Жадные алгоритмы

Дискретный анализ 2012/13

Андрей Калинин, Татьяна Романова

16 марта 2013 г.

Литература

- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.. Алгоритмы: построение и анализ, 2-е издание, М.:Вильямс, 2005, стр. 442-478, глава 16, «Жадные алгоритмы».
- Дэн Гасфилд, «Строки деревья и последовательности в алгоритмах: Информатика и вычислительная биология», 2003. Глава 12, «Улучшение процедур выстраивания», стр. 351–359.

Жадные алгоритмы

- Задача о выборе процессов
- Элементы жадной стратегии
- Коды Хаффмана
- Наибольшая общая подпоследовательность

Раздел

Жадные алгоритмы

- Задача о выборе процессов
- Элементы жадной стратегии
- Коды Хаффмана
- Наибольшая общая подпоследовательность

Идея

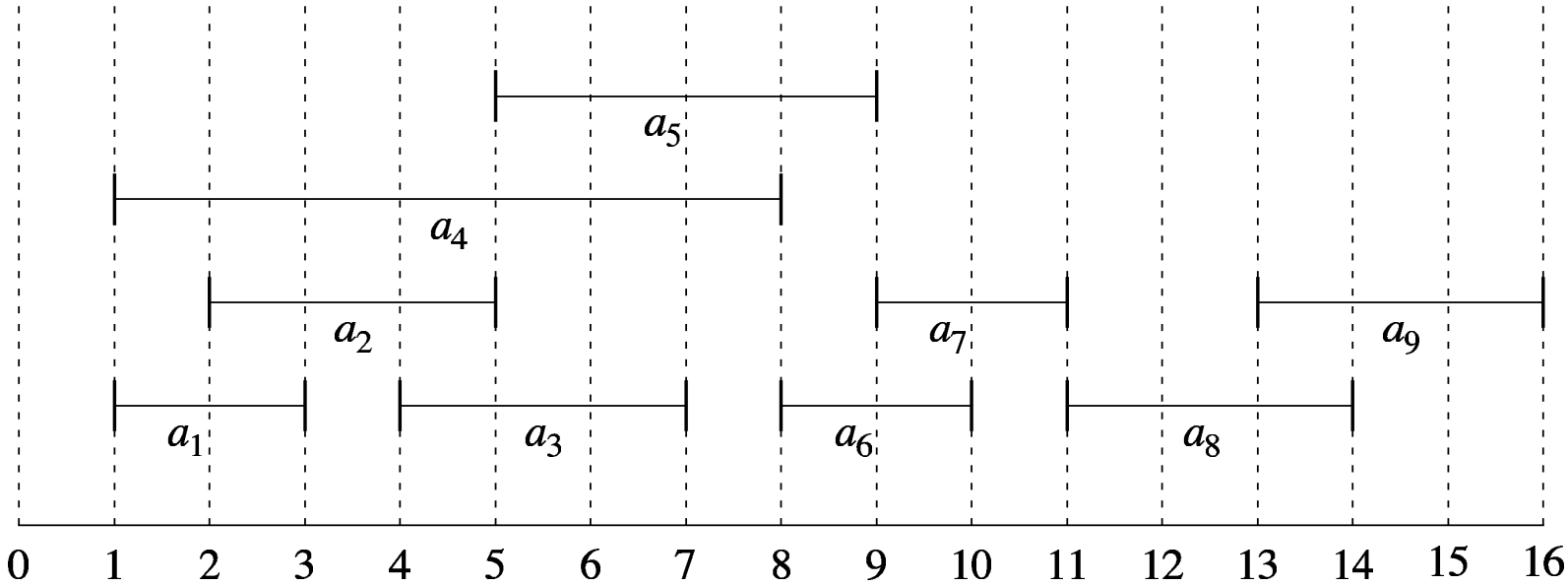
- Похожи на динамическое программирование, используются для решения задач оптимизации.
- Когда приходит необходимость сделать выбор, то делается выбор такого решения, которое кажется лучшим *сейчас*. Надеемся, что локально-оптимальный выбор приведёт к глобально-оптимальному решению.
- Жадные алгоритмы далеко не всегда дают оптимальное решение. Однако, в тех случаях, когда они всё-таки приводят к нему, они работают за очень короткое время.
- Рассмотрим задачи, решаемые «жадным» способом и определим область применимости жадных алгоритмов.

