МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

В.В. ЛАНДОВСКИЙ

АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2018

Рецензенты: канд. техн. наук, доц. *В.А. Астапчук*,
старший преподаватель. *С.А. Менжулин*.

Работа подготовлена на кафедре автоматизированных систем управления для студентов III курса АВТФ направления 09.03.01 - «Информатика и вычислительная техника»

Ландовский В.В. Алгоритмы обработки данных: учеб. пособие. / В.В. Ландовский – Новосибирск: Изд-во НГТУ, 2018. – 64 с.

В настоящем пособии рассмотрены две группы алгоритмов: алгоритмы сортировки и алгоритмы на строках. Среди алгоритмов сортировки выделены простые обменные методы, имеющие полиномиальную временную сложность, методы с линейно-логарифмической и линейной оценками времени. Представлено описание классических алгоритмов быстрого поиска образца в тексте с использованием вспомогательных структур, приведены алгоритмы их построения. Рассмотрены алгоритмы вычисления редакционного расстояния между строками.

ОГЛАВЛЕНИЕ

ВМЕСТО ПРЕДИСЛОВИЯ

В настоящее время незнание фундаментальных алгоритмов становится нормой среди разработчиков программ. Это неудивительно, так как пробелы в базовом образовании компенсируются с одной стороны обилием легкодоступных учебных материалов, а с другой – возможностями применения готовых библиотечных рецептов. Нет ничего дурного, в том, что специалист (в особенности начинающий) заглядывает в поисках информации в первый попавшийся интернет-источник. Использование макросредств без их глубокого изучения тоже может быть оправдано, если единственными критериями являются скорость написания и корректность работы программы. Но если наоборот – потребуется обеспечить достаточную скорость обработки данных, то возможны проблемы. В зависимости от таланта и временных рамок один с нуля изобретет велосипед, другой наконец-то изучит то, что пропустил в школе. И только тот, кто заранее уделил достаточно времени изучению основополагающих концепций и приобрел достаточную широту знаний, способен сразу выбрать правильный путь решения задачи.

Темпы развития компьютерной техники и программного обеспечения в последнее время требуют от специалистов постоянной, почти непрерывной актуализации своих знаний. Может показаться, что обучение не связанное с решением реальных задач – непозволительная роскошь. Однако среди многообразия современных материалов посвященных отдельным языкам, средствам разработки и конкретным прикладным решениям теряются общие, нестареющие подходы к решению большого класса задач. Меняется главным образом форма: всё более дружественные интерфейсы, всё более высокоуровневые средства, высокие скорости обработки и наглядные способы отображения информации. Содержание же остается прежним, идеи, которые вот-вот отпразднуют вековой юбилей, не теряю своей актуальности. Это позволяет сделать вывод о том, что незнание фундамента лишает возможности занимать твердую позицию и делает человека заложником чужих коммерческих интересов,

связанных с развитием тех или иных программных средств. И напротив – наличие базовых знаний позволяет уверенно адаптироваться к изменениям, быстро осваивать новые технологии.

В дополнение к сказанному уместно будет процитировать фрагмент из книги одного из ведущих исследователей и преподавателей в области информатики Н. Вирта: «Программирование – это искусство конструирования. Как можно научить конструкторской, изобретательской деятельности? Есть такой метод: выделить простейшие строительные блоки из многих уже существующих программ и дать их систематическое описание. Но программирование представляет собой обширную и разнообразную деятельность, часто требующую сложной умственной работы. Ошибочно считать, что ее можно свести к использованию готовых рецептов. В качестве метода обучения нам остается тщательный выбор и рассмотрение характерных примеров. Конечно, не следует считать, что изучение примеров всем одинаково полезно. При этом подходе многое зависит от сообразительности и интуиции обучающегося» [1].

1. АЛГОРИТМЫ СОРТИРОВКИ

Сортировка – широкое понятие, в общем случае под сортировкой понимают упорядочивание множества объектов по одному или нескольким признакам. Во всяком случае, отнесение изделий к той или иной категории качества согласно некоторому набору критериев называют именно сортировкой.

Если говорить о сортировке в контексте программирования, то первое что приходит на ум – это упорядочивание элементов массива по их значениям. Однако следует помнить, что в общем случае сортируемые объекты могут иметь сложную внутреннюю структуру, и признак, по которому производится упорядочивание, может быть составным. Этот признак (совокупность признаков), как правило, называют ключевым полем или ключом. На множестве значений ключа должно существовать отношение линейного порядка. В большинстве случаев количество операций реализующих данное отношение не зависит от числа сортируемых объектов, но можно привести примеры такой зависимости (п. 2.4.1). Структура данных, использующаяся для доступа к объектам, может существенно влиять на асимптотику алгоритмов сортировки. При выборе алгоритма следует обращать внимание на время доступа к объекту, при обращении по номеру.

Важной характеристикой алгоритма сортировки является **устойчивость**, она заключается в сохранении относительного порядка объектов с одинаковыми значениями ключа.

1.1. Простые методы

В этом разделе рассмотрены простые алгоритмы, в которых процесс сортировки представляет собой последовательность обменов. К числу преимуществ этих методов следует отнести минимальный объем дополнительной памяти, небольшие константы в оценках времени и, разумеется, простоту реализации. Основной недостаток – асимптотика времени выполнения $O(n^2)$, где n – количество сортируемых элементов.

1.1.1. Сортировка пузырьком

Для понимания этого алгоритма полезно изобразить последовательность сортируемых элементов сверху вниз. Разделим последовательность на две части упорядоченную и неупорядоченную, первая располагается сверху и вначале отсутствует. Элементы перемещаются в неупорядоченной части путем последовательных обменов с соседними. На каждой итерации элемент с наиболее «легким» ключом поднимаются на верхнюю границу неупорядоченной части, а размер (глубина) упорядоченной части увеличивается на единицу. Очевидно, что количество таких итераций должно быть на единицу меньше количества элементов, так как неупорядоченная часть из одного элемента упорядочена. Подробный алгоритм представлен на рисунке 1.2. На рисунке 1.1. показан пример работы алгоритма. Столбцы соответствуют состояниям массива до сортировки, на каждой итерации и после сортировки, граница упорядоченной части обозначена двойной чертой. Стрелками показано перемещение элементов.

Очевидным усовершенствованием будет добавление выхода из внешнего цикла, при условии, что в процессе выполнения внутреннего цикла не было ни одного обмена.

Если просматривать неупорядоченную часть, как в обратном, перемещая локальный минимум, так и в прямом направлении, перемещая локальный максимум, то это будет соответствовать алгоритму сортировки «перемешиванием» (шейкерной сортировке).

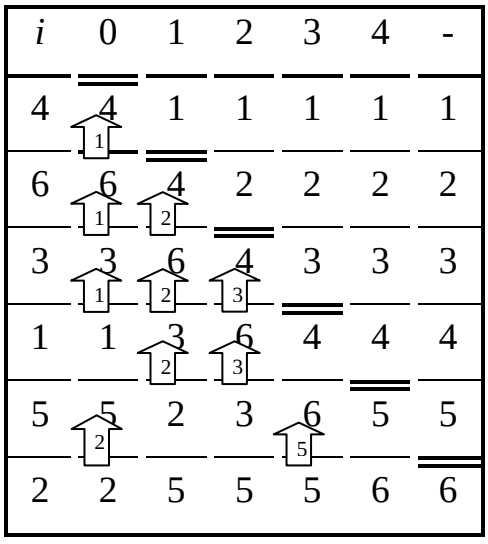

Рисунок 1.1. Пример сортировки пузырьком.

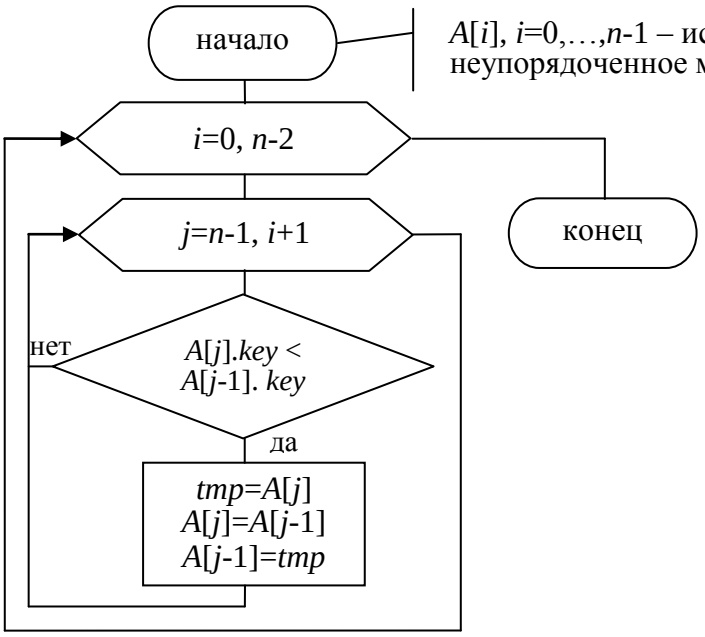

Рисунок 1.2. Алгоритм сортировки пузырьком.

1.1.2. Сортировка вставками

Отличие этого метода от предыдущего заключается в том, что элементы перемещаются в упорядоченной части. На каждой итерации очередной элемент, находящийся на верхней границе неупорядоченной части, помещается на нужную позицию в отсортированной части (вставляется). Очевидно, что уже на первой итерации необходимо наличие неупорядоченной части, для этого первым добавляют фиктивный элемент с минимально возможным значением ключа. Пример сортировки вставками показан на рисунке 1.3, на рисунке 1.4 представлена блок-схема алгоритма.

i	1	2	3	4	5	-
-∞	-∞	-∞	-∞	-∞	-∞	-∞
4	4	4	3	1	1	1
6	6	6	4	3	3	2
3	3	3	6	4	4	3
1	1	1	1	6	5	4
5	5	5	5	5	6	5
2	2	2	2	2	2	6

Рисунок 1.3. Пример сортировки вставками.

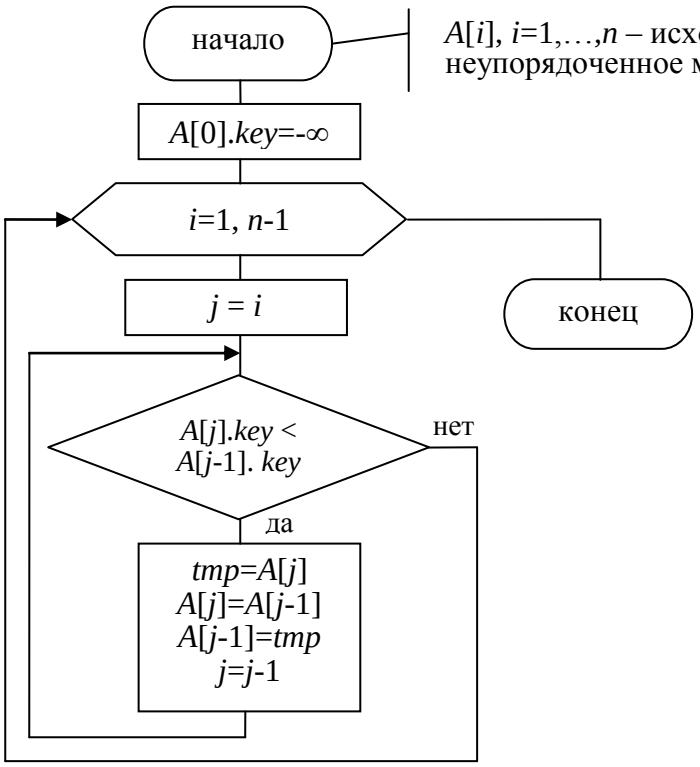

Рисунок 1.4. Алгоритм сортировки вставками.

Внешний цикл сортировки вставками обязан дойти до конца, но внутренний может заканчиваться в произвольный момент.

1.1.3. Сортировка выбором

Два предыдущих метода меняли местами только соседние элементы. Эта особенность обеспечивает одинаковую асимптотику как для структур данных с последовательным доступом (списков), так и для структур с произвольным доступом (массивов). При этом количество обменов выглядит необоснованно высоким. Сортировка выбором на каждой итерации помещает очередной элемент на подходящее место, меняя его местами с тем, который прежде занимал это место. Иначе говоря, на i-ой итерации из $A[i]\ldots A[n\text{-}1]$ выбирается элемент с наименьшим ключом и меняется местами с $A[i]$.

На рисунке 1.5 показан пример работы сортировки выбором, позиция, на которую выбирается элемент, обведена двойной чертой. Блок-схема алгоритма приведена на рисунке 1.6. Данный алгоритм, также как и предыдущие два

нетрудно адаптировать для сортировки, множества реализованного на базе связного списка. Это объясняется тем, что обращение к элементам осуществляется последовательно.

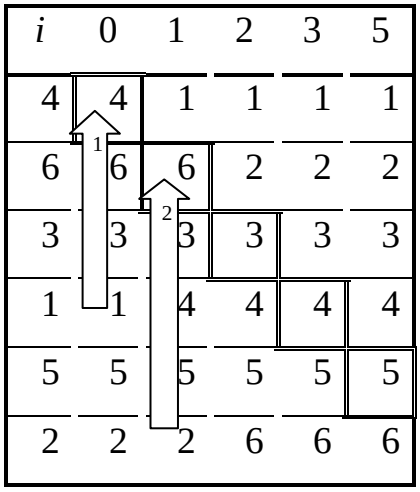

Рисунок 1.5. Пример сортировки выбором.

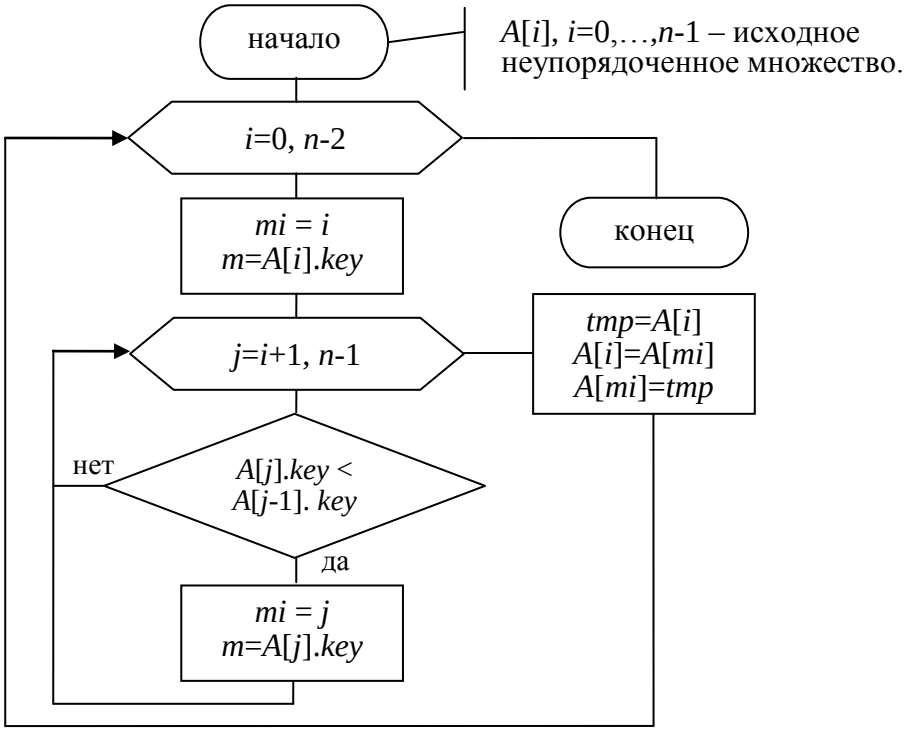

Рисунок 1.6. Алгоритм сортировки выбором.

Следует отметить, что время выполнения такого алгоритма есть $\Theta(n^2)$, так как на каждой итерации обязательно просматривается вся неупорядоченная часть.

1.2. Сортировка Шелла

Метод Шелла [2], предложенный еще в 1959 г. считается усовершенствованием сортировки вставками. Ключевая идея заключается в сравнении и перемещении элементов находящихся на значительном удалении друг от друга. Похожий подход применительно к пузырьковой сортировке используется в методе сортировки «расческой», который был опубликован значительно позже.

В алгоритме, представленном на рисунке 1.7, переменная *m* определяет расстояние между сравниваемыми элементами. На первом шаге внешнего цикла сортируются элементы, образующие пары, количество таких пар *n*/2. На последнем шаге сортируются соседние элементы, можно считать это обычной сортировкой вставками, т.е. с каждым новым значением *j* очередной элемент занимает свое место в отсортированной части. Однако, с учетом работы проделанной при больших значениях *m*, высока вероятность того, что элементы уже находятся на своих местах, или рядом с ними.

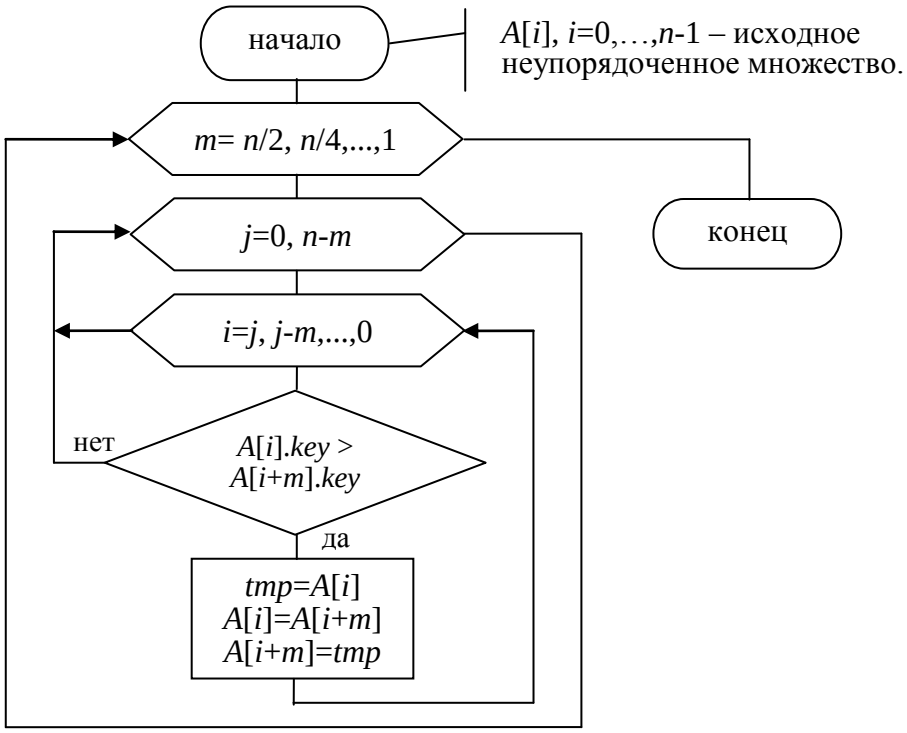

Рисунок 1.7. Алгоритм сортировки Шелла.

Очевидно, что внешний цикл всегда имеет $\log_2(n)$ итераций. Так как m уменьшается, число итераций цикла, в котором изменяется j, линейно по n. Максимальное количество итераций на третьем уровне (в цикле по переменной i) с уменьшением m растет экспоненциально и компенсирует логарифм внешнего цикла. Таким образом, в худшем случае время выполнения сортировки растет как квадрат количества элементов. Порядок сравнения и возможного перемещения элементов показан на рисунке 1.8, где указаны все возможные значения i на каждом шаге внешних циклов. Если i-й элемент уже находится на своем месте, то цикл прекращается и дальнейшего изменения i не происходит.

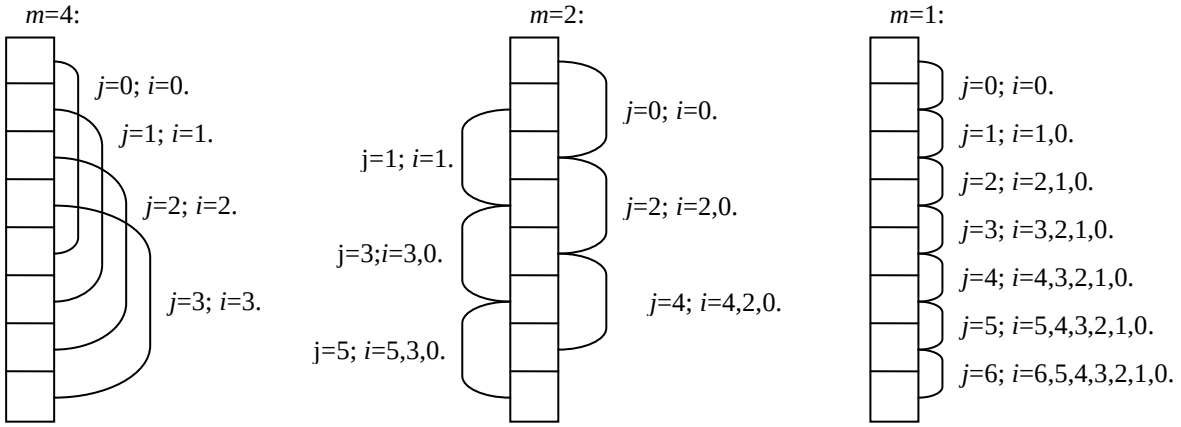

Рисунок 1.8. Порядок сравнения элементов в сортировке Шелла, n=8.

