дают результаты, превосходящие лучшие технологические достижения. Сейчас на простом примере я покажу, почему так происходит[1].

Рассмотрим три алгоритма оценки времени ввода-вывода с возрастающей сложностью: $C_1(n) = n$, $C_2(n) = n^2$ и $C_3(n) = 2^n$. Здесь $C_i(n)$ —количество обращений к диску, к которому i-й алгоритм прибегает для обработки n входных данных. В первых двух случаях мы видим *полиномиальный* рост количества операций ввода-вывода, в последнем же случае оно растет *экспоненциально*. Для упрощения вычислений взяты максимально простые и поэтому нереалистичные формулы, так как в данном случае моя задача — просто продемонстрировать принцип. Оценим количество данных, которое каждый алгоритм обрабатывает за время t, если каждая операция ввода-вывода занимает время c. Для этого нужно найти n из уравнения $C_i(n)c = t$. Для первого алгоритма этот параметр рассчитывается по формуле t/c, для второго — $\sqrt{t/c}$, а для третьего — $\log_2(t/c)$, что наглядно показывает, почему полиномиальные алгоритмы

[1] Я использовал объяснение Фабрицио Луччо [8], заменив шаги алгоритма операциями ввода-вывода.

считаются *эффективными*, а экспоненциальные — нет, ведь увеличение времени t не сильно меняет объем обрабатываемых экспоненциальным алгоритмом данных.

Разумеется, это утверждение допускает исключения, например, для задач с ограниченным размером входных данных. Бывает и так, что распределение данных благоприятствует эффективному выполнению задачи. Но такие ситуации возникают довольно редко, так что меньшее время выполнения полиномиальных алгоритмов означает, что они считаются *доказуемо* эффективным и предпочтительным способом решения задач. Опыт показывает, что в большинстве случаев экспоненциальное время появляется при решении задачи методом полного перебора, тогда как полиномиального времени можно добиться только благодаря более глубокому пониманию сути задачи. Именно поэтому полиномиальные алгоритмы со многих точек зрения можно считать оптимальным вариантом.

Предположим, что наши алгоритмы удалось запустить на компьютере, подсистема ввода-вывода которого работает быстрее в k раз. С каким объемом данных сможет справиться такой компьютер? Для ответа на этот вопрос нужно поменять временной интервал на $k \times t$, обозначив, что теперь для выполнения алгоритмов доступно в k раз больше времени, чем в предыдущем случае. Мы увидим, что первый алгоритм будет работать в k раз быстрее, скорость второго домножится на $\sqrt{k}$, а к скорости последнего *всего лишь прибавится небольшое число* $\log_2 k$. То есть рост вычислительных мощностей в k раз незначительно сказывается на времени выполнения экспоненциального алгоритма. На алгоритмы со степенной зависимостью от времени, такие как второй из упомянутых ранее алгоритмов, рост мощности влияет позитивно, но чем выше степень, тем меньшее улучшение производительности мы увидим. А именно, для алгоритма $C(n) = n^{\alpha}$ в k раз более мощный компьютер даст прирост скорости в $\sqrt[\alpha]{k}$ раз. Так что можно с достаточной уверенностью утверждать: правильный выбор алгоритма влияет на скорость выполнения намного сильнее, чем любой рост производительности HDD или SSD[1].

Теперь, когда я показал вам важность корректного проектирования алгоритмов, вернемся к анализу их производительности. Для примера рассмотрим простую задачу — нахождение суммы целых чисел, хранящихся в массиве $A[1, n]$. Первое, что приходит в голову, — это поэлементный просмотр и суммирование элементов массива с сохранением промежуточного результата во временной переменной. Этот алгоритм состоит из n шагов, так как операция сложения выполняется после обращения к каждому элементу массива A.

Для перехода к общему случаю рассмотрим семейство алгоритмов $A_{s, b}$, в которых шаблон доступа к элементам задается параметрами s и b. В частности, каждый массив будет логически разделен на блоки из b элементов, например $A_j = A[j \times b + 1, (j + 1) \times b]$ для $j = 0, 1, 2, ..., n/b - 1$[2]. После суммирования элементов в блоке A_j происходит переход к блоку A_{j+s}, находящемуся на s блоков правее. Будем считать массив A циклическим. В этом случае, если переход к следующему блоку приводит

[1] Более подробно эта тема рассматривается в книге Джеффри С. Виттера [11].