Раздел

Постановка задачи

- Есть n процессов, требующих эксклюзивного использования общего ресурса (например лекции в аудитории).
- $S = \{a_1, \ldots, a_n\}$.
- a_i требует ресурс на период $[s_i, f_i)$.
- Нужно выбрать наибольшее возможное множество неперекрывающихся по использованию ресурса процессов.
- В задачах о расписаниях могут быть другие цели: занять компнату на самое длительное время, максимизировать доход от аренды и т.п.

Пример



i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

Множества:

$\{a_1, a_3, a_6, a_8\}$;

$\{a_2, a_5, a_7, a_9\}$.

Оптимальная подструктура

- Количество процессов, начинающихся после окончания a_i и заканчивающихся перед началом a_j:

$$S_{ij} = \{a_k \in S \mid f_i \leq s_k < f_k \leq s_j\}$$

- Процессы в S_{ij} совместны со всеми другими процессами, которые заканчиваются до f_i и начинаются после s_j.
- Добавим фиктивные процессы:

$$\begin{aligned} a_0 &= [-\infty, 0) \\ a_{n+1} &= [\infty, \infty + 1) \end{aligned}$$

- $S = S_{0,n+1}$.

Оптимальная подструктура

- Предположим, что все процессы отсортированы по возрастанию времени окончания:

$$f_0 \leq f_1 \leq f_2 \cdots \leq f_n \leq f_{n+1}.$$

- Тогда для $i \geq j$ $S_{ij} = \emptyset$. Предположим обратное:
  1. Если существует $a_k \in S_{ij}$, то

$$f_i \leq s_k < f_k \leq s_j < f_j \quad \Rightarrow \quad f_i < f_j$$

  2. Но из $i \geq j$ должно следовать $f_i \geq f_j$, т.е. противоречие.

  То есть, нужно рассматривать только S_{ij} для котрых $0 \leq i < j \leq n + 1$

Оптимальная подструктура

Допустим, оптимальное решение A_{ij} для S_{ij} включает в себя a_k. Тогда есть две подзадачи:

- Найти оптимальное решение A_{ik} для S_{ik}.
- Найти оптимальное решение A_{kj} для S_{kj}.

Необходимость оптимального решения доказывается от противного. Тем самым:

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

Рекурсивное решение

- $c[i,j] = |A_{ij}|$.
- Если $a_k \in A_{ij}$, то

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

- Тем самым:

$$c[i,j] = \begin{cases} 0 & S_{ij} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & S_{ij} \neq \emptyset \end{cases}$$

Рекурсивное решение

- $c[i, j] = |A_{ij}|$.
- Если $a_k \in A_{ij}$, то

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

- Тем самым:

$$c[i, j] = \begin{cases} 0 & S_{ij} = \emptyset \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & S_{ij} \neq \emptyset \end{cases}$$

Далее просто решить задачу методом динамического программирования, но мы пойдём иным путём.

Обоснование жадного выбора

Теорема

Расмотрим произвольную непустую задачу S_{ij} и пусть a_m — процесс, который заканчивается раньше других:

$$f_m = \min\{f_k \mid a_k \in S_{ij}\}.$$

В этом случае справедливы такие утверждения:

1. *Процесс a_m используется в некоторм подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij}.*
2. *Подзадача S_{im} пустая, поэтому в результате выбора процесса a_m непустой остаётся только подзадача S_{mj}.*

Доказательство.

2. Допустим, $a_k \in S_{im}$. Тогда $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$. Т.е., $a_k \in S_{ij}$ и заканчивается раньше f_m, что противоречит выбору a_m. Следовательно не существует $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$.
1. Покажем, что a_m входит в одно из решений задачи S_{ij}:
   - Организуем процессы в A_{ij} в порядке возрастания времени окончания.
   - a_k — первый процесс в A_{ij}.
   - Если $a_k = a_m$, то то утверждение доказано.
   - Иначе, сделаем $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (заменим a_k на a_m).
   - Процессы в A'_{ij} не перекрывается, т.к. $f_m \leq f_k$.
   - $|A_{ij}| = |A'_{ij}|$, т.е. A'_{ij} — подмножество максимального размера, включающее в себя процесс a_m.