Усовершенствованные версии алгоритма отличаются от оригинальной последовательностью расстояний между сравниваемыми элементами. Роберт Седжвик предложил следующую последовательность: $d_i=9\cdot2^i-9\cdot2^{i/2}+1$ если i четное, $d_i=8\cdot2^i-6\cdot2^{(i+1)/2}+1$, если нечетное, i=0,1,.... Последнее значение i определяется соотношением $3\cdot d_{i+1}>n$. Последовательность используется в порядке обратном вычислению. При использовании таких расстояний временная сложность алгоритма в худшем случае $O(n^{4/3})$ [3].

1.3. «Быстрая» сортировка

Алгоритм известный под названием «quicksort» [4] в худшем случае имеет временную сложность $O(n^2)$, но в среднем работает за $O(n\log(n))$. На каждом шаге алгоритма исходное множество разделяется на две части: с ключами меньше

ключа «опорного элемента» и большими или равными ему. Для полученных таким образом подмножеств процедура повторяется. Разделение продолжается до тех пор, пока в текущем подмножестве присутствуют хотя бы два различных значения ключа. В качестве опорного элемента выбирается элемент, имеющий наибольшее из двух первых различных значений ключа. Время сортировки в каждом конкретном случае зависит от того насколько удачным будет этот выбор. На рисунке 1.9. показан пример работы «быстрой» сортировки, для каждого подмножества указано опорное значение, обозначенное как *v*.

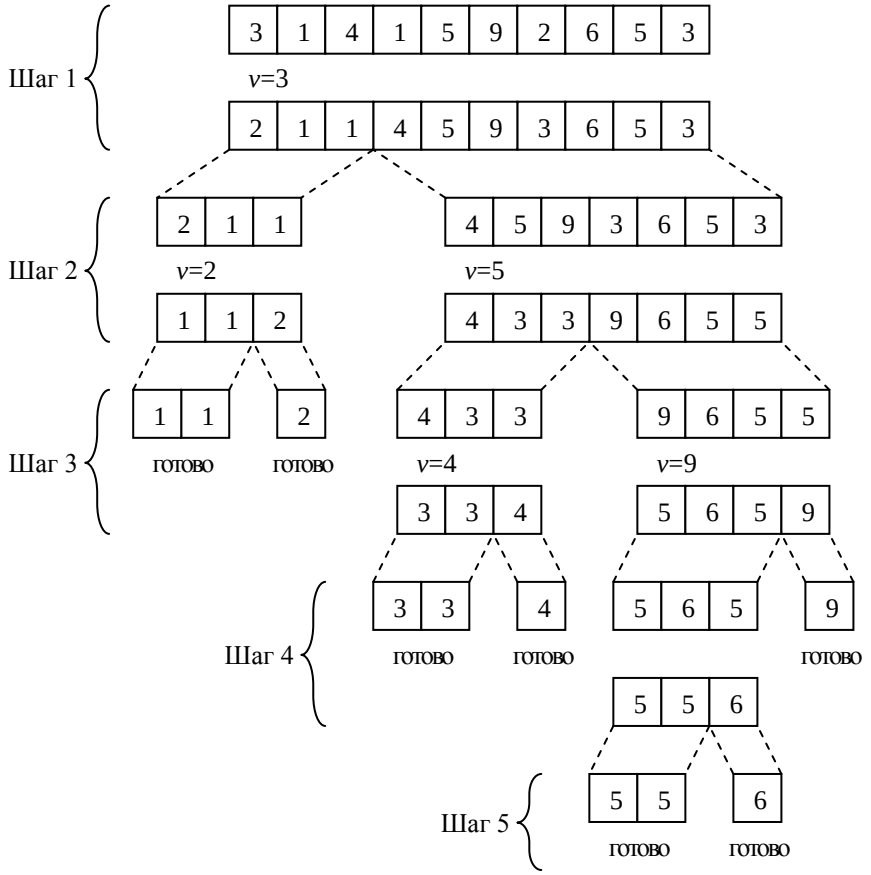

Рисунок 1.9. Пример «быстрой» сортировки.

Для реализации описанного подхода с одной стороны можно пойти по пути создания новых коллекций, копируя элементы исходного множества непосредственно как на рисунке, но это потребует $O(n^2)$ дополнительной памяти. С другой стороны для представления подмножеств достаточно использовать индексы начала и конца, как в алгоритме, представленном на рисунке 1.10. Подробный алгоритм можно найти в [5].

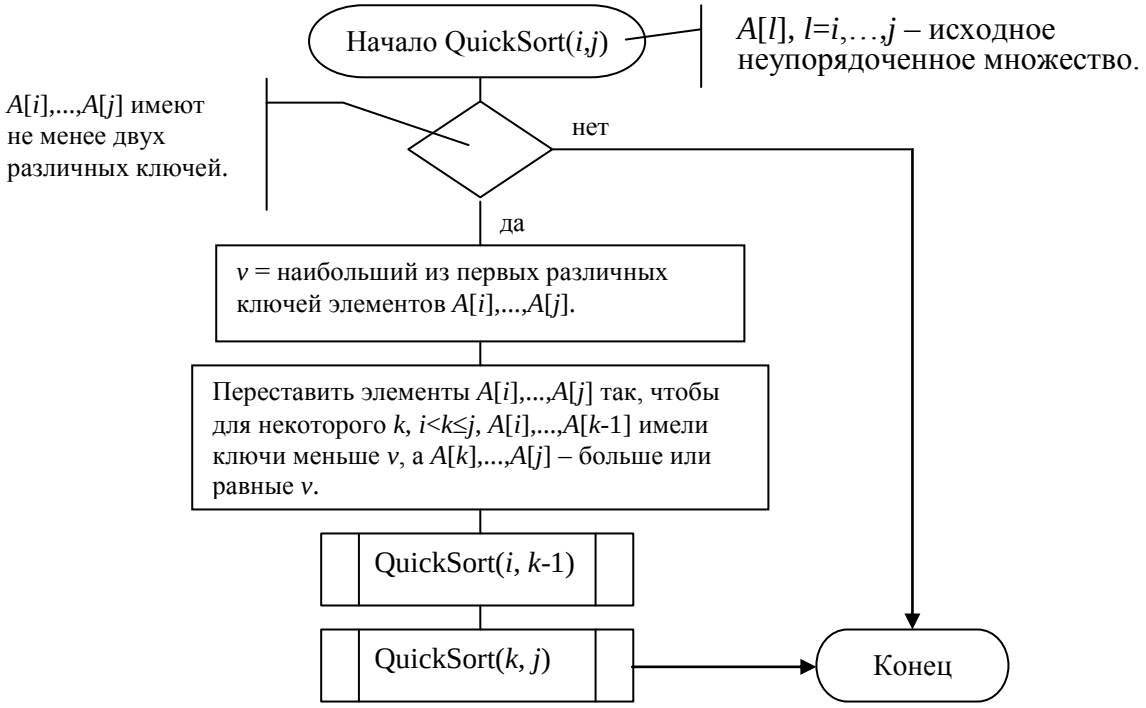

Рисунок 1.10. Алгоритм «быстрой» сортировки.

1.4. Сортировка слиянием

В отличие от «быстрой» сортировки, сортировка слиянием (Merge sort) позволяет гарантированно разделять исходное множество на две равные части с точностью до единицы. При условии, что процедура объединения отсортированных подмножеств выполняется за время линейное относительно количества элементов, время выполнения такой сортировки $\Theta(n \log(n))$. На рисунке 1.11 представлен псевдокод абстрактного алгоритма объединения двух упорядоченных множеств *A* и *B*. Выполнение первого цикла прекращается, когда одно из множеств полностью обработано. В зависимости от того, какое множество закончилось раньше, один из следующих циклов добавит к результату остаток второго множества.

Примеры реализаций сортировки массива методом слияния показаны на рисунке 1.12, итерационный вариант (mergeSortIterative) требует $O(n)$ дополнительной памяти, рекурсивный (mergeSortRecursive) с учетом глубины рекурсии – $O(n\log(n))$. Выделение дополнительной памяти может производиться как при каждом вызове процедуры слияния, так и заранее, передавая в функцию

адрес одного и того же буфера *r*. Остальные параметры функций имеют следующее значение: *a* – исходный массив, *n* – количество элементов массива, *left* – начальный индекс, *right* – увеличенный на единицу индекс последнего элемента.

Текущие элементы *A* и *B* установить на первые элементы соответственно.
Пока не достигнут конец *A* **И** не достигнут конец *B* **Цикл**
    Если текущий элемент *A* меньше текущего элемента *B* **Тогда**
        Добавить текущий элемент *A* к результату.
        Перейти к следующему элементу *A*.
    Иначе
        Добавить текущий элемент *B* к результату.
        Перейти к следующему элементу *B*.
    Конец если
Конец цикла
Пока не достигнут конец *A* **Цикл**
    Добавить текущий элемент *A* к результату. Перейти к следующему элементу *A*.
Конец цикла
Пока не достигнут конец *B* **Цикл**
    Добавить текущий элемент *B* к результату. Перейти к следующему элементу *B*.
Конец цикла

Рисунок 1.11. Алгоритм объединения упорядоченных множеств *A* и *B*.

```
void merge(int a[], int r[],int left, int  mid, int  right) {
	//int *r = new int[right - left];
	int i1 = left, i2 = mid, i3 = 0;
	while ((i1 < mid) && (i2 < right))	{
		if (a[i1] < a[i2])
			r[i3++] = a[i1++];
		else
			r[i3++] = a[i2++];
	}
	while (i1 < mid)
		r[i3++] = a[i1++];
	while (i2 < right)
		r[i3++] = a[i2++];
	for (int i = 0; i < (right - left); i++)
		a[i + left] = r[i];
	//delete[] r;
}
void mergeSortRecursive(int a[],int r[], int left, int right) {
	if (left + 1 >= right)
		return;
	int mid = (left + right) / 2;
	mergeSortRecursive(a, r, left, mid);
	mergeSortRecursive(a, r, mid, right);
	merge(a, r,left, mid, right);
}
void mergeSortIterative(int a[],int r[],int n) {
	for (int i = 1; i < n; i *= 2)
		for (int j = 0; j < n - i; j += 2 * i)
			merge(a,r, j, j + i, min(j + 2 * i, n));
}
```

Рисунок 1.12. Пример реализации сортировки слиянием.

На рисунке 1.13 показан пример работы сортировки слиянием применительно к массиву содержащему десять элементов: 3,1,4,1,5,9,2,6,5,3. Сверху показана работа итерационного алгоритма, в нижней части – рекурсивного.

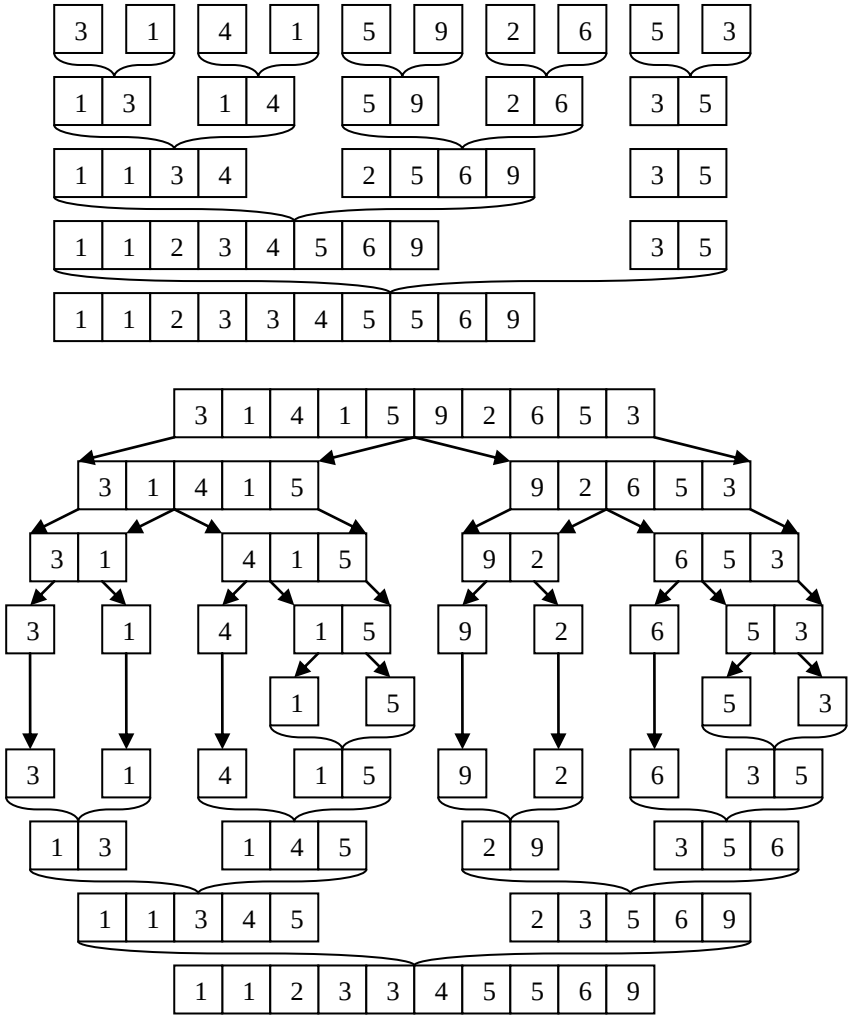

Рисунок 1.13. Схема работы сортировки слиянием.

Объединение упорядоченных последовательностей может использоваться для сортировки больших множеств, размеры которых превышают объемы оперативной памяти.

1.5. Пирамидальная сортировка

Сортировка пирамидой [6] – чрезвычайно простой метод: достаточно вставить все элементы исходной последовательности в пустое частично упорядоченное дерево (пирамиду), а затем извлечь все элементы из дерева,

каждый раз забирая элемент, находящийся в вершине. В результате получим упорядоченную последовательность.

Сложности могут возникнуть на уровне работы с пирамидой, для которой, прежде всего, необходимо выбрать структуру данных – пусть это будет обычный массив. На рисунке 1.14 показан пример пирамиды и соответствующей реализации, в которой индекс родителя для i-го элемента равен $i/2$, левый дочерний элемент i-го имеет индекс $2i$, правый дочерний – индекс $(2i + 1)$.

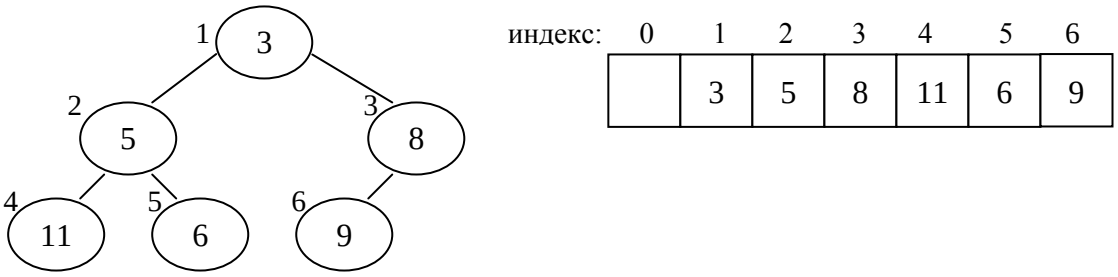

Рисунок 1.14. Реализация пирамиды с помощью массива.

Для работы с таким представлением пирамиды потребуется хранить текущее количество элементов, если тип элементов – целые числа, то для этого можно задействовать нулевую позицию массива. При добавлении нового элемента, он помещается в ячейку, следующую за последним элементом. С учетом иерархии новый элемент будет находиться на листе, который добавится на самый нижний уровень, если нижний уровень полностью заполнен, то появится новый уровень. Остается только проверить и при необходимости восстановить корректный порядок элементов. Для этого новый элемент сравнивается с родителем, если порядок нарушен, они меняются местами и процесс проверки повторяется, такая последовательность обменов может достигнуть корня.

При извлечении корневого элемента на его место перемещается последний элемент. И выполняется проверка, которая в этом случае чуть сложнее. Новый корневой элемент сравнивается с потомками, если порядок верный процедура завершается. В противном случае необходим обмен: при сортировке по возрастанию (используя min-heap) для замены выбирается потомок с меньшим значением ключа, при сортировке по убыванию (используя max-heap) – с

большим. После обмена проверка повторяется, последовательность проверок и обменов может доходить до уровня листьев.

Пример реализации пирамидальной сортировки массива целых чисел показан на рисунке 1.15.

```
void insert(int *h, int v) {
	h[++h[0]] = v;
	for (int i = h[0]; i > 1; i /= 2)
		if (h[i] < h[i / 2])
			std::swap(h[i], h[i / 2]);
		else
			break;
}
int removeMin(int h[]) {
	int r = h[1], m;
	h[1] = h[h[0]--];
	for (int i = 1; i <= h[0] / 2;) {
		if (((i * 2 + 1)<=h[0]) && h[i * 2] > h[i * 2 + 1])
			m = i * 2 + 1; //если правый потомок существует и он меньше левого
		else
			m = i * 2;
		if (h[i] > h[m]) {
			std::swap(h[i], h[m]);
			i = m;
		}
		else
			break;
	}
	return r;
}
void heapSort(int a[], int n) {
	int *h = new int[n + 1], i;
	h[0] = 0;
	for (i = 0; i < n; i++)
		insert(h, a[i]);
	i = 0;
	while (h[0]>0)
		a[i++] = removeMin(h);
	delete[] h;
}
```

Рисунок 1.15. Пример реализации сортировки пирамидой.

1.6. Сортировка подсчетом

Сортировка подсчетом (counting sort) может применяться не во всех случаях, ее эффективность зависит от того, насколько мал диапазон изменения значений ключа. Устойчивый вариант алгоритма, для сортировки массива A содержащего n цифр, представлен на рисунке 1.16. Количество возможных значений ключа обозначено k, оно равно десяти, C – вспомогательный массив для подсчета, B – массив для записи результатов.

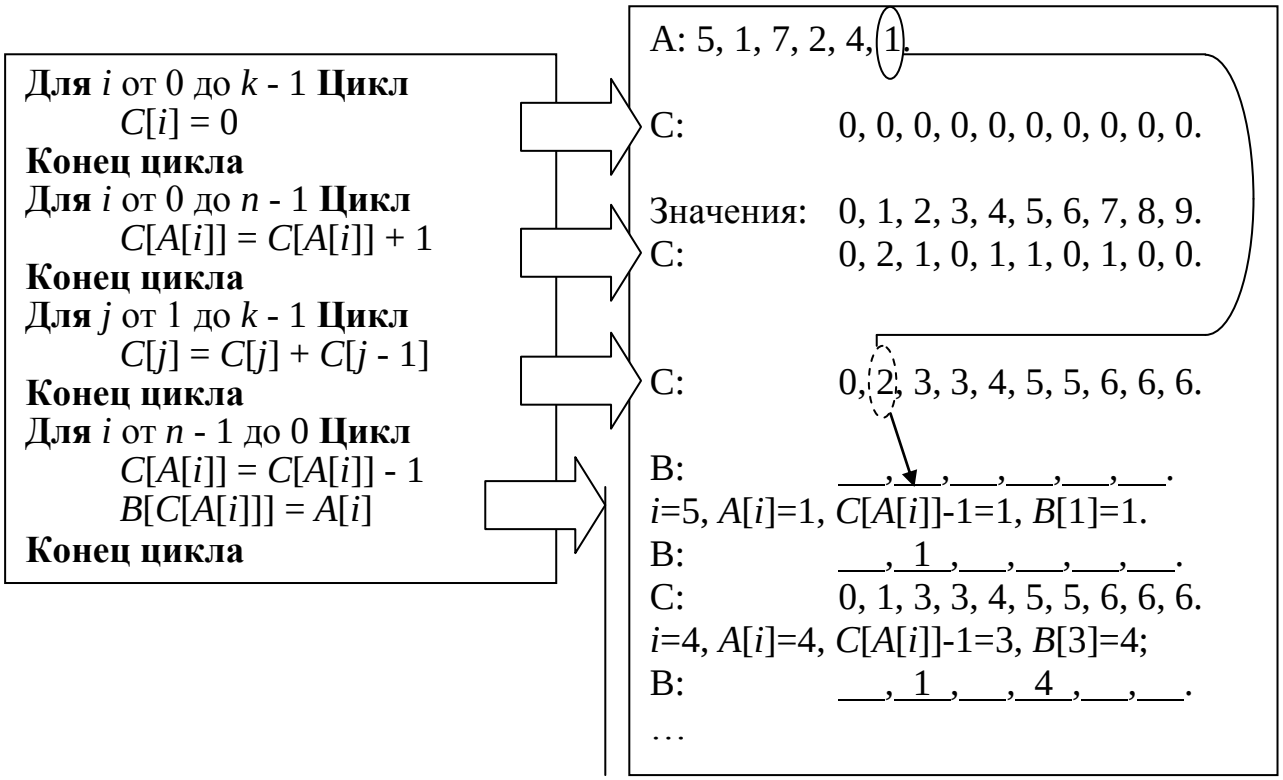

Рисунок 1.16. Алгоритм (слева) и пример (справа) сортировки подсчетом.