[2] Для удобства предположим, что n и b являются степенями двойки, то есть b — делитель n.

к выходу за границы массива, алгоритм начинает просмотр с начала, рассчитывая индекс блока по формуле $(j + s) \bmod (n/b)$[1]. Очевидно, что далеко не при всех s мы сможем учесть все блоки массива A и таким образом суммировать содержащиеся в нем целые числа. Но в случае, когда s и n/b являются взаимно простыми числами, последовательность индексов посещенных блоков, то есть $j = s \times i \bmod (n/b)$ для $i = 0, 1... n/b - 1$, будет перестановкой целых чисел $\{0, 1... n/b - 1\}$. Соответственно, шаблон $A_{s, b}$ затронет все блоки массива A, и мы получим сумму всех содержащихся в нем целых чисел. Зачем нужна такая параметризация? Дело в том, что, меняя s и b, мы сможем суммировать числа в массиве A в соответствии с различными шаблонами доступа к памяти: от упомянутого ранее последовательного перебора (случай $s = b = 1$) до последовательного доступа к блокам (при большем b) и случайного доступа к блокам (при большем s). С точки зрения вычислений все алгоритмы $A_{s, b}$ эквивалентны, так как каждый из них читает и суммирует ровно n элементов. Но на практике они имеют разную производительность, причем эта разница увеличивается с ростом n, ведь чем больше размер массива, тем по большему количеству уровней памяти будут распределяться данные. При этом каждый из уровней имеет собственные емкость, задержку, пропускную способность и метод доступа. Соответственно, эквивалентная эффективность, полученная ранее на модели фон Неймана, не дает представления о том, сколько же времени на самом деле займет суммирование элементов массива A.

Нужна другая модель, лучше описывающая работу реальных компьютеров и при этом достаточно простая для того, чтобы мы по-прежнему могли разрабатывать и анализировать алгоритмы. Ранее я утверждал, что с учетом большого разрыва в производительности дисковой и оперативной памяти хорошей оценкой временной сложности алгоритма может служить количество операций ввода-вывода. Это отражено в *двухуровневой модели памяти*, которая абстрагирует компьютер до внутренней памяти ограниченного размера M и неограниченной дисковой памяти, которая работает путем записи данных в блоки размером B, называемые *страницами диска*, и чтения с них. Модель может включать в себя D дисков неограниченного размера. В этом случае при каждом вводе-выводе на D страниц, расположенных на разных дисках, записывается в общей сложности $D \times B$ элементов и столько же читается с них. Следует отметить, что двухуровневое представление не ограничивает модель абстрактными вычислениями на основе дисковой памяти. Мы можем выбрать любые два уровня иерархии памяти с корректно заданными параметрами M и B. Производительность алгоритма в этой модели оценивается путем подсчета:

- обращений к страницам диска (далее будем называть их вводом-выводом);
- времени выполнения (процессорного времени);
- страниц диска, используемых алгоритмом в качестве рабочего пространства.

Кроме того, для проектирования хороших алгоритмов, работающих с большими наборами данных, предлагается учитывать принципы *пространственной* и *временной*

[1] Функция деления по модулю для двух положительных целых чисел x и $m > 1$ определяется как остаток от деления x на m.

локальности. Первый предписывает структурировать данные на диске (-ах) таким образом, чтобы каждая считанная оттуда страница содержала как можно больше полезных данных. Второй требует, чтобы перед записью на диск с данными во внутренней памяти было проделано как можно больше полезной работы.