Жадный выбор

Чтобы решить задачу S_{ij}:

- Найдём $a_m \in S_{ij}$ с наименьшим временем окончания — жадный выбор.
- Решим задачу S_{mj}. Последовательно выбираем процессы, проходя по ним в порядке возрастания времени окончания.

Рекурсивный алгоритм

$\textsc{Rec-Activity-Selector}(s, f, i, n)$

1 $m \leftarrow i + 1$
2 **while** $m \leq n$ *and* $s_m < f_i$
3 $\quad m \leftarrow m + 1$
4 **if** $m \leq n$
5 $\quad$ **return** $\{a_m\} \cup \textsc{Rec-Activity-Selector}(s, f, m, n)$
6 **else return** $\emptyset$

Итеративный алгоритм

GREEDY-ACTIVITY-SELECTOR(s, f, i, n)

```
1  A ← {a_1}
2  i ← 1
3  for m ← 2 to n
4        if s_m ≥ f_i
5              A ← A ∪ {a_m}
6              i ← m
7  return A
```

Раздел

Что было сделано?

1. Определена оптимальная подструктура задачи.
2. Разработано рекурсивное решение.
3. Доказано, что на любом этапе рекурсии один из оптимальных выбров является жадным.
4. Показано, что все возникающие в результате жадного выбора подзадачи, кроме одной, — пустые.
5. Разработан рекурсивный алгоритм, реализующий жадную стратегию.
6. Рекурсивный алгоритм преобразован в итеративный.

Процесс разработки жадных алгоритмов

1. Привести задачу оптимизации к виду, когда после сделанного выбора остаётся решить только одну подзадачу.
2. Доказать, что всегда существует такое оптимиальное решение исходной задачи, которое можно получить путём жадного выбора, так что такой выбор всегда допустим.
3. Показать, что после жадного выбор остаётся подзадача, обладающая тем свойством, что объединение оптимального решения подзадачи со сделанным жадным выброром приводит к оптимальному решению исходной задачи.

Свойство жадного выбора

Глобальное оптимальное решение можно получить, делая локальный оптимальный (жадный) выбор. Для динамического программмирования:

- На каждом шаге делается выбор.
- Выбор зависит от известных оптимальных решений для подзадач, т.е. сначала нужно решить подзадачи.
- Порядок решения «снизу-вверх».

Для жадного выбора:

- На каждом шаге делается выбор.
- Выбор делается перед решением подзадач.
- Порядок решения «сверху-вниз».

Оптимальная подструктура

Для применимости жадных алгоритм требуется, как и в динамическом программировании, наличие оптимальной подструктуры в задаче.

Сравнение жадных алгоритмов и динамического программирования

- Дискретная задача о рюкзаке: Вор во время ограбления магазина обнаружил n предметов. Предмет под номером i имеет стоимость v_i и вес w_i, где v_i и w_i — целые числа. Нужно унести вещи, суммарная стоимость которых была бы как можно большей, однако грузоподъёмность рюкзака ограничивается W килограммами. Какие предметы следует взять с собой?
- Непрерывная задача о рюкзаке та же, но теперь тот или иной товар вор может брать с собой частично, а не делать каждый раз бинарный выбор.

Задачи о рюзкаке: оптимальная подструктура

В дискретной задаче есть оптимальная подструктура:

- Допустим, есть оптимальное решение задачи для рюкзака весом W.
- Если из рюкзака вынуть j-ый предмет, то остальные предметы должны быть наиболее ценными, вес которых не превышает $W - w_j$ и которые можно составить из $n - 1$ исходных предметов.

Для непрерывной задачи рассуждения аналогичны.

Жадное решение задач о рюкзаке

- Отсортируем вещи по «удельной ценности», v_i/w_i.
- Будем последовательно выбирать вещи с максимальной удельной ценностью и помещать их в рюкзак.

Жадное решение задач о рюкзаке

- Отсортируем вещи по «удельной ценности», v_i/w_i.
- Будем последовательно выбирать вещи с максимальной удельной ценностью и помещать их в рюкзак.

Такое решение работает для непрерывной задачи о рюкзаке и не работает для дискретной. Например, $W = 50$:

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

Жадное решение: вещи 1 и 2, сумма 160, вес 30.
Оптимальное решение: вещи 2 и 3, сумма 220, вес 50.