Этот алгоритм прекрасно подходит для сортировки отдельных символов. Очевидно, что при постоянной мощности алфавита временная сложность и требования к памяти линейны по n. В общем случае оценку следует уточнить: $O(n+k)$.

1.7. Карманная сортировка

Идея обобщенной карманной сортировки (bucket sort) заключается в следующем. Элементы исходного множества распределяются по подгруппам, которые принято называть карманами или корзинами. Каждому карману может соответствовать одно или несколько значений ключа. В последнем случае, потребуется отсортировать содержимое карманов – можно использовать сортировку вставками непосредственно при добавлении элементов. Чтобы получить упорядоченную последовательность необходимо объединить карманы.

Сортировку подсчетом (рисунок 1.16) можно рассматривать как частный случай карманной. В ней первые три цикла для каждого из значений ключа подготавливают заранее объединенные карманы нужного размера. В последнем

цикле элементы исходного массива распределяются по карманам. Для этого требуется не менее 2(k+n) операций, и O(k+n) дополнительной памяти.

Для представления карманов удобно использовать связные списки, это позволит объединять их за линейное время и не заботиться о предварительной оценке размеров. Схема структуры данных показана на рисунке 1.17, для обращения к спискам используется массив, пунктирными линиями показаны изменения связей при объединении списков. Алгоритм карманной сортировки объектов, ключи которых – целые числа, принимающие значения из интервала от m до k-1, представлен на рисунке 1.18. Временная и пространственная сложность данного алгоритма O(k+n).

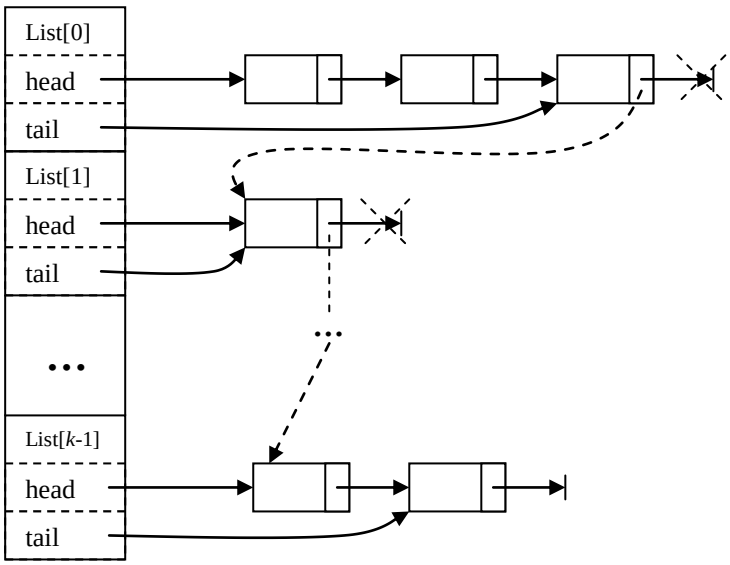

Рисунок 1.17. Объединение списков.

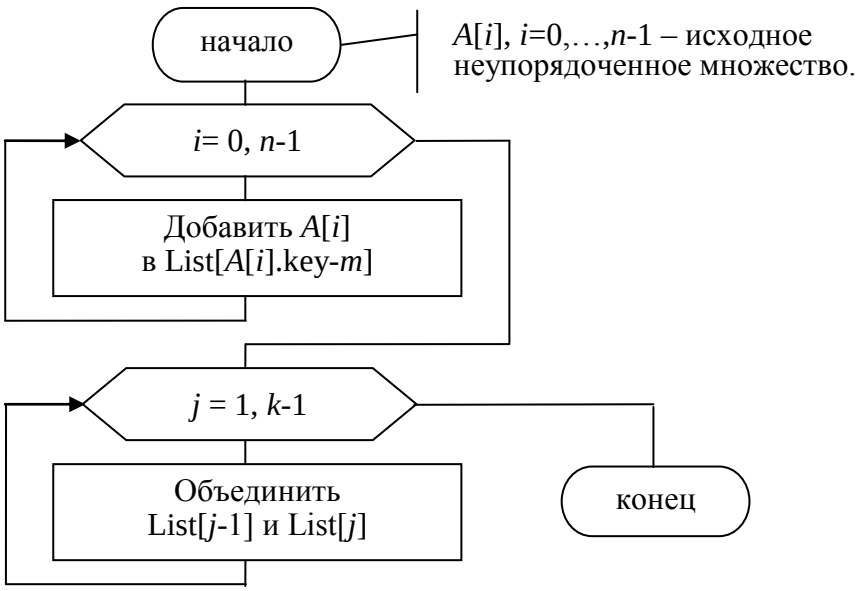

Рисунок 1.18. Алгоритм карманной сортировки.

1.8. Поразрядная сортировка

Два метода рассмотренные выше обладают прекрасной асимптотикой, но они не могут эффективно применяться в случаях больших диапазонов изменения ключа. Поразрядная сортировка позволяет сохранить линейную оценку времени в случаях, когда ключ состоит из нескольких компонент, имеющих небольшое множество возможных значений. В качестве примера можно привести символьные строки ограниченной длины или числа ограниченной разрядности. Компоненты могут быть различных типов, например дата (день; месяц; год), при условии, что область значений года искусственно ограничена.

Поразрядная сортировка заключается в том, что исходное множество отдельно сортируется карманной сортировкой (или подсчетом) по каждому разряду, начиная с младшего. Пример сортировки последовательности двузначных чисел (36, 9, 0, 25, 1, 49, 64, 16, 81, 4) показан на рисунке 1.19.

Разряд 0		Разряд 1	
«Карман»	Содержимое	«Карман»	Содержимое
0	0	0	0, 1, 4, 9
1	1, 81	1	16
2		2	25
3		3	36
4	4, 64	4	49
5	25	5	
6	36, 16	6	64
7		7	
8		8	81
9	9, 49	9	

36	00	00
09	01	01
00	81	04
25	04	09
01	64	16
49	25	25
64	36	36
16	16	49
81	09	64
04	49	81

Рисунок 1.19. Пример поразрядной сортировки.

Пусть ключ состоит из r компонент, которые имеют s_i возможных значений, тогда время выполнения поразрядной сортировки $\sum_{i=1}^{r} O(n+s_i) = O\left(rn + \sum_{i=1}^{r} s_i\right)$.

Первые упоминания о поразрядной сортировке применительно к обработке информации, записанной на перфокартах, появились на рубеже 19 и 20 веков. Сортировка подсчетом и ее применение для поразрядной сортировки впервые предложены в работе [7], опубликованной в 1954 г.

2. СТРОКОВЫЕ АЛГОРИТМЫ

Автоматизированная обработка текстов не теряет своей актуальности. Потребность в ней возникает как при решении простейших прикладных задач, так и в активно развивающихся областях науки, таких как генетика. Анализ цепочек ДНК, разработка компиляторов, реализация полнотекстового поиска в СУБД, словарные методы сжатия данных и, наконец, небольшие программы для банального редактирования описаний товаров в базе данных – всё это области применения строковых алгоритмов.

Прежде всего, следует определить основные термины, использующиеся в этом разделе. **Символ** в данном случае весьма широкое понятие – это информационная единица, определенного типа, которой, как правило, сопоставляется некая печатная форма.

Алфавит – конечное множество символов, обычно обозначают буквой Σ.

Строка (слово) – последовательность символов некоторого алфавита.

Длина строки – количество символов в строке.

Строку обозначают символами алфавита, например $x=x[1]x[2]...x[n]$ – строка длиной n, где $x[i]$ – i -ый символ. Длину строки x обычно обозначают $|x|$.

Конкатенация – сцепление, операция «склеивания» строк. Конкатенацию строк x и y обозначают xy.

Пустая строка – строка, не содержащая ни одного символа.

Подстрока – некоторая непустая последовательность идущих подряд символов строки. Строка x называется **подстрокой** строки y, если найдутся такие строки z_1 и z_2, что $y = z_1xz_2$.

Префикс – некоторое начало строки. Префикс p строки t – строка такая, что $pv=t$ для некоторой (возможно, пустой) строки v. Префикс называется собственным, если $|v| \neq 0$.

Суффикс – некоторое окончание строки. Суффикс s строки t – строка такая, что $vs=t$ для некоторой (возможно, пустой) строки v. Суффикс называется собственным, если $|v| \neq 0$.

2.1. Поиск подстроки

Поиск подстроки в строке является наиболее распространенной задачей. В общем случае она формулируется как поиск всех позиций в некоторой строке *t*, начиная с которых можно прочитать строку *p* (поиск всех вхождений *p* в *t*).

2.1.1. Наивный алгоритм

Наивный алгоритм решения такой задачи состоит в попытке прочитать *p* на каждой позиции *t*. Лучшим примером входных данных будет такая строка *p*, ни один символ которой не совпадает с символами *t*. Пусть $|t|=n$, $|p|=m$, тогда в лучшем случае алгоритм выполнит $(n-m+1)$ операций сравнения символов. Худшим примером можно считать ситуацию, когда обе строки состоят из одного символа, при этом результат поиска – все индексы *t* начиная с первой позиции до $(n-m)$. Количество сравнений в худшем случае $m(n-m+1)$. Работа наивного алгоритма проиллюстрирована на рисунке 2.1, важно отметить, что как в случае полного совпадения *p*, так и при несовпадении очередного символа индекс строки *t* всегда увеличивается на единицу относительно предыдущей попытки.

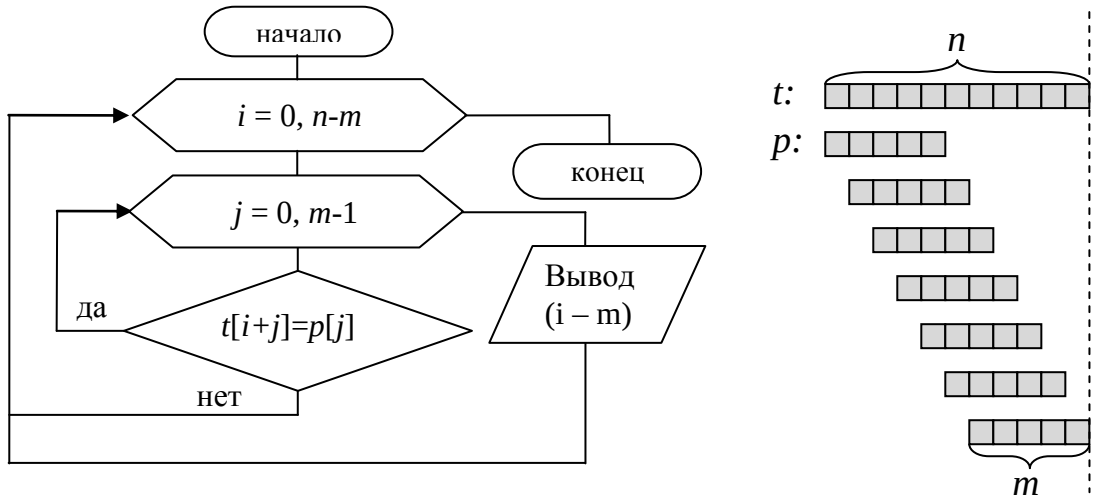

Рисунок 2.1. Наивный алгоритм поиска всех вхождений строки *p* в строку *t*.

2.1.2. Алгоритм Рабина-Карпа

Метод, разработанный в 1987 году Майклом Рабином и Ричардом Карпом [8] использует хеширование. Идея заключается в том, чтобы предварительно сравнивать значения хеш-функции искомого образца и очередного фрагмента строки. В случае совпадения выполняется посимвольная проверка, в противном

случае можно утверждать, что строки не равны без проверки символов. Псевдокод алгоритма представлен на рисунке 2.2., хеш-функция обозначена hash(), фрагмент строки обозначается с помощью индексов: $t[i..j]$ – часть строки t начиная с i-го до j-го символа включительно.

```
(1)   hp = hash(p);
(2)   ht = hash(t[0..(m-1)]);
(3)   Для i от 0 до (n-m) Цикл
(4)         Если hp=ht Тогда
(5)               Если t[i..(i+m-1)]=p Тогда
(6)                     Вывод i;
(7)               Конец если
(8)         Конец если
(9)         ht = hash(t[(i+1)..(i+m)]);
(10)  Конец цикла
```

Рисунок 2.2. Алгоритм Рабина- Карпа.

В условии в строке (5) скрывается посимвольное сравнение, выполняющее порядка m операций. А вычисление хеш-функции в строке (9), напротив – должно осуществляться за постоянное время независимо от значения m. Для этого нужно использовать такую хеш-функцию, которая позволит скорректировать своё значение после удаления первого символа и добавления нового в конце. На примере аддитивного метода для строк (суммирующего коды символов), это будет означать отнять от предыдущего значения код удаляемого символа и прибавить код нового.

Очевидно, что худшее время работы O(nm), но среднее время в сравнении с наивным алгоритмом должно уменьшиться за счет отсутствия сравнений в случае несовпадения хеш-значений. Эффективность будет зависеть от количества коллизий, поэтому предлагается использовать полиномиальный хеш:

$\text{hash}(t[i..j])) = \sum_{k=0}^{j-i} t[i+k]x^k \bmod q$, где x и q – некоторые натуральные числа.

Алгоритм можно применить для поиска множества подстрок, для этого нужно заранее вычислить хеш-функцию для всех искомых образцов, и организовать хранение полученных значений так, чтобы обеспечить минимальное время их поиска.

2.1.3. Префикс-функция

Префикс-функцию строки s в позиции i обозначают $\pi(s,i)$, она равна длине наибольшего собственного префикса строки $s[0..i]$ совпадающего с ее суффиксом. Префикс-функцию строки можно рассматривать как последовательность целых чисел, длина которой равна длине строки. Первым элементом этой последовательности всегда будет значение 0, так как для строки из одного символа не существует собственных префиксов. На рисунке 2.3 приведен алгоритм вычисления префикс-функции позволяющий определить все элементы последовательности за время линейное по отношению к длине строки.

```
(1)   π[0] = 0;
(2)   k = 0;
(3)   Для i от 1 до (n-1) Цикл
(4)         Пока k > 0 И s[k]!=s[i] Цикл
(5)               k=π[k-1];
(6)         Конец цикла
(7)         Если s[k] = s[i] Тогда
(8)               k= k+1;
(9)         Конец если
(10)        π[i] = k;
(11)  Конец цикла
```

Рисунок 2.3. Алгоритм вычисления префикс-функции.

Каждое следующее значение $\pi[i]$ вычисляется с использованием предыдущих. Переменная k отвечает не только за текущее значение префикс-функции, но и является индексом последнего символа некоторого префикса; i продвигается по строке и отвечает за последний символ суффикса. Если k-й и i-й символы совпадают, то этим символом расширяется префикс, совпадающий с суффиксом, а значение префикс-функции увеличивается на единицу. Это соответствует выполнению условия в строке (7), пример обработки строки показан на рисунке 2.4 слева.

Несовпадение символов означает, что текущий «префиксосуффикс» длины k ($s[0..(k-1)]$, равный $s[(i-k)..(i-1)]$) не может быть расширен очередным символом $s[i]$. Чтобы понять насколько уменьшиться значение префикс-функции в позиции i нужно обратиться к префикс-функции самого фрагмента $s[0..(k-1)]$, и попытаться

расширить очередным символом его наибольший «префиксосуффикс». В случае неудачи действия аналогичны. Этим рассуждениям соответствует вложенный цикл, начинающийся в строке (4), пример обработки строки показан на рисунке 2.4 справа.

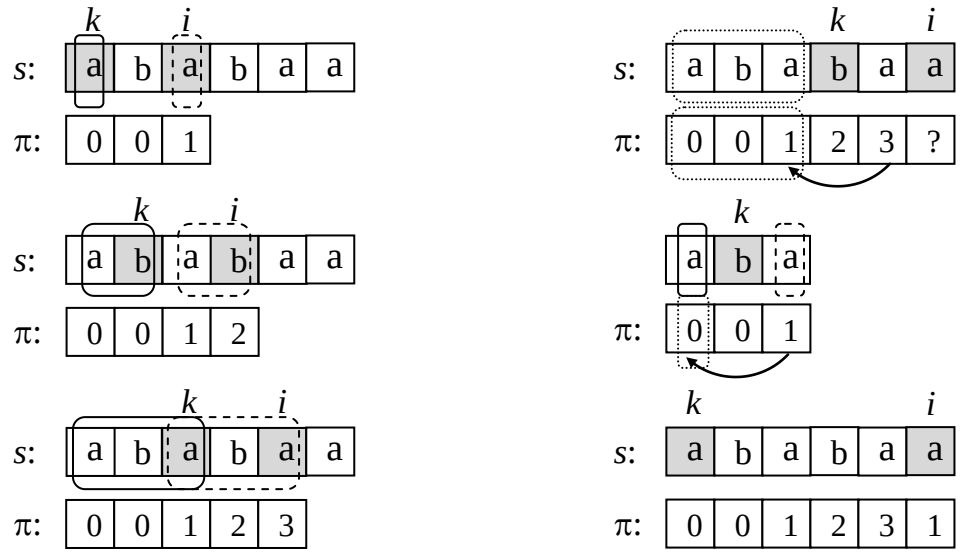

Рисунок 2.4. Пример вычисления префикс-функции строки ‘ababaa’.

В примере показанном на рисунке 2.4 во время вычисления последнего значения осуществляется вход во вложенный цикл, в котором значение *k* падает до нуля, так как ни один «префиксосуффикс» (‘aba’ и ‘a’) не удается расширить последним символом строки *s*. После выхода из цикла *k* увеличивается на единицу, так как символы совпадают. Пример, в котором расширение оказалось возможным, показан на рисунке 2.5.

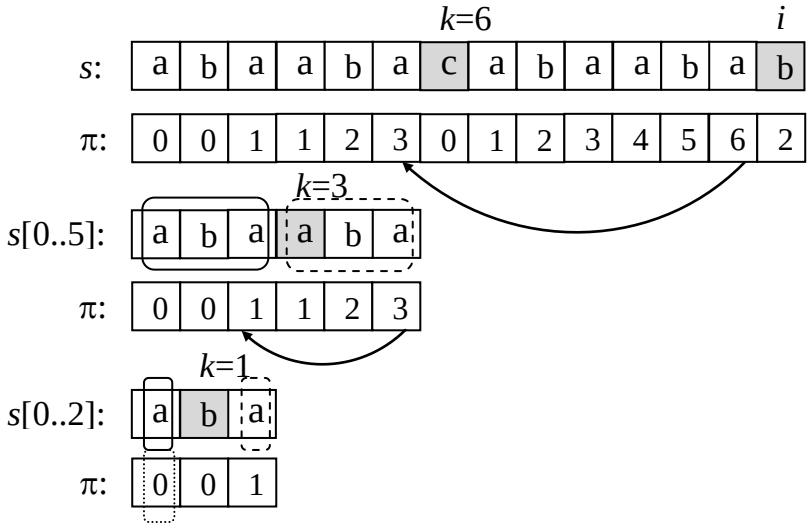

Рисунок 2.5. Пример вычисления префикс-функции строки ‘abaabacabaabab’.

Для доказательства линейной оценки времени данного алгоритма нужно оценить суммарное количество итераций вложенного цикла. На каждом шаге внешнего цикла k увеличивается не более чем на единицу, а во вложенном уменьшается в худшем случае до нуля. Даже если величина уменьшения также будет равна единице, суммарное количество уменьшений не может превосходить количества увеличений и, следовательно, длины строки.

Такой алгоритм вычисления префикс-функции используется в алгоритме Кнута-Морриса-Пратта (КМП). Прежде чем перейти к алгоритму КМП можно рассмотреть упрощенный способ применения префикс-функции для поиска подстроки. Для строки s, сформированной путем конкатенации искомого образца p, добавочного символа ‘$’, отсутствующего в алфавите, и строки t, в которой производится поиск, вычисляется префикс-функция. Если для некоторого i значение $\pi(s,i) = |p|$, то p найдено в t на позиции $(i-|p|+1)$. Пример показан на рисунке 2.6.

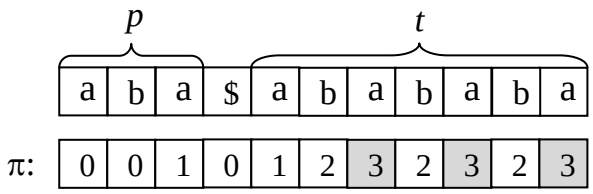

Рисунок 2.6. Пример применения префикс-функции для поиска подстроки.

Очевидным недостатком является объем дополнительной памяти имеющий оценку $O(|p|+|t|)$.

2.1.4. Алгоритм Кнута-Морриса-Пратта

В отличие от упрощенного способа, рассмотренного выше, алгоритм КМП требует вычисления префикс-функции только для искомого образца p. Опираясь на значения $\pi(p,i)$ можно осуществить поиск, выполнив по одному сравнению для каждого символа t. Работу этого алгоритма можно охарактеризовать как оптимизацию сдвигов, пример показан на рисунке 2.7. Каждый раз, когда подстрока найдена полностью или когда очередные символы не совпадают, данный алгоритм изменяет индекс искомого образца с учетом предыстории, которую хранит префикс-функция. Подробный алгоритм приведен на рисунке 2.8.