Проанализируем временную сложность алгоритмов $A_{s,b}$ для новой модели. Будем считать, что процессорное время равно n, а занятое пространство составляет n/B страниц диска при любых значениях s и b. Начнем с простейшего случая $s = 1$. Здесь все очевидно. Алгоритмы $A_{1,b}$ перемещаются по элементам массива A вправо, суммируя элементы по одному блоку за раз, выполняя n/B операций ввода-вывода при любых b. При изменении s и b ситуация немного усложняется. Пусть $s = 2$. Выберем $b < B$, которое для простоты будет делителем B. То есть каждый блок размером B состоит из B/b (логических) блоков размером b и алгоритмы $A_{2,b}$ проверяют только половину из них, ведь мы установили параметр $s = 2$. Фактически это означает, что каждая страница размером B участвует в процессе суммирования наполовину, индуцируя в общей сложности $2n/B$ операций ввода-вывода. Эту формулу несложно обобщить, записав затраты на операции ввода-вывода как $\min\{s, B/b\} \times (n/B)$, что для больших скачков по массиву A дает n/b. Она намного лучше оценивает сложность алгоритмов $A_{s,b}$ в реальном времени, хотя и не учитывает все особенности диска. Все операции ввода-вывода считаются равноценными независимо от их распределения, что не соответствует действительности, потому что на реальных дисках последовательный ввод-вывод выполняется быстрее случайного[1].

Таким образом, мы выяснили, что все алгоритмы $A_{s,b}$ имеют одинаковую сложность ввода-вывода n/B независимо от s, хотя при использовании механических дисков их поведение сильно различается, ведь с ростом s растет и время поиска на диске. Можно сделать вывод, что даже двухуровневая модель памяти достаточно хорошо описывает поведение алгоритмов на реальных компьютерах. Именно поэтому она широко применяется для оценки производительности алгоритмов на больших наборах данных. Для достижения максимальной точности мы будем учитывать не только количество операций ввода-вывода, но и характеристику их *распределения* (случайного или последовательного) по диску.

Здесь можно вспомнить, что за последние годы размер внутренней памяти M увеличился настолько, что теоретически в нее может поместиться большая часть рабочего набора алгоритма, то есть набора страниц, к которым он будет обращаться в ближайшем будущем. Это позволяет значительно сократить количество ошибок ввода-вывода. Но сейчас я вам покажу, как замедляет работу алгоритма даже небольшая часть данных, находящихся на диске, и вы поймете, почему не стоит пренебрегать организацией данных и в чрезвычайно благоприятных ситуациях.

Предположим, размер входных данных $n = (1 + \varepsilon)M$ больше, чем размер внутренней памяти, умноженный на $\varepsilon > 0$. Насколько это повлияет на среднюю стоимость

[1] Эта разница незначительна в случае DRAM и твердотельных накопителей, где распределение обращений к памяти не оказывает существенного влияния на пропускную способность.

шага алгоритма, который обращается к данным, находящимся как во внутренней памяти, так и на диске? Чтобы упростить анализ, введем в рассмотрение такой параметр, как вероятность *промаха* $p(\varepsilon)$ (непопадания в дисковый кэш в оперативной памяти). Когда алгоритм работает только с дисковыми данными, $p(\varepsilon) = 1$, если же набор данных целиком помещается в оперативной памяти, то $p(\varepsilon) = 0$. Для случаев, когда алгоритм работает с данными, хранящимися как в памяти, так и на диске, $p(\varepsilon) = \frac{\varepsilon M}{(1+\varepsilon)M} = \frac{\varepsilon}{1+\varepsilon}$. Другими словами, этот параметр можно считать мерой нелокальности запросов алгоритма к памяти.

Перечислю остальные нужные нам параметры. Пусть c — это соотношение скоростей одной операции ввода-вывода и одного доступа к внутренней памяти. На практике $c \approx 10^5 ... 10^6$, выше я это уже упоминал. Долю шагов, для которых алгоритму требуется доступ к памяти, обозначим f. Согласно [5], этот параметр, как правило, составляет 30–40 %. Средние временные затраты на такой доступ к памяти обозначим t_m, затраты же на каждый шаг вычисления или на доступ к внутренней памяти пусть будут равны 1. Чтобы получить формулу для расчета t_m, следует выделить два случая: доступ к памяти, происходящий с вероятностью $1 - p(\varepsilon)$, и доступ к диску, происходящий с вероятностью $p(\varepsilon)$. В результате мы получим $t_m = 1 \times (1 - p(\varepsilon)) + c \times p(\varepsilon)$.