Раздел

Пример

Допустим, есть файл, состоящий из 100 000 символов, который нужно сжать:

	a	b	c	d	e	f
Частота (в тысячах)	45	13	12	16	9	5
Фикс. длина	000	001	010	011	100	101
Перемен. длина	0	101	100	111	1101	1100

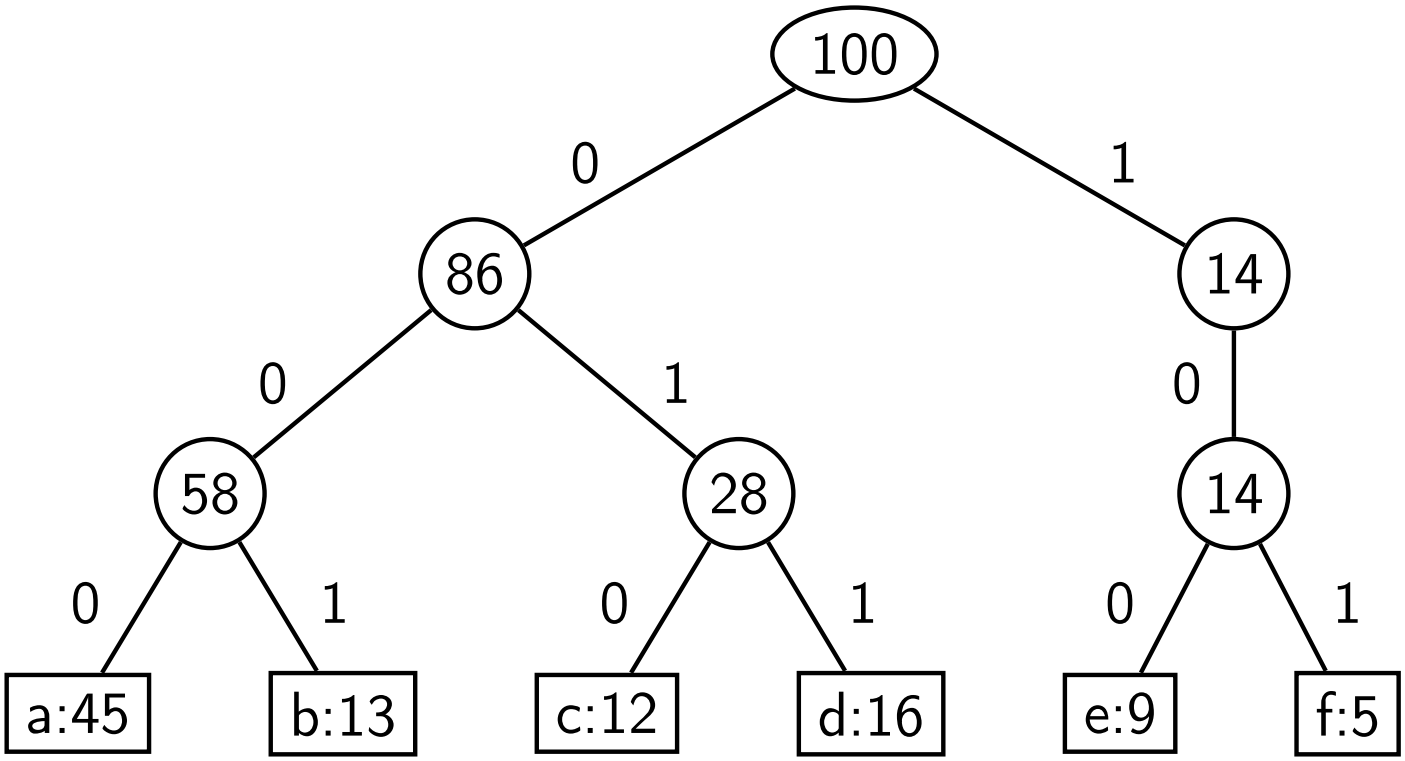
При использовании кода фиксированной длины понадобится 300 000 битов; с помощью кода переменной длины потребуется