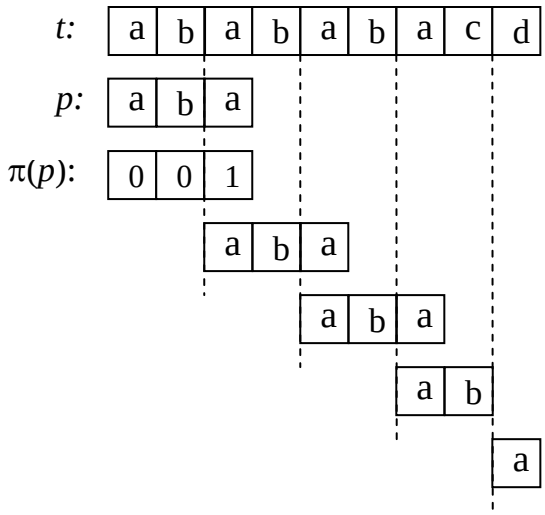

Рисунок 2.7. Алгоритм КМП – оптимизация сдвигов.

```
(1)   Вычислить π(p,i) для i= 0,1,..,m-1;
(2)   j = 0;
(3)   Для i от 0 до (n-1) Цикл
(4)         Пока j > 0 И p[j]!=t[i] Цикл
(5)               j=π[j-1];
(6)         Конец цикла
(7)         Если p[j] = t[i] Тогда
(8)               j= j+1;
(9)         Конец если
(10)        Если j = m Тогда
(11)              Вывод i-j+1;
(12)              j=π[j-1];
(13)        Конец если
(14)  Конец цикла
```

Рисунок 2.8. Псевдокод алгоритма КМП.

Линейная оценка времени работы доказывается аналогично оценке времени вычисления префикс-функции. С учетом вычислений $\pi(p,i)$ общее время работы алгоритма $O(n+m)$; дополнительная память $O(m)$.

2.2. Алгоритм Ахо-Корасик

Алгоритм, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году [9], позволяет найти все вхождения множества строк P_i в строку t, другими словами – поиск слов из словаря P в тексте t. Время работы алгоритма оценивается как $O(\Sigma|P_i|+|t|+l)$, где l – общее количество совпадений, т.е. количество найденных подстрок. В основе алгоритма лежит вспомогательная структура данных для представления искомого множества, первое слагаемое в

оценке времени соответствует построению этой структуры. Текст *t* в процессе поиска будет прочитан один единственный раз, однако в процессе чтения очередного символа, может быть определено появление нескольких искомых слов.

Построение вспомогательной структуры удобнее рассматривать поэтапно, так в первую очередь следует сказать, что это бор, содержащий все искомые строки, на рисунке 2.9 показан бор для словаря *P*={ he, her, hers, him, his, she}.

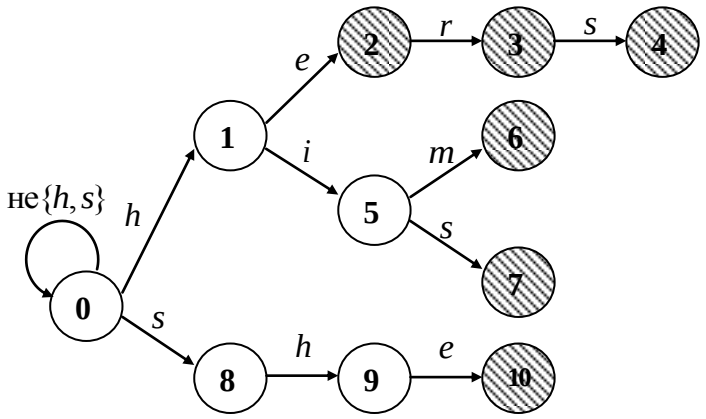

Рисунок 2.9. Нагруженное дерево, содержащее строки he, her, hers, him, his, she.

Вершины обозначены цифрами, корень дерева имеет номер ноль, заштрихованные вершины (2, 4, 6, 7 и 10) соответствуют окончаниям слов. Ребра помечены символами и представляют собой функцию перехода («goto function»), например, из вершины 5 есть путь по символу *m* в вершину 6, и путь в 7 по символу *s*. Петля в вершине 0 предполагает переход по всем символам кроме *h* и *s*. Если, читая текст, в котором осуществляется поиск, перемещаться по такому бору, используя для перехода символы текста, то при удачном стечении обстоятельств, можно оказаться в заштрихованной вершине. Это будет означать, что соответствующая строка найдена, однако дальнейшего пути нет и продолжать поиск невозможно. Кроме того неопределенны дальнейшие действия, в случае когда путь по очередной букве отсутствует. Для решения перечисленных проблем вводится так называемая «failure-function» – «функция отмены». Ее Задача – привести в такую вершину, в которой не будет потеряна прочитанная ранее подстрока, т.е. сохранить прочитанный префикс одной из строк словаря.

Очевидно, что failure-function корня и всех его непосредственных потомков ведет обратно в корень (если прочитана одна буква и далее нет продолжения, то терять нечего). Для остальных вершин построение осуществляется в соответствии с алгоритмом, представленным на рисунке 2.10.

Для каждой вершины *w* **Цикл**
    Перейти к родительской вершине *r* запомнив букву α на ребре.
    Перейти по failure-function(*r*) в вершину *v*.
    Если из *v* есть путь по α в вершину *u* **Тогда**
        failure-function(*w*)=*u*.
    Иначе
        failure-function(*w*)= «корень».
    Конец если
Конец цикла

Рисунок 2.10. Вычисление failure-function.

Для рассмотренного примера значения failure-function некоторых вершин показаны пунктирными стрелками на рисунке 2.11, полный список значений failure-function приведен в таблице 2.1.

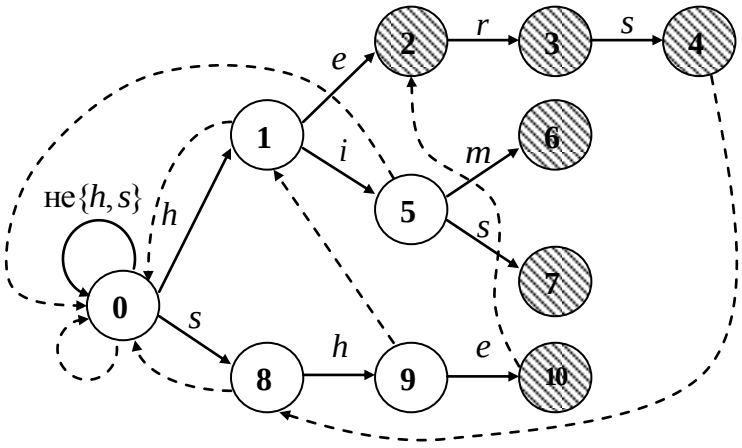

Рисунок 2.11. Бор с добавлением failure-function к вершинам 0, 1, 4, 5, 8, 9, 10.

Таблица 2.1. Значения failure-function.

i=номер вершины	0	1	2	3	4	5	6	7	8	9	10
failure-function(*i*)	0	0	0	0	8	0	0	8	0	1	2

Использование failure-function позволит передвигаться по бору и, попадая в заштрихованные вершины, делать вывод о найденных подстроках, однако возможность ошибки всё еще присутствует. На рисунке 2.12 показан бор для словаря, в котором изменено одно слово: вместо строки ‘she’ добавлена строка ‘shelf’. В процессе чтения этого слова (по пути из восьмой вершины в девятую, а затем в десятую) будет прочитано слово ‘he’, но соответствующая вершина не заштрихована, и, следовательно, сигнала о его нахождении не будет.

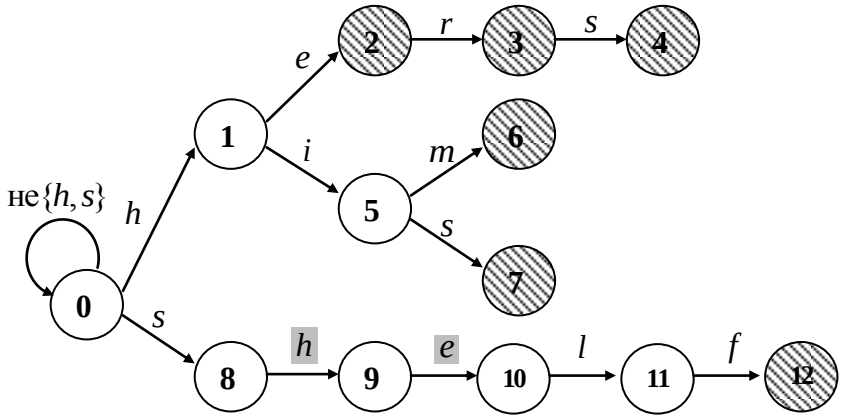

Рисунок 2.12. Нагруженное дерево для словаря *P*={ he, her, hers, him, his, shelf.}.

Для корректного нахождения всех слов предусмотрен последний компонент структуры данных – список вывода (output function). Для каждой вершины формируется список слов, которые в ней заканчиваются, реализуется это с помощью связных списков, на рисунке 2.13 показаны элементы списков для некоторых вершин.

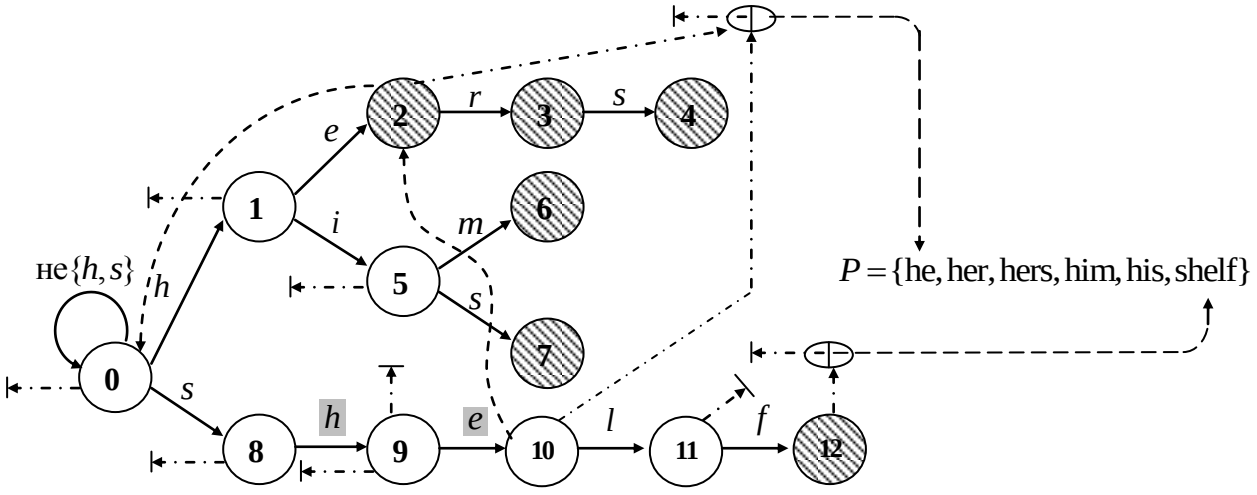

Рисунок 2.13. Указатели на список вывода для вершин 0, 1, 2, 5, 8, 9, 10 ,11, 12.

Связи элементов списка вывода обозначены штрихпунктирной линией, а длинной штриховой линией обозначены указатели на слова соответствующие элементам. Формирование списка происходит по следующим правилам. Корень имеет пустой указатель. Если в вершине заканчивается слово, то создается новый элемент списка, на который она будет указывать. Новый элемент будет некоторым образом связан со словом; а «указатель на следующий» устанавливается на элемент, на который указывает failure-function вершины. В противном случае вершина будет указывать туда же, куда указывает ее failure-function. В рассмотренном примере в вершине 10 не заканчивается ни одно слово, а failure-function ведет в вершину 2, которая указывает на элемент списка связанный со словом ‘he’. Failure-function вершины 2 ведет в корень, поэтому указатель на следующий элемент списка пустой. Для демонстрации окончания нескольких слов в одной вершине нужно вернуться к предыдущей версии словаря со словом ‘she’, список вывода для этого примера частично показан на рисунке 2.14. В данном случае в вершине 10 заканчивается слово‘she’, и ей соответствует элемент списка вывода, следующий за которым (согласно failure-function) – элемент, соответствующий вершине 2 и слову ‘he’.

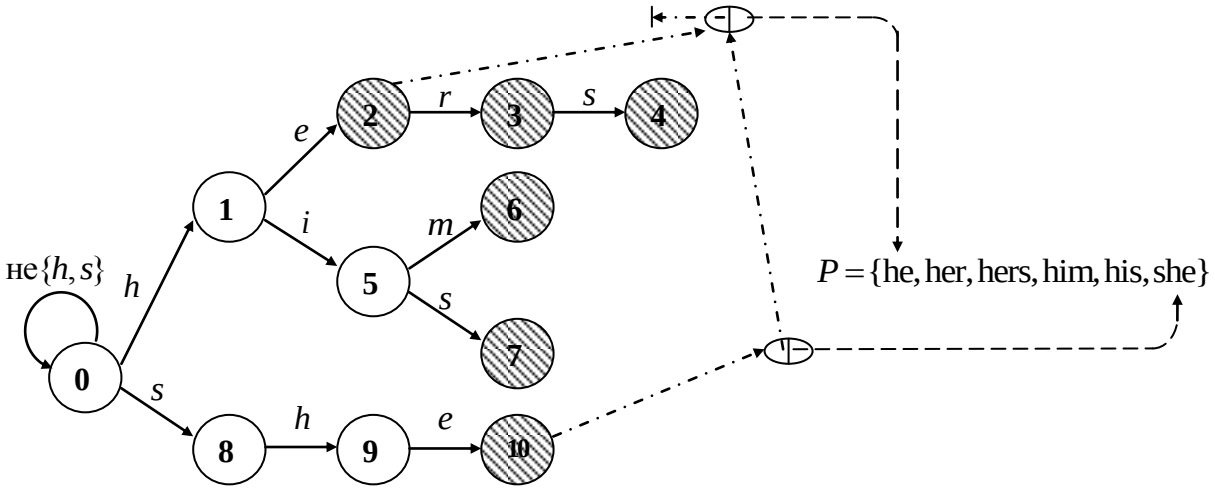

Рисунок 2.14. Указатели и элементы списка вывода для вершин 2 и 10.

Алгоритм, использующий построенную структуру, представлен на рисунке 2.15. Очевидно, что фаза поиска без учета вывода осуществляется за O(|*t*|).

```
Текущая вершина w = «корень».
Для i от 0 до |t|-1 Цикл
        Если из w есть ребро помеченное t[i] Тогда
                Переход по ребру w = goto(w, t[i]);
        Иначе
                Переход по failure-function w = failure-function(w);
        Конец если
        Установить текущий элемент списка вывода x = w.out;
        Пока x != «пустой указатель» Цикл
                Вывод «найдена строка x.text»;
                Переход к следующему элементу x = x.next;
        Конец цикла
Конец цикла
```

Рисунок 2.15. Алгоритм поиска.

2.3. Суффиксные деревья

Суффиксное дерево – это нагруженное дерево, содержащее все суффиксы некоторой строки. Можно сказать, что это специальный способ представления строки, позволяющий быстро решать множество задач. Автором идеи является Питер Вейнер, применение суффиксных деревьев для поиска впервые опубликовано в 1973 году в работе [10].

Суффиксное дерево занимает большой объем памяти, пример дерева без каких-либо оптимизаций для строки ‘acacg’ показан на рисунке 2.16 слева. Следует отметить одно важное свойство – каждый суффикс должен заканчиваться в своем листе, для его выполнения последний символ строки не должен совпадать с каким-либо другим символом. Для гарантированного выполнения указанного свойства к строке, не глядя на ее содержимое, добавляют специальный уникальный символ, которого нет в исходном алфавите. На рисунке 2.16 справа показано суффиксное дерево для строки ‘aaa’. Листья дерева обычно нумеруют, их номера совпадают с позициями начала соответствующих суффиксов.

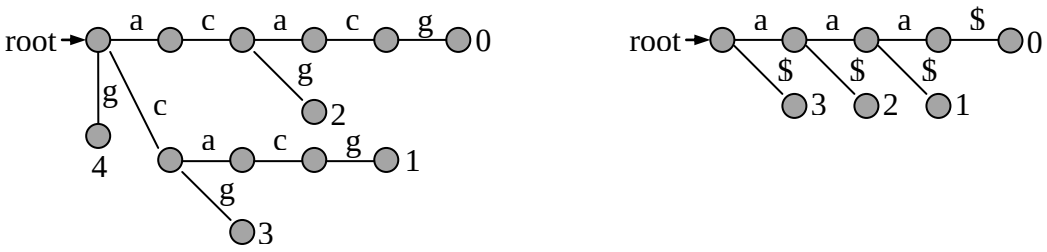

Рисунок 2.16. Несжатые суффиксные деревья для строк ‘acacg’ и ‘aaa’.

Эдвард МакКрейт в 1976 году предложил использовать сжатое суффиксное дерево [11], в котором присутствуют только те вершины, в которых есть ветвления. Ребра в реализации сжатого дерева помечаются двумя числами – начальным и конечным индексами соответствующего ребру фрагмента строки. При изображении сжатых деревьев лучше воспринимаются пометки в виде последовательности символов. Пример сжатого дерева показан на рисунке 2.17. В настоящее время, говоря о суффиксных деревьях, подразумевают сжатые.

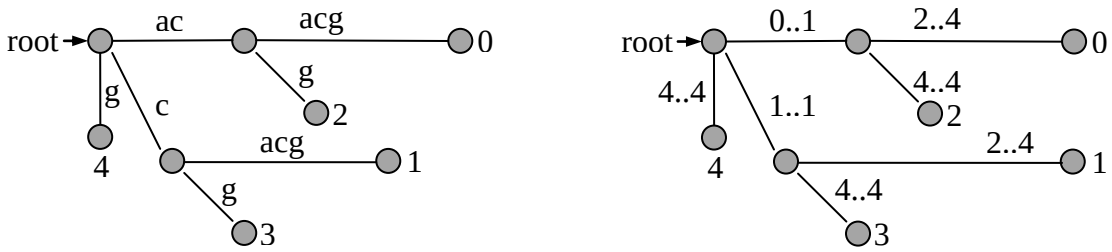

Рисунок 2.17. Сжатое суффиксное дерево для строки ‘acacg’.

Количество вершин в сжатом дереве для строки t растет линейно относительно ее длины. Можно привести точную оценку – количество вершин не находится в интервале от $|t|+2$ до $2|t|+1$. Вершины можно разделить на листья и внутренние (вершины ветвления). Количество листьев совпадает с количеством суффиксов и, следовательно, совпадает с количеством символов, что с учетом дополнительного символа дает ровно $|t|+1$ вершин. Вершины ветвления за исключением корня могут полностью отсутствовать, либо появляться в процессе ответвления каждого суффикса кроме самого длинного (идущего из корня в первый лист) и самого короткого (ребро в последний лист помечено одним символом). Таким образом максимальное количество вершин ветвления включая корень – это $|t|$, плюс дополнительный символ, минус два и плюс корень. Примеры деревьев для крайних случаев приведены на рисунке 2.18.

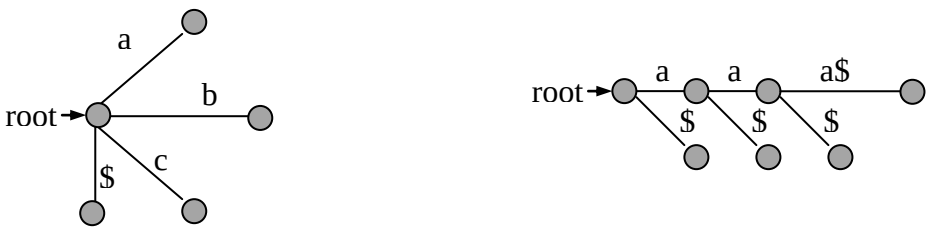

Рисунок 2.18. Сжатые суффиксные деревья для строк ‘abc’ и ‘aaa’.

2.3.1. Построение: наивные алгоритмы

Интуитивно понятным представляется добавление суффиксов начиная с самого длинного. Добавление каждого суффикса предполагает попытку прочитать его от корня и в момент, когда чтение невозможно (пути по очередному символу нет) создание ответвления и нового листа. Более подробно алгоритм представлен на рисунке 2.19, пример его выполнения для строки ‘acacg’ показан на рисунке 2.20.

```
(1)   Создать корень дерева;
(2)   n = |t|;
(3)   Для i от 0 до (n-1) Цикл
(4)       w = «корень дерева»;
(5)       j = i;
(6)       Если Есть путь из w по символу t[j] Тогда
(7)           Для k от «метка начала ребра» до «метка окончания ребра» Цикл
(8)               Если t[j] = t[k] Тогда
(9)                   j = j+1;
(10)              Иначе
(11)                  Создать новую внутреннюю вершину v;
(12)                  Добавить лист u с номером i;
(13)                  Добавить ребро из v в u помеченное t[j..n];
(14)                  Перейти к следующей итерации цикла (3);
(15)              Конец если
(16)          Конец цикла
(17)          w = «очередная вершина»;
(18)          Перейти к шагу (6);
(19)      Конец если
(20)      Добавить лист u с номером i;
(21)      Добавить ребро из w в u помеченное t[j..n];
(22)  Конец цикла
```

Рисунок 2.19. Наивный алгоритм построения суффиксного дерева.