Теперь для нашего алгоритма можно оценить *средние временные затраты на один шаг*. Они рассчитываются по формуле $1 \times (1 - f) + t_m \times f$, где $1 - f$ — это доля вычислительных шагов, а f — доля обращений к памяти, как внутренней, так и дисковой. Подставив вычисленное значение t_m, мы можем понизить эту оценку до $3 \times 10^4 \times p(\varepsilon)$. Эта формула ясно показывает, что даже для алгоритмов, использующих локальность обращений, то есть небольшую $p(\varepsilon)$, замедление может оказаться значительным — где-то на четыре порядка выше ожидаемой величины, то есть $p(\varepsilon)$. Для примера возьмем алгоритм, в котором локальность обращений к памяти строго ограниченна: скажем, всего 1 из 1000 обращений идет к данным, хранящимся на диске, то есть $p(\varepsilon) = 0{,}001$. В результате его производительность будет более чем в 30 раз ниже, чем производительность алгоритма, ведущего вычисления полностью во внутренней памяти.

Разумеется, это лишь вершина айсберга. Чем больше объем данных, которые должен обработать алгоритм, тем больше уровней памяти будет задействовано в их хранении и тем разнообразнее будут типы промахов, с которыми придется бороться для повышения эффективности. Надеюсь, я смог продемонстрировать вам, что в системах с иерархической памятью игнорирование затрат на обращения к памяти может помешать использовать алгоритм для больших входных данных.

В этой книге я разобрал несколько сложных задач, имеющих элегантные алгоритмические решения, ведь именно эффективность алгоритмов обеспечивает возможность работы с большими наборами данных, которые встречаются во многих реальных приложениях. Я продемонстрирую подробности разработки каждого алгоритма, отдельно прокомментировав трудности, с которыми пришлось столкнуться. Мы поговорим о способах превращения теоретически эффективных алгоритмов

в практически эффективный код. Как теоретик, я слишком часто слышал фразу: «Ваш алгоритм вряд ли допускает эффективную реализацию!» Кроме того, внося свой вклад в недавний всплеск исследований в области *проектирования алгоритмов* [10], мы углубленно рассмотрим вычислительные особенности некоторых из них, прибегнув к другим успешным моделям вычислений — в основном к потоковой [9] и кэш-независимой модели [4]. Эти модели позволят зафиксировать и выделить некоторые интересные проблемы базовых вычислений, такие как проходы по диску (потоковая модель) и универсальная масштабируемость (модель с независимостью от кэша). Я постараюсь максимально просто описать все эти проблемы, однако вряд ли смогу превратить эту высшую математику для неспециалистов в науку для «чайников» [2], ведь для создания рабочего алгоритма необходимо учитывать гораздо больше вещей. В ведущих ИТ-компаниях, таких как Amazon, Facebook, Google, IBM, Microsoft, Oracle, Spotify и т. д., прекрасно понимают, как сложно найти людей с навыками, подходящими для проектирования и разработки хороших алгоритмов. Моя книга лишь поверхностно коснется проектирования и разработки алгоритмов, ее главная цель — вдохновить вас на повседневную работу в качестве разработчика программного обеспечения.

Список литературы

1. Person of the year // Time Magazine, 168: 27, December 2006.
2. Business by numbers // The Economist, September 2007.
3. *Butler D.* 2020 computing: Everything, everywhere // Nature, 440 (7083): 402–405, 2006.
4. *Fagerberg R.* Cache-oblivious model // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 264–269, 2016.
5. *Hennessy J. L., Patterson D. A.* Computer architecture: A quantitative approach. Morgan Kaufmann, fourth edition, 2006.
6. *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 2016.
7. *Knuth D.* The Art of computer programming: Fundamental algorithms, Vol. 1. Addison-Wesley, 1973.
8. *Luccio F.* La struttura degli algoritmi. Boringhieri, 1982.
9. *Muthukrishnan S.* Data streams: Algorithms and applications // Foundations and Trends in Theoretical Computer Science, 1 (2): 117–236, 2005.
10. *Sanders P.* Algorithm engineering — an attempt at a definition // *Albers S., Alt H., Näher S.* Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, 5760, Springer, 321–323, 2009.
11. *Vitter J. S.* External memory algorithms and data structures // ACM Computing Surveys, 33 (2): 209–271, 2001.