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000$$

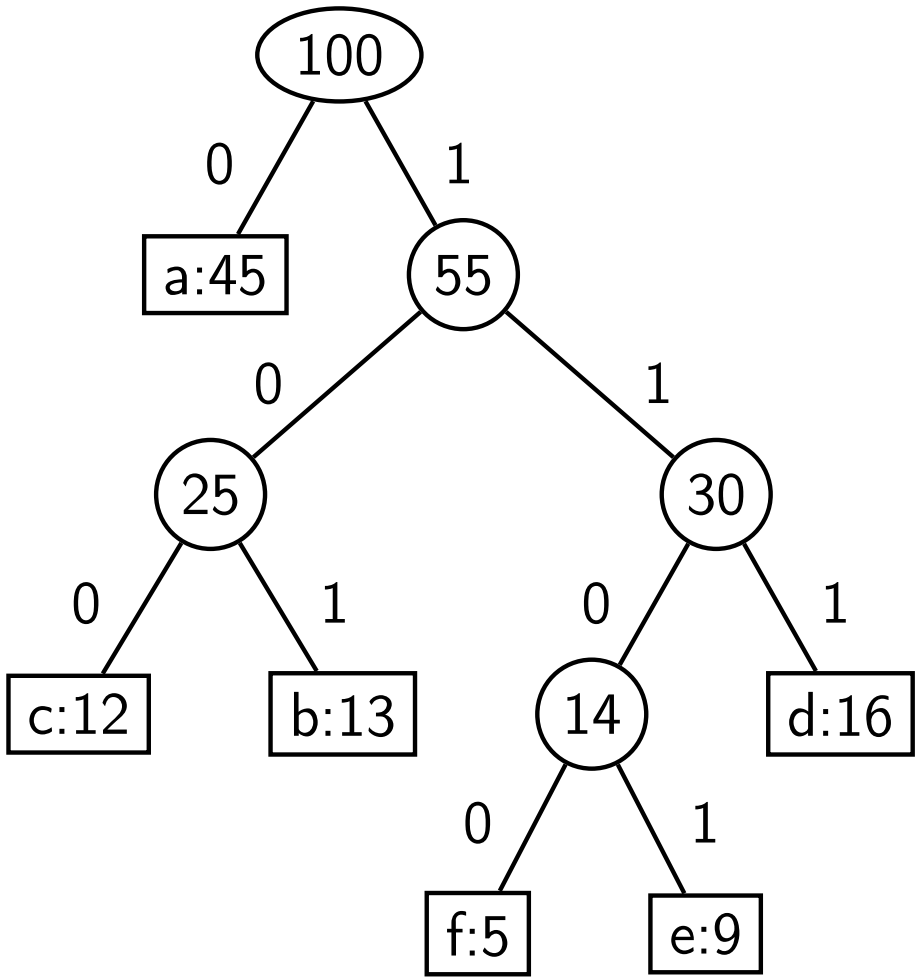
Префиксные коды

- Префиксный код — код, в котром ни одно кодовое слово не является префиксом другого кодового слова.
- Тогда кодирование текста — конкатенация всех кодов символов.
- Префиксные коды упрощают декодирование.
- Код можно представить в виде дерева.

	a	b	c	d	e	f
Частота (в тысячах)	45	13	12	16	9	5
Фикс. длина	000	001	010	011	100	101
Перемен. длина	0	101	100	111	1101	1100



	a	b	c	d	e	f
Частота (в тысячах)	45	13	12	16	9	5
Фикс. длина	000	001	010	011	100	101
Перемен. длина	0	101	100	111	1101	1100



Дерево оптимального кода

- Оптимальный код текста всегда может быть представлен полным бинарным деревом, в котором у каждого узла (кроме листьев) имеется по два дочерних узла.
- Т.е., если C — алфавит, то дерево, представляющее оптимальный префиксный код, содержит ровно $|C|$ листьев и $|C| - 1$ внутренних узлов.
- Если есть дерево T некоторого префиксного кода, то обозначив через $f(c)$ частоту символа $c \in C$, а через $d_T(c)$ — глубину листа c в дереве T, то для кодировки текста потребуется:

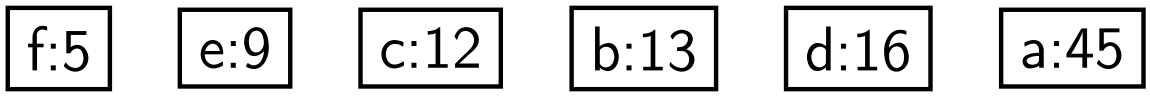
$$B(T) = \sum_{c \in C} f(c) d_T(c).$$

$B(T)$ — стоимость дерева T.