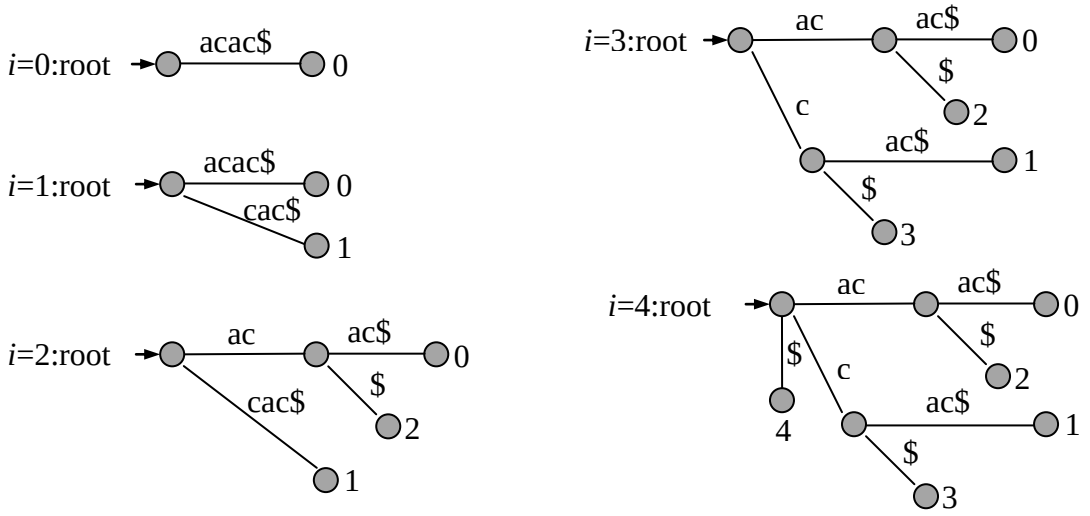

Рисунок 2.20. Иллюстрация работы наивного алгоритма для строки ‘acac’.

Основной цикл делает $|t|$ шагов, на каждом из которых производится $O(|t|)$ сравнений символов. Итоговая оценка времени работы данного алгоритма $O(|t|^2)$.

Альтернативный подход заключается в постепенном увеличении суффиксного дерева, на каждом из $|t|$ шагов изначально пустое дерево расширяется очередным символом строки. Такие алгоритмы называют online алгоритмами, так как они позволяют строить дерево в процессе ввода строки, что позволяет более рационально использовать вычислительные ресурсы.

Суффиксное дерево, которое строится в online алгоритме для некоторого префикса строки, называют неявным. В таком дереве суффиксы могут заканчиваться на ребре или во внутренней вершине. Пример неявного дерева показан на рисунке 2.21.

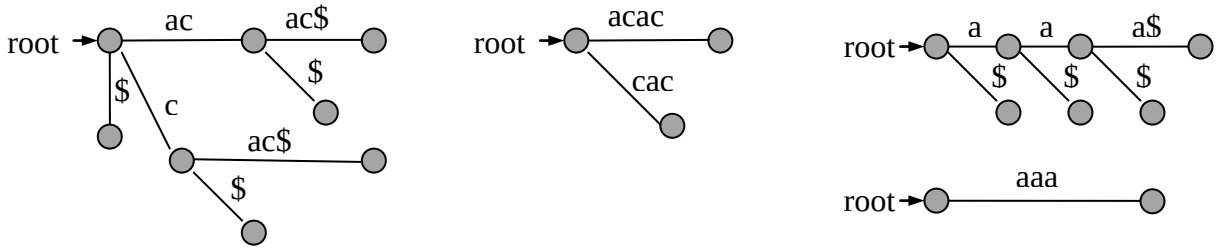

Рисунок 2.21. Явные и неявные суффиксные деревья строк ‘acac’ и ‘aaa’.

Абстрактный online алгоритм, представленный на рисунке 2.22, может иметь как квадратичную, так и кубическую оценку времени работы, в зависимости от способа реализации продления суффикса во внутреннем цикле. Если никаких оптимизаций нет, и продление суффикса начинающегося в позиции j выполняется за $(i\text{-}j)$ операций (путем его чтения от корня), то время работы алгоритма оценивается как $O\left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i-j)\right)=O\left(n^3\right)$.

```
(1)     Создать корень дерева;
(2)     Для i от 0 до (|t|-1) Цикл
( )             //Добавление символа t[i] в дерево.
(3)             Для j от 0 до i-1 Цикл
(4)                     Продлить суффикс t[j..i-1] символом t[i].
(5)             Конец цикла
(6)             Добавить суффикс t[i..i] в дерево.
(7)     Конец цикла
```

Рисунок 2.22. Абстрактный online алгоритм построения суффиксного дерева.

Последовательность деревьев, построенных по данному алгоритму для строки ‘abca’ показана на рисунке 2.23. Для перехода к квадратичному, а затем и линейному варианту online алгоритма целесообразно вначале рассмотреть оптимизацию алгоритма представленного на рисунке 2.19. Способы построения линейных алгоритмов приведены в разделе 2.3.4.

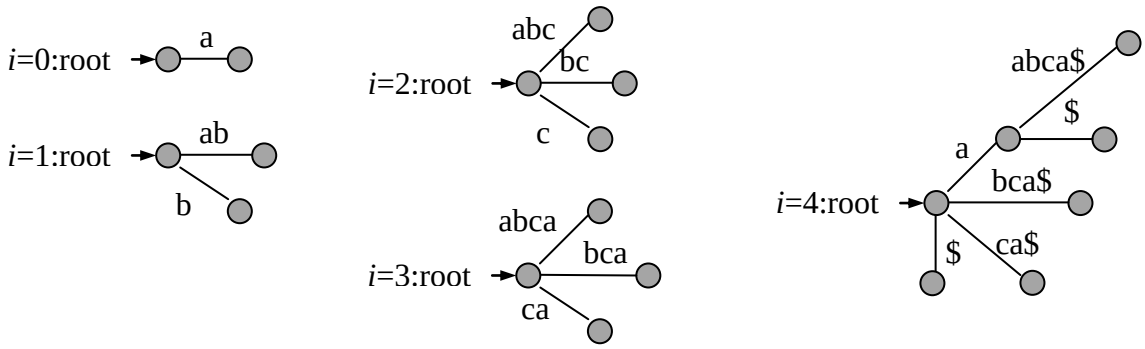

Рисунок 2.23. Построение суффиксного дерева строки ‘abca’, online алгоритм.

2.3.2. Применение: поиск подстроки

Суффиксное дерево, при условии, что оно уже построено для строки *t*, поможет ответить на вопрос о том, встречается ли образец *p* в тексте *t* за время линейное относительно |*p*|. Для получения ответа достаточно попытаться прочитать искомый образец в дереве.

Время поиска всех вхождений, как и в алгоритме КМП, O(|*t*|). Для поиска вех позиций текста, начиная с которых можно прочитать искомый образец, нужно продолжить обход дерева в глубину. Дойдя до листьев, вывести их номера. Поясняющий пример показан на рисунке 2.24.

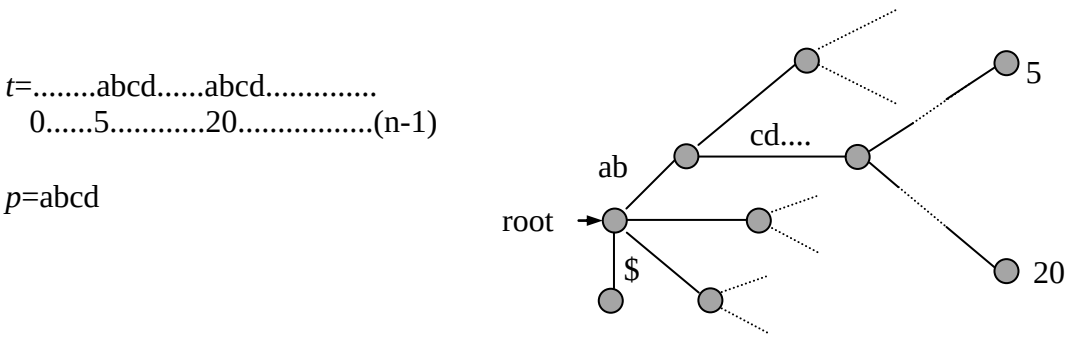

Рисунок 2.24. Поиск подстроки с использованием суффиксного дерева.

2.3.3. Применение: наибольшая общая подстрока

Задачу можно сформулировать так: найти строку наибольшей длины, которая одновременно является подстрокой двух строк t_1 и t_2. Другая задача,

которая решается похожим образом – это поиск наибольшей повторяющейся подстроки в некоторой строке *t*. Для ее решения построим суффиксное дерево строки *t* и найдем в нем самую удаленную от корня внутреннюю вершину. Текст, читающийся по пути от корня в эту вершину, и есть искомая подстрока. Объясняется это тем, что внутренняя вершина суффиксного дерева это место, начиная с которого некоторый суффикс перестал совпадать с каким-то другим суффиксом.

Для поиска наибольшей общей подстроки строк t_1 и t_2 вводят строку $t = t_1\#t_2$, где ‘#’ – уникальный символ. Для строки t строится суффиксное дерево. При этом вершины, появившиеся при добавлении суффиксов, начинающихся в t_1 и в t_2 помечаются различными пометками. Наибольшая общая подстрока читается по наибольшему пути от корня к вершине имеющей обе отметки. Пример дерева для решения такой задачи показан на рисунке 2.25.

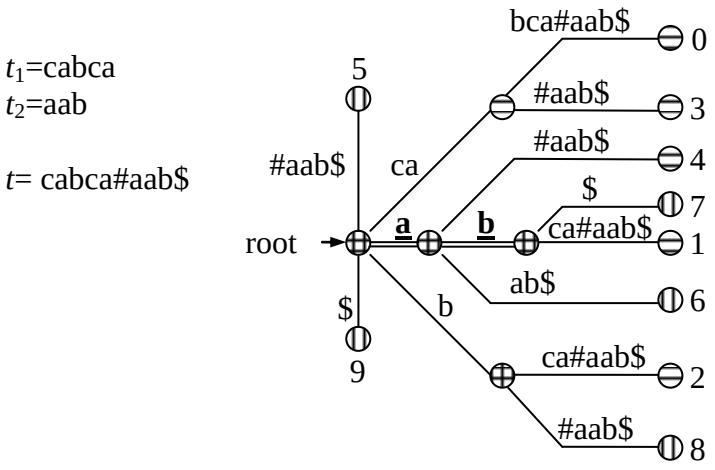

Рисунок 2.25. Поиск наибольшей общей подстроки.

2.3.4. Построение за линейное время

Все алгоритмы построения суффиксных деревьев, работающие за линейное по отношению к длине строки время, используют дополнительные данные - суффиксные ссылки. Суффиксная ссылка, идущая из вершины *u*, в которой читается строка *t*[*i*..*j*], ведет в такую вершину *w*, в которой читается строка *t*[*i*+1..*j*] (отличается от строки, читающейся в вершине *u*, отсутствием первой буквы). В примере, изображенном на рисунке 2.26, суффиксные ссылки показаны пунктирными стрелками.

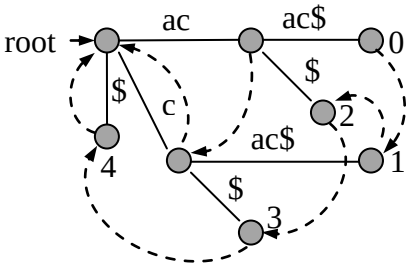

Рисунок 2.26. Суффиксные ссылки в дереве для строки ‘acac’.

Для удобства дальнейшего изложения материала требуется ввести ряд дополнительных терминов. ***Положение в дереве*** называют ***явным,*** если оно соответствует какой-либо уже существующей вершине. Если говорят о каком-либо положении в середине ребра, его называют ***неявным***, иногда говорят о неявной (отсутствующей) вершине.

Алгоритм Маккрейта добавляет суффиксы в том же порядке как первый алгоритм, рассмотренный в пункте 2.3.1 – начинает с самого длинного. Прежде чаем разобраться, как суффиксные ссылки помогают при построении дерева, важно понять в какой момент они появляются. Глядя на пример построения дерева, показанный на рисунке 2.27, можно заметить, что вершина, в которую будет направлена суффиксная ссылка, появляется на следующем шаге – при добавлении следующего по порядку суффикса. Отсутствие на рисунке ссылок для листьев объясняется тем, что они не используются в алгоритме. В любом случае суффиксные ссылки для листьев излишни, так как ссылка, ведущая из листа с номером *i*, всегда будет направлена в лист (*i*+1).

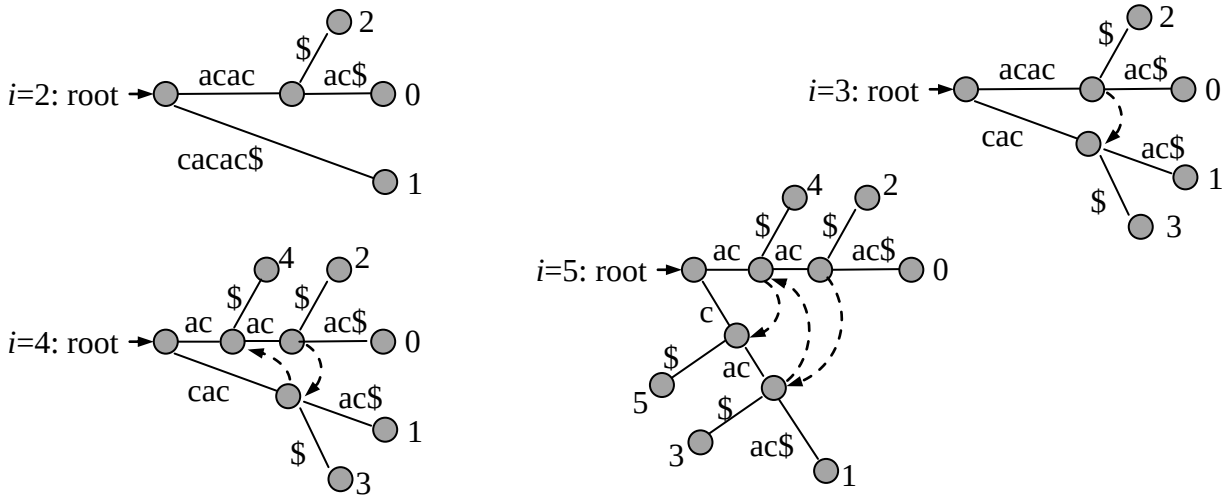

Рисунок 2.27. Порядок появления суффиксных ссылок.

Как уже было сказано при добавлении каждого суффикса, кроме первого и последнего, может быть создана новая внутренняя вершина. Каждому суффиксу соответствует свой собственный лист. Если назвать часть суффикса, которую не удалось прочитать в дереве ***хвостом*** (tail) этого суффикса. Тогда для достижения линейной оценки времени требуется за постоянное, не зависящее от длины строки время находить положение в дереве, от которого будет ответвляться новый лист и хвост соответствующего суффикса. Иными словами если строки (2) и (3) алгоритма представленного на рисунке 2.28 будут выполняться за O(1), тогда весь алгоритм будет иметь оценку O(n).

```
(1)  Для i от 0 до (|t|-1) Цикл
(2)      Найти положение w, от которого ответвляется i-ый лист;
(3)      Найти tail_i, –хвост i-го суффикса;
(4)      Если положение w неявное Тогда
(5)          Сделать положение w явным, создав новую внутреннюю вершину;
(6)      Конец если
(7)      Добавить лист u_i;
(8)      Добавить ребро из w в u_i помеченное tail_i;
(9)  Конец цикла
```

Рисунок 2.28. Абстрактный алгоритм построения суффиксного дерева.

На рисунке 2.29 показан фрагмент дерева после добавления суффикса с номером i. Согласно своему определению, суффиксная ссылка должна помочь попасть на путь, от которого будет ответвляться следующий (i+1)-й суффикс (при условии, что некоторый его префикс уже читается в дереве, в противном случае ответвление должно быть от корня). Для вершины z, созданной на предыдущем шаге, суффиксная ссылка еще не определена, но ее можно найти выше – у родителя этой вершины. Путь, пройденный с использованием суффиксной ссылки, показан штрихпунктирной линией. Очевидно, что символы, прочитанные по пути к родителю, должны вновь встретиться по пути вниз, после перехода по ссылке.

Если положение w' оказалось явным, тогда следует пытаться прочитать продолжение (i+1)-го суффикса по одному из путей, выходящих из данной вершины. Это объясняется тем, что подстрока ‘bcd..pqrst’ могла встречаться раньше и иметь совпадающее некоторым количеством символов ‘uv..’

продолжение (полного совпадения не может быть, так как последний символ уникален). Аналогично объясняется возможность существования вершины между символами ‘pq’ и ‘rst’ – подстрока ‘bcd..pq’ могла встретиться ранее с альтернативным продолжением.

Если же *w'* распложено на ребре, то нужно разорвать ребро и добавить вершину, так как продолжения пути символами ‘uv..’ не может быть. Доказать это нетрудно. Если бы подстрока ‘abcd..pqrst’ имела продолжение ‘uv..’, то вершина *z* оказалась бы глубже, следовательно, она имела другое продолжение. В таком случае, при наличии где-то еще подстроки ‘bcd..pqrstuv..’, *w'* обязано быть явным, а это не так.

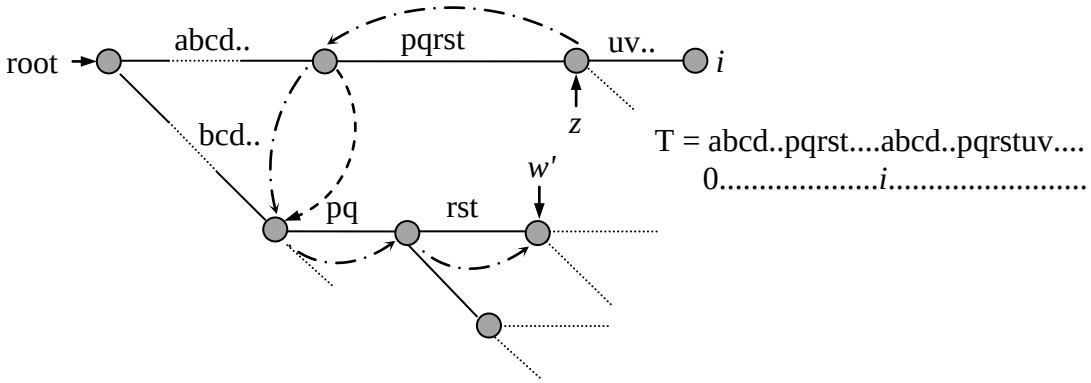

Рисунок 2.29. Использование суффиксных ссылок.

Таким образом, в алгоритме Маккрейта контролируя только последний добавленный лист можно отказаться от обязательного чтения суффиксов от корня. По мере увеличения дерева количество разнообразных префиксов, соответствующее количеству внутренних вершин будет увеличиваться и, соответственно, будет всё больше возможностей экономить на сравнениях символов, перемещаясь по ссылкам внутри дерева. Однако для достижения линейного времени этого недостаточно. Если, например, строка представляет собой повторение одного символа, то выгода от использования суффиксных ссылок полностью пропадет.

Для продвижения по дереву используются две процедуры. Первая ищет нужный путь проверяя каждый символ, ее называют «scanning». Она применяется для уточнения положения в ситуациях, когда *w'* – явная вершина. Вторая находит путь, который гарантированно присутствует в дереве – «rescanning». Она

используется после перехода по суффиксной ссылке, когда необходимо прочитать текст, сохраненный по пути к родителю вершины *z*. Особенность этой процедуры в том, что для нее нет необходимости сравнивать каждый символ, если она идет по ребру, то может пропустить несколько символов, доходя либо до конца ребра, либо до конца искомого текста. Если на пути встречается вершина, то необходимо сравнить всего один символ для выбора направления. Временная сложность процедуры «rescanning» линейна относительно количества пройденных вершин.

Отправной точкой при добавлении очередного (*i*+1)-го суффикса считается родитель *i*-го (добавленного на предыдущем шаге) листа, обозначают его $head_i$. Эта вершина не обязательно должна быть добавлена на предыдущем шаге, она могла появиться раньше. Подстроку соответствующую этой вершине можно назвать наибольшим префиксом *i*-го суффикса, который появился в дереве до начала *i*-го шага. Корень и нулевой лист создаются до начала итерационного процесса. Позиция $head_0$ соответствует корню. Псевдокод алгоритма, представленный на рисунке 2.30, основан на идеях изложенных Маккрейтом, однако не является точной копией оригинального алгоритма. Листовые вершины обозначены $leaf_i$, переход по суффиксной ссылке вершины *u* обозначается *f*(*u*). Строка, написанная на ребре, ведущем из $head_i$ в $leaf_i$, обозначается $tail_i$. Процедура scan(*x*, *y*) начиная в вершине *x*, ищет строку *y* и возвращает пару (*x*', *y*'), где *x*' – положение в дереве, начиная с которого невозможно продолжать чтение, а *y*' – остаток строки *y*. Процедура rescan(*x*, *y*) возвращает такое положение, в котором, начиная с вершины *x*, читается строка *y*.

Строка псевдокода (20) требует пояснений. Если родитель вершины $head_i$ оказался корнем, то дальнейший переход по суффиксной ссылке невозможен. Но поскольку $head_i$ явная внутренняя вершина, подстрока *v* встречается как минимум второй раз с различными продолжениями. Из этого следует, что все суффиксы *v*, включая *v*[1..|*v*|-1] уже читаются в дереве. Объяснить это можно тем, что либо все суффиксы строки *t* начинающиеся суффиксами *v* уже добавлялись, либо *v* состоит из повторяющегося символа. Примеры строк и соответствующих им деревьев изображены на рисунке 2.31.

Разветвления алгоритма в строке (19) можно было бы избежать, добавив фиктивную вершину SuperRoot, в которую из корня ведет суффиксная ссылка. А из SuperRoot в корень по каждому символу алфавита нужно добавить ребро, проход по которому и будет выполнять удаление первого символа *v*.

```
(1)   Создать «корень»;
(2)   head_0 = «корень»;
(3)   tail_0 = t[0.. |t|-1];
(4)   Создать leaf_0, ребро из head_0 в leaf_0, с пометкой tail_0;
(5)   Для i от 0 до (|t|-2) Цикл
(6)       Если head_i = «корень» Тогда
(7)           (head_{i+1} , tail_{i+1})= scan(«корень», tail_i[1..| tail_i|-1]);
(8)           Если head_{i+1} это неявное положение Тогда
(9)               Создать новую вершину для положения head_{i+1};
(10)          Конец если
(11)          Создать leaf_{i+1}, ребро из head_{i+1} в leaf_{i+1}, с пометкой tail_{i+1};
(12)          Перейти к следующей итерации цикла;
(13)      Конец если
(14)      Если Существует f(head_i) Тогда
(15)          w = f(head_i);
(16)      Иначе
(17)          u = родитель head_i;
(18)          v = метка ребра между u и head_i;
(19)          Если u = «корень» Тогда
(20)              w=rescan(«корень»,v[1..|v|-1]);
(21)          Иначе
(22)              w=rescan(f(u),v);
(23)          Конец если
(24)      Конец если
(25)      Если w это неявное положение Тогда
(26)          Создать новую вершину для положения w;
(27)          head_{i+1} = w;
(28)          tail_{i+1}= tail_i;
(29)      Иначе
(30)          (head_{i+1} , tail_{i+1})= scan(w, tail_i);
(31)          Если head_{i+1} это неявное положение Тогда
(32)              Создать новую вершину для положения head_{i+1};
(33)          Конец если
(34)      Конец если
(35)      Создать f(head_i) = w;
(36)      Создать leaf_{i+1}, ребро из head_{i+1} в leaf_{i+1}, с пометкой tail_{i+1};
(37)  Конец цикла
```

Рисунок 2.30. Линейный алгоритм построения суффиксного дерева.

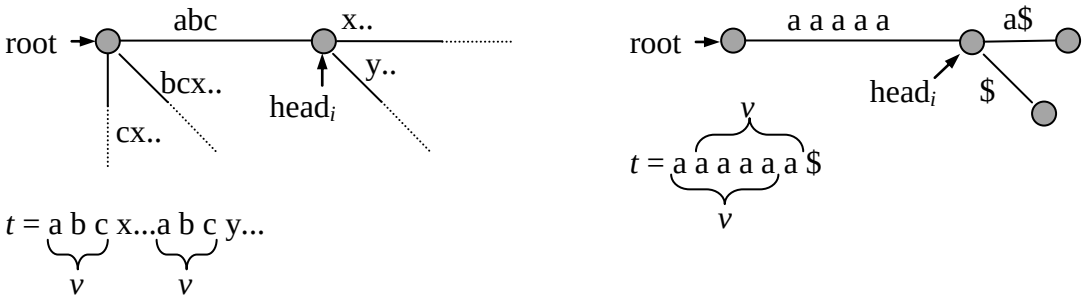

Рисунок 2.31. Примеры повторяющихся подстрок в суффиксном дереве.

На отдельном шаге номер i процедура scan выполняет не более ($|head_{i+1}|-|head_i|+1$) операций, где $|head_i|$ – длина строки, соответствующей в вершине $head_i$. Суммарно для всех i количество операций ограничено величиной ($|head_{(|t|-1)}|-|head_0|+n$). Переход по суффиксной ссылке, предшествующий вызову rescan, уменьшает глубину вершины не более чем на 1. Количество вершин, которое может встретиться по пути вниз, при выполнении rescan на некотором шаге, не имеет обоснованного ограничения, однако суммарное количество переходов вниз не может превышать количества переходов вверх. С учетом того, что общее количество внутренних вершин не превосходит $|t|$, суммарное количество операций в процедуре rescan по всем i есть $O(|t|)$. Все остальные операции, выполняющиеся на шаге цикла, имеют сложность O(1), что приводит к итоговой оценке времени работы алгоритма $O(|t|)$.

Понимание линейных алгоритмов осложняется тем, что практически невозможно привести конкретный пример, который одновременно будет достаточно компактным для изображения и сможет наглядно продемонстрировать преимущества алгоритма. На рисунке 2.32 приведен пример построения дерева для строки ‘aaacaaac’ в момент добавления суффикса ‘aac$’. Пример показывает добавление суффикса без посещения корня дерева.

Алгоритм, разработанный в 1995 году финским математиком Эско Укконеном [12], является online алгоритмом, работающим за линейное время. Глядя на пример построения суффиксного дерева наивным online алгоритмом, показанный на рисунке 2.23, можно увидеть два варианта изменений при продлении суффикса очередным символом. Каждый суффикс, который

заканчивается в своем собственном листе продляется новым символом. Суффиксы, которые заканчиваются на ребре (в неявном дереве это возможно) при продлении могут продляться ответвлением нового символа. Н расунке 2.33 показан другой пример, иллюстрирующий ситуацию, в которой суффикс ‘abc’ уже читается в дереве, и для его продления не нужно никаких дополнительных действий. Нужно отметь, что в этом случае все меньшие по длине (следующие по порядку) суффиксы тоже не нуждаются в продлении.

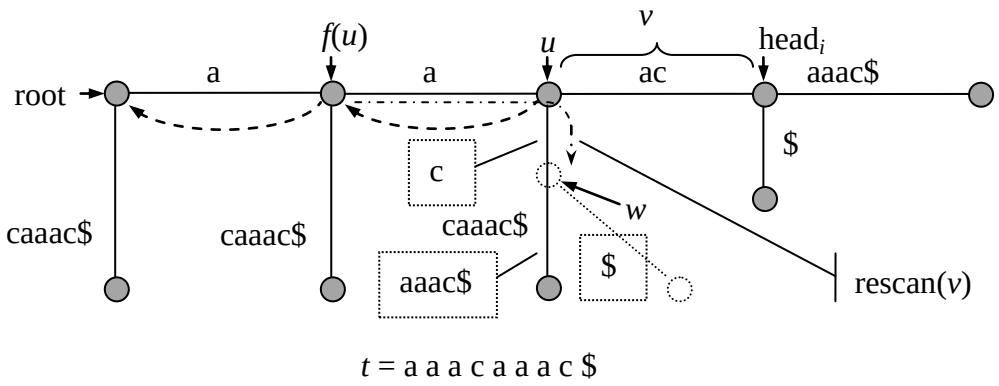

Рисунок 2.32. Построение суффиксного дерева: добавление суффикса ‘aac$’.

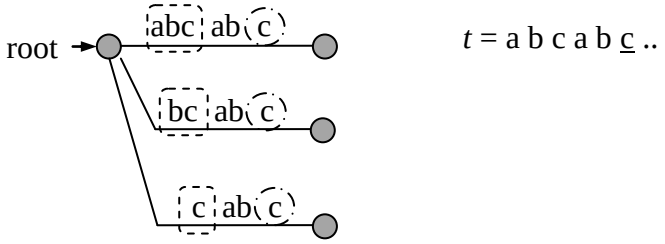

Рисунок 2.33. Продление строки ‘abcab’ символом ‘c’.

Наиболее простая оптимизация заключается в том, чтобы оставлять метку ребер ведущих в листья «открытой». То есть некоторым образом связать позиции окончания всех таких ребер с номером текущего шага. Тогда все продления данного вида на отдельном шаге можно выполнить за O(1).

Основная работа связана с продлениями второго вида – ответвлениями новых букв. В отличие от предыдущего алгоритма ребро, ведущее в новый лист, всегда помечено одним символом, но количество таких ответвлений, создаваемых в процессе выполнения одного шага (добавления очередного символа) может быть различным, и достигает общего количества суффиксов (находящихся в

дереве на данный момент). На рисунке 2.34 изображен простой пример, в котором к неявному дереву для строки ‘aabb’ добавляется символ ‘a’. Первый и единственный суффикс, который требует ответвления это ‘ba’.

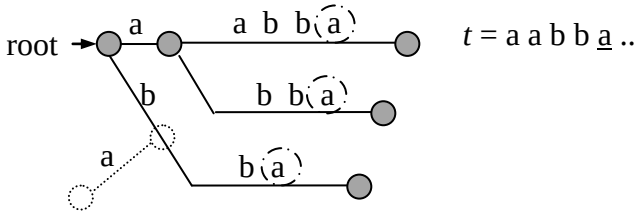

Рисунок 2.34. Продление строки ‘aabb’ символом ‘a’.

Два ключевых понятия в алгоритме Укконена – «active point» и «end point», их можно перевести как начальная и конечная точка. Начальная точка представляет собой положение в дереве, начиная с которого требуется создавать ответвления. Конечная точка связана с первым по порядку суффиксом, который уже продлялся текущим символом. Говоря точнее, это положение, начиная с которого очередной суффикс уже имеет продолжение текущим символом. В примере на рисунке 2.34 конечная точка – это корень. Начальная точка в рассмотренном примере расположена на ребре. Для обращения к произвольному положению в дереве, предлагается использовать так называемую «reference pair», что можно перевести как ссылочная пара. Она состоит из явной вершины и строки, которую необходимо прочитать, начиная из этой вершины, что приведет к заданному положению. Обозначают ссылочную пару $(s,(k,i))$, где s – явная вершина, а k и i определяют строку $t[k..i]$. Ссылочная пара может характеризоваться как «canonical» – каноническая – стандартная или нормализованная. Так называют пару, вершина в которой является ближайшей к определяемому положению. Нормализованная ссылочная пара для явной вершины – это сама вершина и пустая строка.

Продвижение от начальной точки к конечной осуществляется с использованием суффиксных ссылок. Принципы использования суффиксных ссылок аналогичны алгоритму Маккрейта, различается только терминология. Пусть произвольное положение r, заданное нормализованной ссылочной парой $(s,(k,i))$, является окончанием j-го суффикса, чтобы перейти к окончанию $(j+1)$-го

суффикса нужно пройти по суффиксной ссылке вершины *s* и спуститься по дереву читая подстроку t[*k*..*i*]. На рисунке 2.35 показан пример такого перемещения при добавлении очередного символа ‘x’. Для поддержания инварианта добавлена вершина SuperRoot (обозначена символом ⊥), обладающая свойствами описанными выше.

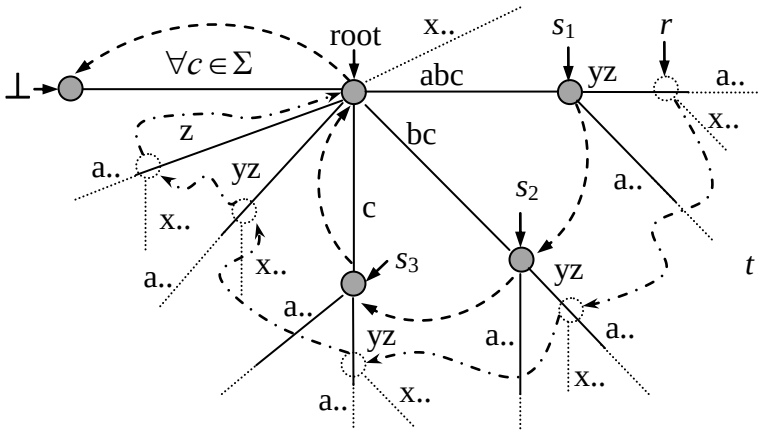

Рисунок 2.35. Пример построения суффиксного дерева.

Псевдокод алгоритма Укконена полностью приведен в оригинальной работе [12]. В алгоритме выделяются три основные процедуры: update, test–and–split и canonize. Первая, используя две другие, выполняет все ответвления при добавлении очередного символа, она вызывается в главном цикле. Процедура test–and–split проверяет условие достижения конечной точки и, при необходимости, создает новую внутреннюю вершину. Результаты ее работы возвращаются в update, где после этого либо создается новый лист, либо работа update завершается. Процедура canonize следит за тем, чтобы ссылочная пара оставалась нормализованной. Пример того, как переход по суффиксной ссылке может приводить к ненормализованной паре, можно увидеть на рисунке 2.36 в основе которого уже рассмотренный пример (рис. 2.29). Переход от положения z_1, заданного парой (u_1, ‘pqrst’), к z_2 описывается как ($f(u_1)$, ‘pqrst’), однако нормализованной парой этого положения будет (u_2, ‘rst’). В примере построения, показанном на рисунке 2.35, также видна необходимость нормализации. Цепочку положений, полученных переходами по суффиксным ссылкам можно описать так:

(s_1, ‘yz’), (s_2, ‘yz’), (s_3, ‘yz’), (root, ‘yz’), (⊥, ‘yz’) или (root, ‘z’) после нормализации, (⊥, ‘z’) или (root, ‘’) после нормализации.

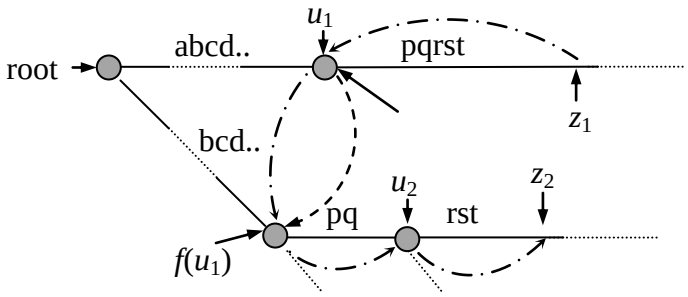

Рисунок 2.36. Использование суффиксных ссылок.

При переходе к следующему шагу алгоритма требуется определить начальную точку. Очевидно, что начальная точка – это положение соответствующее окончанию самого длинного суффикса, который заканчивается не в листе (на ребре или во внутренней вершине). Но так может закончиться только суффикс, соответствующий подстроке, добавленной в дерево как минимум дважды. Согласно своему определению, конечная точка соответствует самому длинному суффиксу, который уже продлялся очередным символом. Таким образом, если продвинуться на символ $t[i]$ от конечной точки шага i, можно получить начальную точку шага i+1.

В заключении следует отметить, что суффиксные ссылки создаются как и в алгоритме Маккрейта – с опозданием. Если при продлении суффикса j была создана новая явная вершина s, то $f(s)$ станет определена при продлении суффикса j+1. Очевидно, что после продлений суффиксные ссылки сохраняются.

Все действия, которые выполняет процедура update, для создания новой внутренней вершины и нового листа, за исключением нормализации (действий внутри процедур canonize), выполняются за O(1). Поскольку общее количество вершин O(|t|), суммарное количество таких действий, включая вызовы canonize, оценивается как O(|t|). Действия canonize схожи с процедурой rescan в алгоритме Маккрейта, линейную оценку их общего количества (суммарно по всем i) можно доказать аналогично.

Рассуждения, приведенные в настоящем пособии, являются попыткой облегчить и упростить описание рассмотренных алгоритмов, но не могут в полной мере заменить формальные доказательства, предложенные авторами.

2.4. Суффиксные массивы

Суффиксный массив – это массив целых чисел, определяющий лексикографический порядок суффиксов некоторой строки, он используется для решения тех же задач, что и суффиксное дерево. Использовать такую структуру данных было предложено Джином Майерсом и Уди Манбером в 1989 году [13]. Несомненным преимуществом массива являются скромные требования к памяти. Асимптотика использования памяти не изменится, но константы уменьшатся на порядок – для каждого суффикса требуется хранить одно единственное целое число, в отличие нескольких сложных объектов в случае дерева. Цена, которую приходится платить за экономию памяти – это увеличение времени решения задач, так например поиск образца p в тексте t с использованием заранее построенного суффиксного массива займет время $O(|p|\log|t|)$. Примеры суффиксных массивов изображены на рисунке 2.37.

t = a c a c g 0, 1, 2, 3, 4 pos = {0, 2, 1, 3, 4}	0: acacg 2: acg 1: cacg 3: cg 4: g	*t* = a a b c a 0, 1, 2, 3, 4 pos = {4, 0, 1, 2, 3}	4: a 0: aabca 1: abca 2: bca 3: ca

Рисунок 2.37. Суффиксные массивы pos для строк *t*.

Одним из способов построения суффиксного массива является обход суффиксного дерева в глубину. Если путь из вершины выбирается в соответствии алфавитным порядком первых символов ребер, то последовательность номеров листьев в порядке их посещения дает суффиксный массив. В этом случае время построения массива можно считать линейным. Обоснованность применения этого способа – вопрос спорный, поскольку массивы рассматриваются как альтернатива деревьям.

2.4.1. Построение без использования деревьев

Ниже описан алгоритм выполняющий построение суффиксного массива строки t за O($n$log(n)), где n=|t|. На k-ой фазе алгоритма k=0,1,..., $\lceil \log(n) \rceil$ ($\lceil\ \rceil$ – округление к большему целому) сортируются циклические подстроки длины 2^k, то есть строки t[i..(i+2^k)], i=0,1,...,n-1. После сортировки для каждой такой подстроки определяется ***номер класса эквивалентности***. Класс эквивалентности – это целое число от 0 до n-1. Равные циклические подстроки получают одинаковый номер класса эквивалентности, меньшая подстрока получает меньший. На рисунке 2.38 показаны подстроки, строки ‘aaba’ на каждой фазе алгоритма. Различные вхождения символа ‘a’ изображены по-разному. В таблице 2.2 для рассмотренного примера приведены массивы классов эквивалентности C и массивы pos, содержащие индексы строки t (позиции) на которых начинаются циклические подстроки, упорядоченные по алфавиту. Массив pos в последней фазе совпадает с суффиксным массивом. Чтобы порядок суффиксов в общем случае совпадал с порядком циклических сдвигов, в конце строки t нужно добавить символ заведомо меньший любого из ее символов. Массив.

Таблица 2.2. Содержимое массивов pos и C для различных фаз алгоритма.

k	i:	0	1	2	3	i:	0	1	2	3
0	pos[i]:	3	1	0	2	C[i]:	0	0	1	0
1	pos[i]:	0	3	1	2	C[i]:	0	1	2	0
2	pos[i]:	3	0	1	2	C[i]:	1	2	3	0

t = **a** a b a

k = 0: **a**, a, b, a
k = 1: **a**a, ab, ba, a**a**
k = 2: **a**aba, aba**a**, ba**a**a, a**a**ab

Рисунок 2.38. Циклические подстроки строки ‘aaba’.

На нулевой фазе с помощью сортировки подсчётом сортируются отдельные символы строки. Затем, сравнивая символы по порядку, строится массив C. Время выполнения нулевой фазы O(n). Фрагмент алгоритма, соответствующий нулевой фазе приведен на рисунке 2.39. Полная блок схема алгоритма включает три рисунка: 2.39, 2.42 и 2.43.

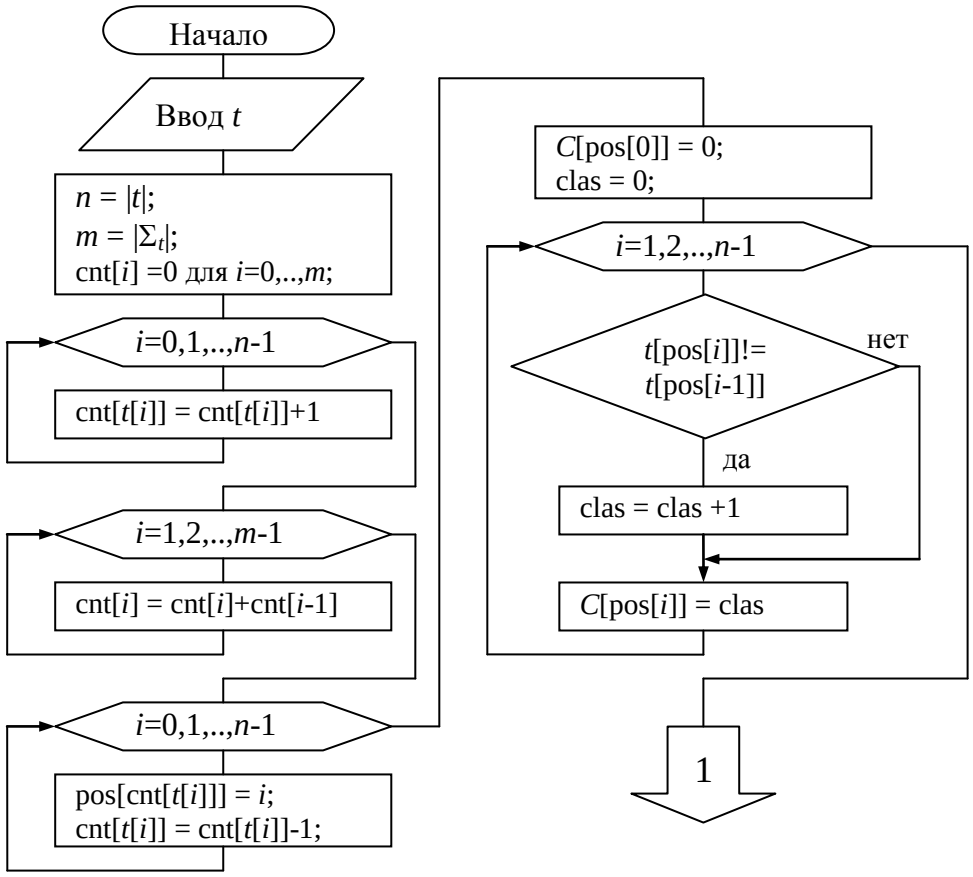

Рисунок 2.39. Начало алгоритма: определение констант, нулевая фаза.

Как показано на рисунке 2.40 подстрока длины 2^k состоит из двух подстрок длины 2^{k-1}, которые можно сравнивать между собой за O(1), используя информацию с предыдущей фазы – номера классов эквивалентности $C[i]$, $C[i+2^{k-1}]$.

$$|t_i \ldots t_{i+2^k-1}| = 2^k$$

$$\ldots \underbrace{t_i \ldots t_{i+2^{k-1}-1}\, t_{i+2^{k-1}} \ldots t_{i+2^k-1}} \ldots$$

$|t_i \ldots t_{i+2^{k-1}-1}| = 2^{k-1}$ Класс = $C[i]$

$|t_{i+2^{k-1}} \ldots t_{i+2^k-1}| = 2^{k-1}$ Класс = $C[i+2^{k-1}]$

Рисунок 2.40. Подстроки смежных фаз алгоритма.

Сортировка по вторым элементам уже содержится в массиве pos предыдущей фазы. Вспомогательный массив pn будет содержать соответствующий порядок циклических сдвигов текущей фазы: $pn[i]=pos[i]-2^{k-1}$, если получено отрицательное значение к нему прибавляется n. Пример иллюстрирующий соотношение массивов изображен на рисунке 2.41. Для

упорядочения по первым элементам используется устойчивая сортировка подсчетом. Фрагмент алгоритма показан на рисунке 2.42.

t = **a** a b a

k = 0: **a**, a, b, a pos = {3,1,0,2}

k = 1: **a**a, ab, ba, a**a** pn = { 2, 0, 3, 1 } pos = {0,3,1,2}

k = 2: **a**aba, aba**a**, ba**a**a, a**a**ab pn = { 2 ,1 ,3 ,0 }

Рисунок 2.41. Соотношение массивов pos и pn.

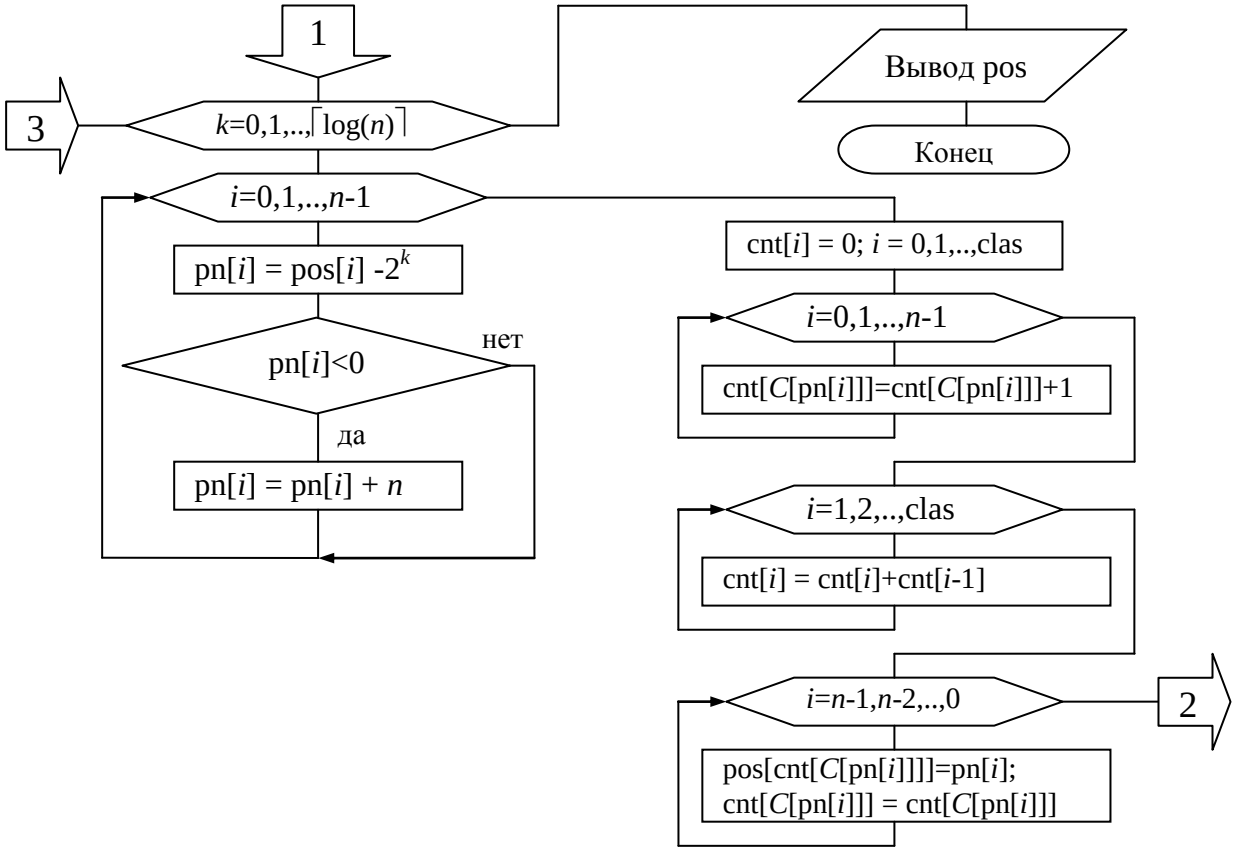

Рисунок 2.42. Основной цикл: фрагмент соответствующий сортировке.

Для вычисления классов эквивалентности на текущей фазе необходимо заглянуть в массив C предыдущей фазы и сравнить две пары значений, соответствующих половинам текущих подстрок. Фрагмент алгоритма, соответствующей части тела основного цикла, выполняющий перестроение массива C показан на рисунке 2.43.

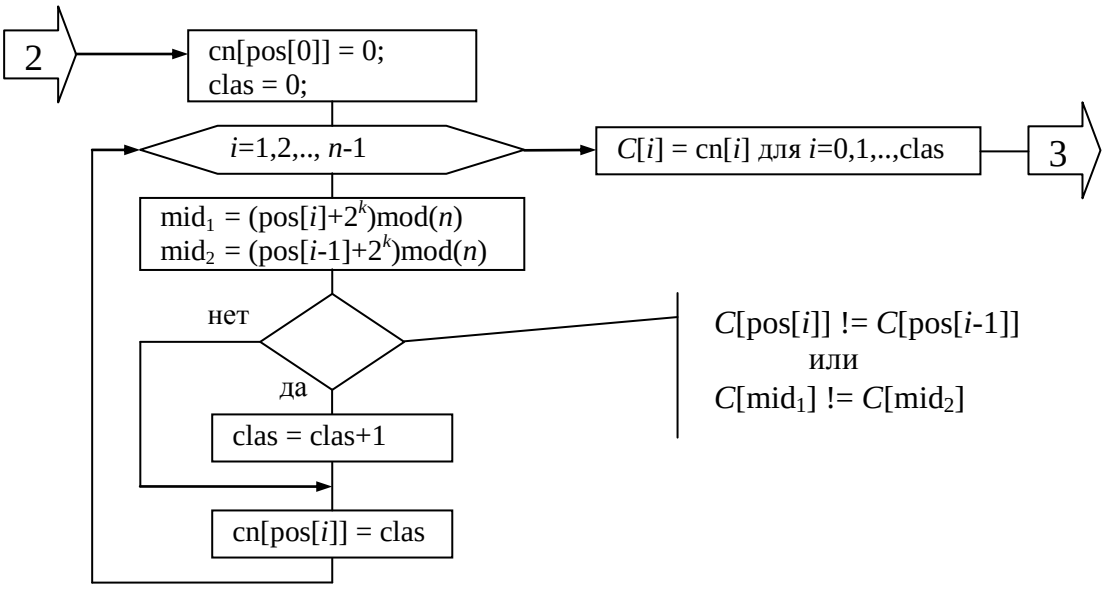

Рисунок 2.43. Основной цикл: вычисление классов эквивалентности.

2.4.2. Поиск подстроки

Для поиска образца p длины m в тексте t длины n с использованием суффиксного массива pos необходимо при помощи двоичного поиска найти наименьший индекс i, такой что t[pos[i−1]..n-1] не начинается с p и наибольший i', что t[pos[i'+1]..n-1] не начинается с p. Пример и алгоритм двоичного поиска в массиве чисел показаны на рисунках 2.44, 2.45 (⌊⌋ - округление к меньшему целому). При использовании двоичного поиска для строк время выполнения каждого сравнения зависит от числа сравниваемых символов – m сравнений в худшем случае. Пессимистичная оценка времени работы O($m$log(n)), достигается при наличии в t большого количества длинных префиксов p.

```
L=0, R=Длина(A);
Пока L > R Цикл
        M=⌊(L+R)/2⌋;
        Если A[M] < P Тогда
                L=M+1;
        Иначе
                R=M;
        Конец если
Конец цикла
```

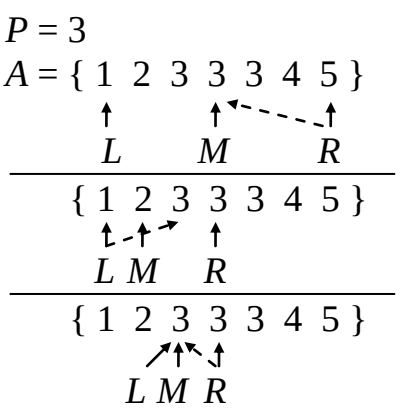

Рисунок 2.44. Поиск первого вхождения числа P в массиве A.

```
L=Индекс первого вхождения P в A;
R=Длина(A);
Пока L > R Цикл
        M=⌊(L+R)/2⌋;
        Если A[M] <= P Тогда
                L=M;
        Иначе
                R=M-1;
        Конец если
Конец цикла
```

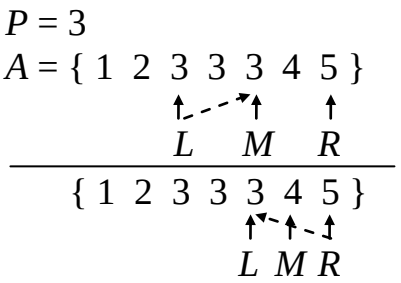

Рисунок 2.45. Поиск последнего вхождения числа *P* в массиве *A*.

Поиск можно ускорить, если запоминать количество совпадающих символов. Пусть L и R – левая и правая границы поиска соответственно. Если длину префикса $t[pos[L]..n-1]$ совпадающего с префиксом p обозначить l, а соответствующую длину для правой границы – r, то при очередном сравнении в позиции $M=\lfloor (R+L)/2 \rfloor$ можно начинать обрабатывать символы не с первой позиции, а с $\min(l, r)+1$. Такое улучшение позволяет достигать оценки $O(m+\log(n))$ на «хороших» входных данных, однако в худшем случае остается $O(m\log(n))$.

Для гарантированного уменьшения количества дополнительных сравнений на шаге бинарного поиска можно использовать дополнительную информацию – длину наибольшего общего префикса (longest common prefix) LCP. Пусть $LCP(i,j)$ – длина наибольшего общего префикса суффиксов $pos[i]$ и $pos[j]$. Тогда если l не равно r, а, например, больше, то соотношение $LCP(L,M)$ и l поможет сделать дальнейшие выводы. Если $LCP(L,M)$ меньше l, то p совпадает с суффиксом $pos(L)$ на большее количество символов, чем с суффиксом $pos(M)$, следовательно, нужно продолжить поиск в левой части и изменить значение r на $LCP(L,M)$. Если $LCP(L,M)$ больше l, то суффиксы $pos(L)$ и $pos(M)$ совпадают до $(l+1)$-го символа, при этом суффикс $pos(L)$ в позиции $(l+1)$ уже отличается от p, следовательно, нужно продолжить поиск в правой части, а l оставить без изменений. Т только в случае равенства $LCP(L,M)$ и l необходимо сравнивать соответствующие символы суффикса $pos(M)$ и строки p начиная с $(l+1)$-го, до тех пор, пока не встретятся

различные. Ситуация, в которой r больше l, обрабатывается аналогично. Таким образом, время работы поиска будет складываться из $\log(n)$ шагов, на которых в сумме производится m сравнений символов, но на каждом шаге необходимо вычисление LCP(i,j).

Один из способов определения LCP(i,j), требующий $\log(n)$ времени и $n\log(n)$ памяти состоит в следующем. Массивы классов эквивалентности C, которые использовались при построении, содержат информацию о равенстве подстрок строки t. На k-й фазе алгоритма это подстроки длины 2^k. Каждый суффикс можно представить как конкатенацию таких подстрок (начиная с наибольшей), как показано на рисунке 2. 46.

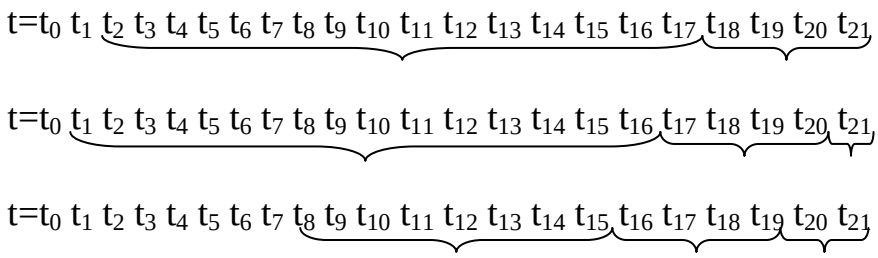

Рисунок 2.46. Суффиксы как конкатенация подстрок различных фаз алгоритма.

Пусть Ck[k] – массив классов эквивалентности k-й фазы. Чтобы определить LCP(i,j) нужно перебирать степени двойки (от большей к меньшей) и проверять: совпадают ли подстроки соответствующей длины. Если совпадают, то к ответу следует прибавить эту степень двойки, а наибольший общий префикс продолжить искать справа от одинаковой части. Алгоритм вычисления LCP(i,j) приведен на рисунке 2.47.

```
Результат =0;
Для k =⌊log(n)⌋, log(n)/2, log(n)/4,..,0 Цикл
        Если Ck[k][i] = Ck[k][j] Тогда
                Результат = Результат + 2^k;
                i= i +2^k;
                j= j +2^k;
        Конец если
Конец цикла
```

Рисунок 2.47. Алгоритм вычисления LCP(i,j).

2.5. Наибольшая общая подпоследовательность

Согласно определению ***подпоследовательностью*** – последовательности $\{x_n\}$ называется последовательность $\left\{x_{n_k}\right\}$, составленная из некоторых членов $\{x_n\}$, взятых в порядке возрастания номеров n_k. Иначе говоря, чтобы получить подпоследовательность можно выбрать некоторые элементы исходной последовательности, не меняя их взаимного расположения, или (в случае, если исходная последовательность конечна) удалить из нее некоторое количество элементов. Например, для строки ‘abcdef’ строки ‘acf’, ‘bd’, ‘bcef’ и т.п. будут подпоследовательностями.

Наибольшей общей подпоследовательностью (longest common subsequence, LCS) строк *x* и *y* будет строка наибольшей длины, одновременно являющаяся подпоследовательностью *x* и *y*. Длина LCS может представлять интерес как самостоятельная характеристика. Она тесно связана с редакционным расстоянием – минимальным количеством операций (вставки, удаления или замены символа), необходимых для преобразования одной строки в другую. Редакционное расстояние впервые упомянуто в работе советского математика Владимира Иосифовича Левенштейна [14]. Очевидно, что длина LCS является величиной обратной расстоянию Левенштейна, чем больше совпадающая подпоследовательность, тем меньше операций необходимо для преобразования.

Нахождение LCS может быть полезным не только при сравнении текстовых строк, но и в задачах биоинформатики при анализе ДНК.

2.5.1. Рекурсивный поиск длины LCS

Простейший алгоритм поиска длины LCS представлен на рисунке 2.48. Каждый рекурсивный вызов сравнивает только первые символы строк, на входе в рекурсию одна из строк или обе лишаются первого символа, это происходит до тех пор, пока одна из строк (или обе) не станут пустыми. Важно отметить, что в случае несовпадения символов, порождается два вызова. Пример работы алгоритма приведен на рисунке 2.49, решение задачи представлено в виде дерева. Очевидным недостатком алгоритма является то, что одинаковые подзадачи

(выделены овалом на рис. 2.49) решаются несколько раз. На рисунке 2.50 представлено дерево решений для другого примера, в котором у входных строк нет совпадающих символов. Это двоичное дерево высотой (|*x*|+|*y*|-1), что позволяет оценить временную сложность алгоритма как $O(2^{(|x|+|y|)})$.

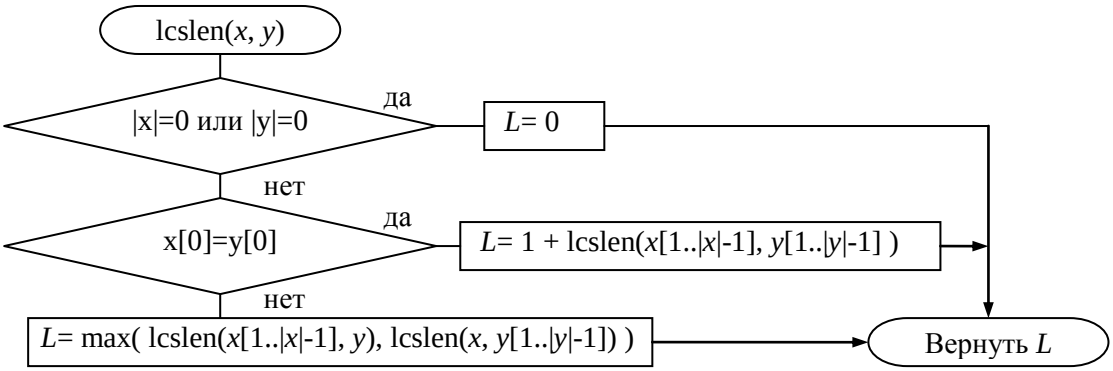

Рисунок 2.48. Алгоритм рекурсивного поиска длины LCS.

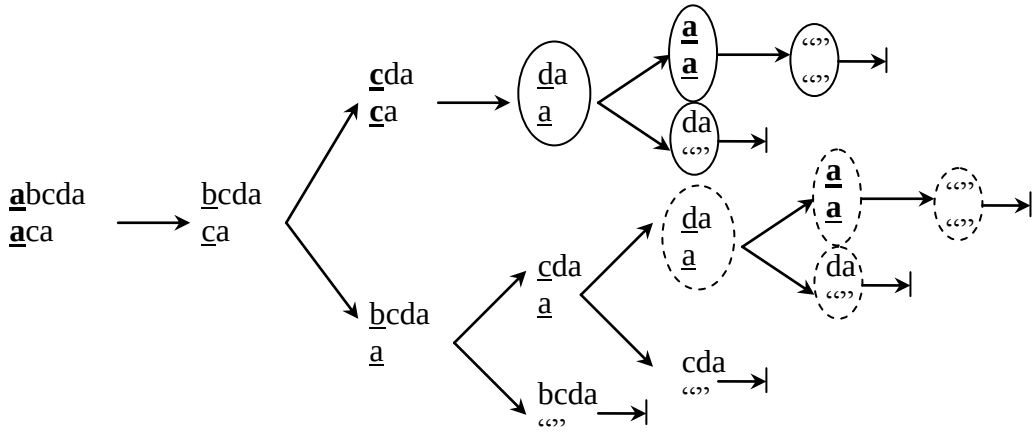

Рисунок 2.49.Поиск LCS для строк *x*=abcda и *y*=aca.

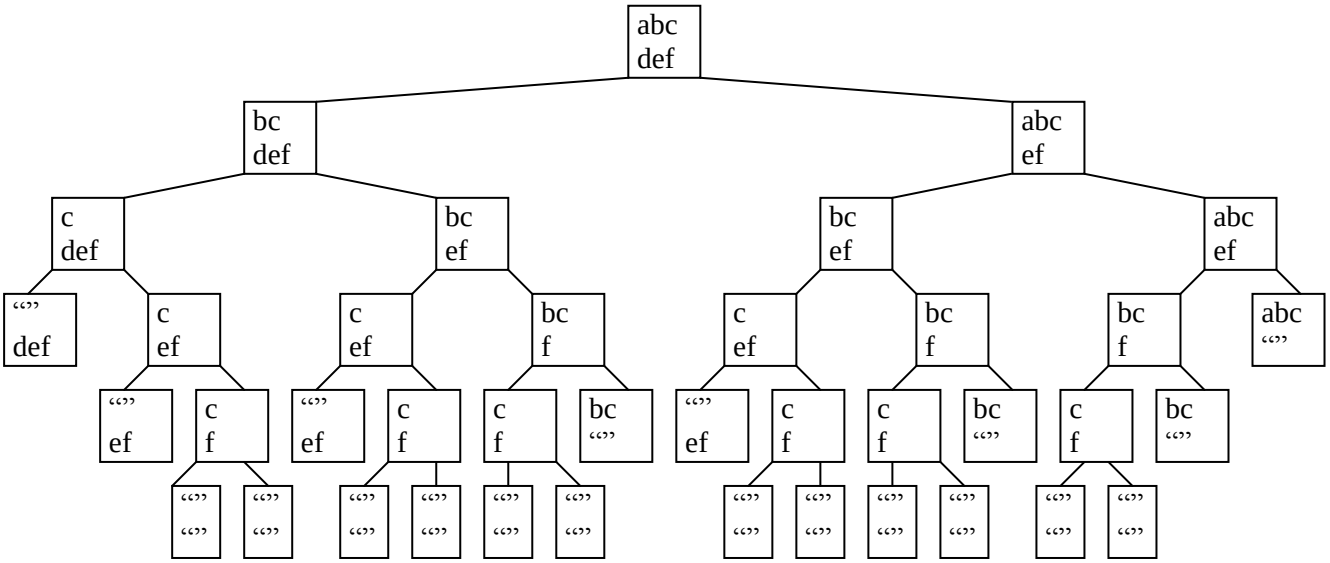

Рисунок 2.50.Поиск LCS для строк *x*=abc и *y*=def.

Значение, которое возвращает каждый рекурсивный вызов можно интерпретировать как длину LCS некоторых суффиксов *x* и *y*. Количество различных сочетаний суффиксов исходных строк – это |*x*||*y*|. Если в процессе работы запоминать решения подзадач и обращаться к ним вместо повторных расчетов, то временная сложность значительно уменьшится и составит O(|*x*||*y*|), но это потребует такого же объема памяти. Псевдокод рекурсивного алгоритма с запоминанием приведен на рисунке 2.51. Невыполнение условия в строке (6) означает, что для данной подзадачи решение уже найдено, и вся «ветка» рекурсии, которая могла бы быть запущена, отсекается.

```
(1)     m=|x|; n=|y|;
(2)     выделить память для двумерного массива L;
(3)     L[i][j]=-1; для i=0,..,m; j=0,..,n;
(4)     Вернуть lcs_len(0,0);
(5)     Процедура lcs_len(i,j)
(6)         Если L[i][j] < 0 Тогда
(7)             Если i=m или j=n Тогда
(8)                 L[i][j]=0;
(9)             Иначе
(10)                Если x[i]=y[j] Тогда
(11)                    L[i][j] = 1 + lcs_len(i+1,j+1);
(12)                Иначе
(13)                    L[i][j] = max(lcs_len(i+1,j), lcs_len(i,j+1));
(14)                Конец если
(15)            Конец если
(15)        Конец если
(16)        Вернуть L[i][j];
(17)    Конец процедуры
```

Рисунок 2.51. Рекурсивный алгоритм с запоминанием.

2.5.2. Динамическое программирование

Итерационный алгоритм поиска LCS, в котором решения всех подзадач вычисляются последовательно от меньших к большим, является частным случаем широкого подхода к решению оптимизационных задач известного как «динамическое программирование». На рисунке 2.52 представлен алгоритм заполнения массива *L*, каждый элемент которого равен длине LCS различных суффиксов исходных строк. Элемент с индексами *i, j* соответствует LCS строк *x*[*i*..|*x*|-1] и *y*[*j*..|*y*|-1]. Пример заполненного массива показан на рисунке 2.53. Окончательный результат – длина LCS строк *x* и *y* находится в ячейке с

индексами 0, 0. Если требуется определить не только длину, но и саму подпоследовательность, то нужно проделать путь из этой ячейки вправо и вниз, выбирая направление с большим значением. Позиции, в которых символы совпадают, составляют искомую последовательность. Алгоритм вывода LCS и пример его работы показаны на рисунке 2.54.

```
(1)   m=|x|; n=|y|;
(2)   выделить память для двумерного массива L;
(3)   Для i = m, (m-1),..,0 Цикл
(4)       Для j = n, (n-1),..,0 Цикл
(5)           Если i=m или j=n Тогда
(6)               L[i][j]=0;
(7)           Иначе
(8)               Если x[i]=y[j] Тогда
(9)                   L[i][j] = 1 + L[i+1][j+1];
(10)              Иначе
(11)                  L[i][j] = max(L[i+1][j], L[i][j+1]);
(12)              Конец если
(13)          Конец если
(14)      Конец цикла
(15)  Конец цикла
```

Рисунок 2.52. Итерационный алгоритм поиска LCS.

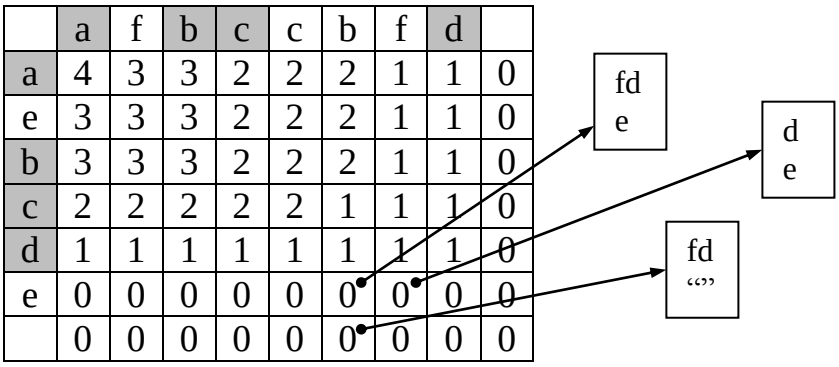

Рисунок 2.53. Заполненный массив *L* для строк *x*=‘aebcde’ и *y*=‘afbccbfd’.

	a	f	b	c	c	b	f	d	
a	4	3	3	2	2	2	1	1	0
e	3	3	3	2	2	2	1	1	0
b	3	3	3	2	2	2	1	1	0
c	2	2	2	2	2	1	1	1	0
d	1	1	1	1	1	1	1	1	0
e	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0

```
(1)   s=“”; i=0; j=0; m=|x|; n=|y|;
(2)   Пока i <m и j <n Цикл
(3)       Если x[i]=y[j] Тогда
(4)           Добавить символ x[i] к s;
(5)           j=j+1; i=i+1;
(6)       Иначе
(7)           Если L[i+1][j] ≥L[i][j+1] Тогда
(8)               i=i+1;
(9)           Иначе
(10)              j=j+1;
(11)          Конец если
(12)      Конец если
(13)  Конец цикла
```

Рисунок 2.54. Вывод LCS для строк *x*=‘aebcde’ и *y*=‘afbccbfd’.

Если требуется определить только длину LCS, то можно уменьшить требования к памяти – запоминать только предыдущую строку матрицы *L*. Первым примером приложения динамического программирования к поиску последовательностей аминокислот является алгоритм Нидлмана-Вунша [15]. Принцип динамического программирования лежит в основе алгоритма выравнивания двух последовательностей Вагнера-Фишера [16].

2.5.3. Алгоритм Хиршберга

В алгоритме, разработанном Хиршбергом [17], используется особый способ разбиения задачи на две меньшие, так чтобы сумма их решений совпадала с решением исходной. Примеры подходящего и неподходящего разбиения показаны на рисунке 2.55. Первую строку можно разделить пополам, а для выбора варианта разбиения второй необходимо провести расчеты.

|LCS(”aca”, ”abcda”)| = 3;
|LCS(”ac”, ”ab”)| + |LCS(”a”, ”cda”)| = 1+1=2;
|LCS(”ac”, ”abc”)| + |LCS(”a”, ” da”)| = 2+1=3;

Рисунок 2.55. Примеры разбиения задачи поиска LCS.

Если для второй строки и правой половины первой строки заполнить матрицу решений подзадач в соответствии с предыдущим алгоритмом, то верхняя строка (заполненная последней) будет содержать длины LCS правой половины первой строки и суффиксов второй. Для левой половины матрица заполняется в обратном порядке, и будет содержать длины LCS левой половины первой строки и префиксов второй. Для выбора подходящего разбиения нужно определить такие префикс и суффикс, для которых суммарная длина LCS со своими половинами первой строки будет максимальной. В примере, показанном на рисунке 2.56, подходят две пары, для которых сумма длин LCS равна четырем.

Если продолжить рекурсивно разбивать задачи, то одна из строк станет пустой или сократится до одного символа. Если одна из строк пустая, то результат равен нулю. Если строка сократилась до одного символа, то проверяется наличие этого символа во второй строке и возвращается 1 или 0, в первом случае к LCS добавляется этот символ.

С учетом отсутствия необходимости хранить матрицы полностью требования к памяти оцениваются как O(|*x*|+|*y*|), временная сложность по-прежнему O(|*x*||*y*|).

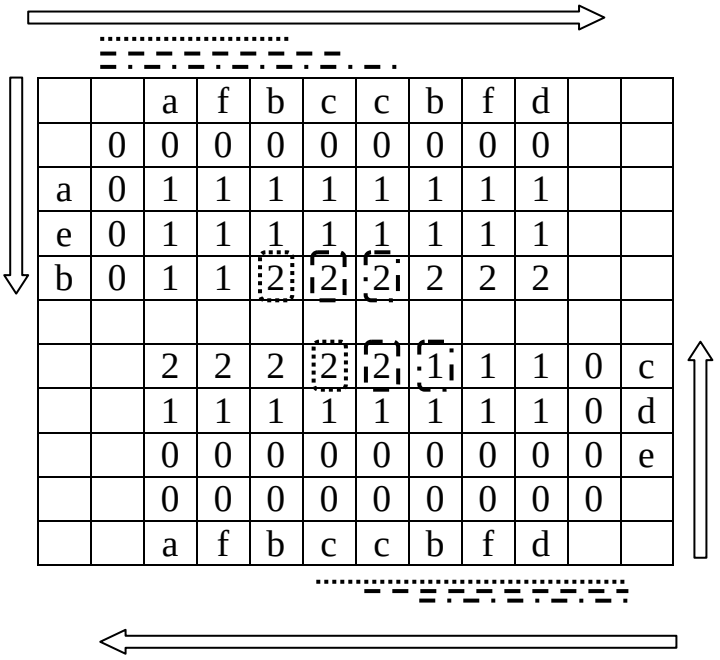

Рисунок 2.56. Критерий выбора разбиения строки *y*=‘afbccbfd’ при *x*=‘aebcde’.

2.5.4. Алгоритм Ханта-Шиманского

В алгоритме, который в 1977 году опубликовали Джеймс Хант и Томас Шиманский [18], используется так называемый «массив пороговых значений». При описании этого алгоритма значительно удобнее считать, что **индекс первого символа строки равен 1**. Элемент массива пороговых значений с индексами *i*, *s* содержит минимальный индекс *j*, при котором строки *x*[1..*i*] и *y*[1..*j*] имеют LCS длины *s*. Пусть этот массив обозначен *k*, тогда можно сказать, что $k_{i,s}$ это длина самого короткого префикса *y*, имеющего с префиксом *x* длины *i* LCS длины *s*. Если для некоторых *i* и *s* таких *j* не существует, то считается что $k_{i,s}$ не определено. Если *x* = ‘aebcde’ и *y* = ‘facbfacfb’, то $k_{5,1} = 2$, $k_{5,2} = 4$, $k_{5,3} = 6$, а $k_{5,4}$ и $k_{5,5}$ не определены. Неопределенные $k_{i,s}$ заполняются значением (|*y*|+1).

Прежде чем перейти к алгоритму необходимо рассмотреть некоторые свойства массива пороговых значений. Поскольку *j* это минимальный индекс, символ *y*[$k_{i,s}$] будет последним членом LCS, следовательно, элементы $\boldsymbol{k_{i,1},..,k_{i,p}}$ **(*p*<(|*y*|+1)) образуют возрастающую последовательность**. Пусть префиксу строки *x* длиной *i* для достижения длины LCS, равной *s*, был сопоставлен префикс

y длиной $k_{i,s}$. Если расширить префикс x на один символ, то для достижения такой же длины LCS предыдущего префикса y будет достаточно, а с учетом того, что берется префикс y наименьшей длины, то он может уменьшиться, **то есть $k_{i+1,s} \le k_{i,s}$.** По определению $x[1..i+1]$ и $y[1..k_{i+1,s}]$ имеют LCS длины s, удаление последнего символа уменьшит LCS не более чем на единицу. Таким образом, $x[1..i]$ и $y[1..k_{i+1,s}-1]$ имеют LCS длины s-1. С другой стороны минимальная длина префикса y с которым $x[1..i]$ имеет LCS длины s-1 – это $k_{i,s-1}$, **поэтому $k_{i,s-1} \le k_{i+1,s}-1$ или $k_{i,s-1} < k_{i+1,s}$.**

С учетом перечисленных свойств можно записать правило вычисления значения $k_{i+1,s}$: оно равно наименьшему j для которого выполняются условия: $x[i+1]=y[j]$ и $k_{i,s-1} < j \le k_{i,s}$, если таких j не существует, то $k_{i+1,s}=k_{i,s}$. Таким образом, $k_{i+1,s}$ можно определить, зная предыдущие значения $k_{i,s}$, s=0,1,..,min(|x|,|y|). На рисунке 2.57 слева показан пример заполненного массива k, справа показаны пары совпадающих символов, звёздочкой отмечены пары, вошедшие в LCS, прочерком – которые не вошли. Для перебора совпадающих символов в алгоритме предлагается использовать массив указателей на списки matchlist, i-й элемент которого указывает на начало списка позиций j, таких что $x[i]=y[j]$, идущих в убывающем порядке. Создать такую структуру данных можно с использованием сортировки за O($n$log(n)), где n – длина большей строки.

i	s	0	1	2	3	4	5	6	7	8
1	d	0	10	10	10	10	10	10	10	10
2	d	0	10	10	10	10	10	10	10	10
3	a	0	2	10	10	10	10	10	10	10
4	b	0	2	4	10	10	10	10	10	10
5	a	0	2	4	6	10	10	10	10	10
6	d	0	2	4	6	10	10	10	10	10
7	c	0	2	3	6	7	10	10	10	10
8	b	0	2	3	6	7	9	10	10	10

	j	1	2	3	4	5	6	7	8	9
i		f	a	c	b	f	a	c	f	b
1	d									
2	d									
3	a		*				-			
4	b				*					
5	a		-				*			
6	d									
7	c			-				*		
8	b				-					*

matchlist[1] = ()
matchlist[2] = ()
matchlist[3] = (6,2)
matchlist[4] = (9,4)
matchlist[5] = matchlist[3]
matchlist[6] = ()
matchlist[7] = (7,3)
matchlist[8] = matchlist[4]

Рисунок 2.57 Пример заполнения массива пороговых значений для строк x='ddabadcb', y='facbfacfb'.

Длина LCS определяется как максимальное s, для которого $k_{|x|,s}$ определено. Укрупненный алгоритм представлен на рисунке 2.58. Для определения самой LCS используется еще одна вспомогательная структура данных – массив указателей на элементы связного списка – link. Пример формирования такого списка приведен на рисунке 2.59. В процессе построения был создан элемент содержащий индекс j=3, указатель на него заменил предыдущее значение link[2]. Однако символ соответствующий элементу не попал в LCS. Этот символ мог бы оказаться частью LCS, если бы строка x имела другое продолжение.

Поиск в строке (8) может быть выполнен за O(log(|y|)), учитывая, что массив k упорядочен. Итоговая оценка времени работы алгоритма O((*r*+*n*)log(*n*)), где r – количество совпадений символов. Объем дополнительной памяти можно оценить как O(*r*+*n*).

```
(1)   Сформировать списки matchlist;
(2)   k[0] = 0; link[0]= «пустой указатель»;
(3)   Для s = 1,2,..,|x| Цикл
(4)       k[s] = |y|+1;
(5)   Конец цикла
(6)   Для i = 1,2,..,|x| Цикл
(7)       Для каждого j из matchlist[i] Цикл
(8)           Найти s удовлетворяющее ограничениям k[s-1]<j≤k[s];
(9)           Если s найдено Тогда
(10)              k[s] = j;
(11)              Создать новый элемент списка, содержащий индексы i,j
                                          и указывающий на link[s-1];
(12)              Записать в link[s] указатель на новый элемент;
(13)          Конец если
(14)      Конец цикла
(15)  Конец цикла
(16)  Вывод максимального s, для которого k[s] != |y|+1;
(17)  Вывод содержимого списка по указателю link[s];
```

Рисунок 2.58 Алгоритм поиска LCS.

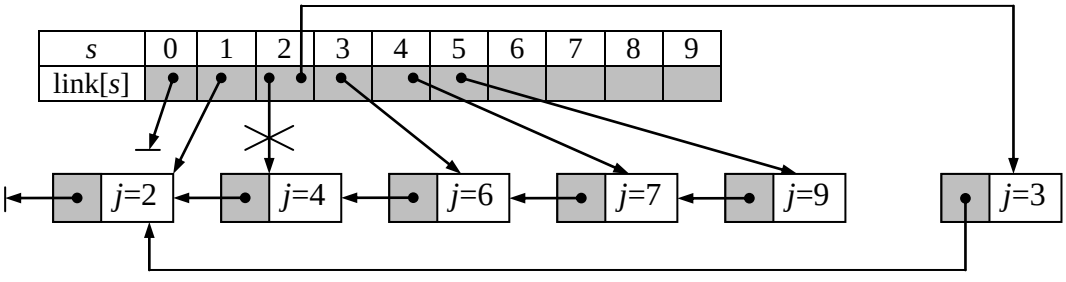

Рисунок 2.59 Содержимое списка link.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вирт Н. Алгоритмы+структуры данных = программы: пер. с англ. – М.: Мир, 1985. – 406 с.
2. Shell D.L. A High-Speed Sorting Procedure // Communications of the ACM. 1959. № 2 (7): 30–32.
3. Incerpi J., Sedgewick R. Improved Upper Bounds for Shellsort // Journal of computer and system sciences № 31. 1985. –Pp. 210–224.
4. Hoare C.A.R. Quicksort // The Computer Journal № 5 (1). 1962. –Pp. 10–16.
5. Ахо А.В., Хопкрофт Д.Э., Ульман Д.Д. Структуры данных и алгоритмы: учеб. пособие; пер. с англ. – М.: Издательский дом «Вильямс», 2000. – 384 с.
6. Williams J.W.J. Algorithm 232 – Heapsort // Communications of the ACM. 1964. № 7 (6). –Pp. 347–348.
7. Seward H.H. Information sorting in the application of electronic digital computers to business operations // Master's thesis, Report R-232, Massachusetts Institute of Technology. 1954.
8. Rabin M.O., Karp R.M. Efficient randomized pattern-matching algorithms // IBM Journal of Research and Development. 1987. № 31 (2). –Pp. 249–260.
9. Aho A.V., Corasick M.J. Efficient string matching: An aid to bibliographic search // Communications of the ACM. 1975. № 18 (6). –Pp. 333–340.
10. Weiner P. Linear pattern matching algorithm // Proc. 14th IEEE Symposium on Swithching and Automata Theory. 1973. № 1-11.
11. McCreight E.M. A Space-Economical Suffix Tree Construction Algorithm // Journal of the ACM. 1976. № 23 (2). –Pp. 262–272.
12. Ukkonen E. On-line construction of suffix trees // Algorithmica. 1995. № 14 (3). – Pp. 249–260.
13. Manber U., Myers G. Suffix arrays: a new method for on-line string searches // First Annual ACM-SIAM Symposium on Discrete Algorithms. 1990. –Pp. 319–327.
14. Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады Академий Наук СССР. 1965. № 163.4. – С 845-848.
15. Needleman S.B., Wunsch C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins // Journal of Molecular Biology. 1970. № 48 (3). – Pp. 443–453.

16. Wagner R.A., Fischer M.J. The String-to-String Correction Problem // Journal of the ACM. 1974. № 21 (1).– Pp. 168–173.
17. Hirschberg D.S. A linear space algorithm for computing maximal common subsequences // Communications of the ACM. 1975. № 18 (6). – Pp. 341–343.
18. Hunt J.W., Szymanski T.G. A fast algorithm for computing longest common subsequences // Communications of the ACM. 1977. № 20 (5). – Pp. 350–353.