Построение дерева Хаффмана

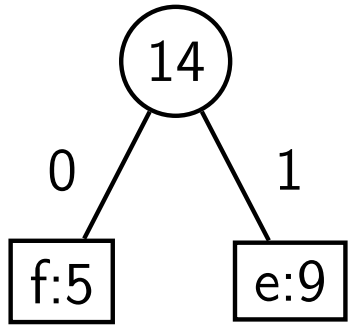
Q — очередь с приоритетами, пирамида.

1 $n \leftarrow |C|$, $Q \leftarrow C$
2 **for** $i \leftarrow 1$ **to** $n - 1$
3 Выделить память для узла z
4 $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)
5 $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)
6 $f[z] \leftarrow f[x] + f[y]$
7 INSERT(Q, z)
8 **return** EXTRACT-MIN(Q)

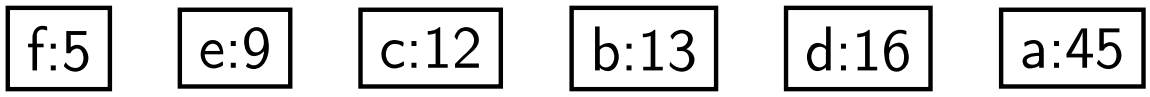

f:5
e:9
c:12
b:13
d:16
a:45

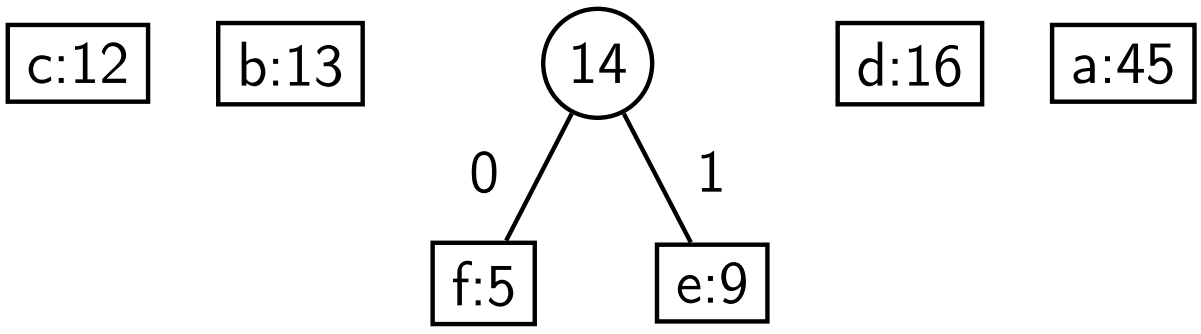
f:5 e:9 c:12 b:13 d:16 a:45

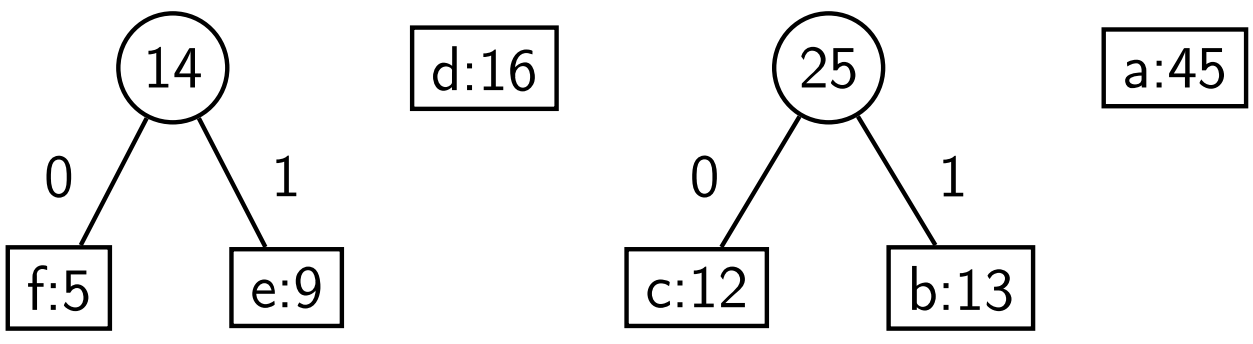
c:12 b:13

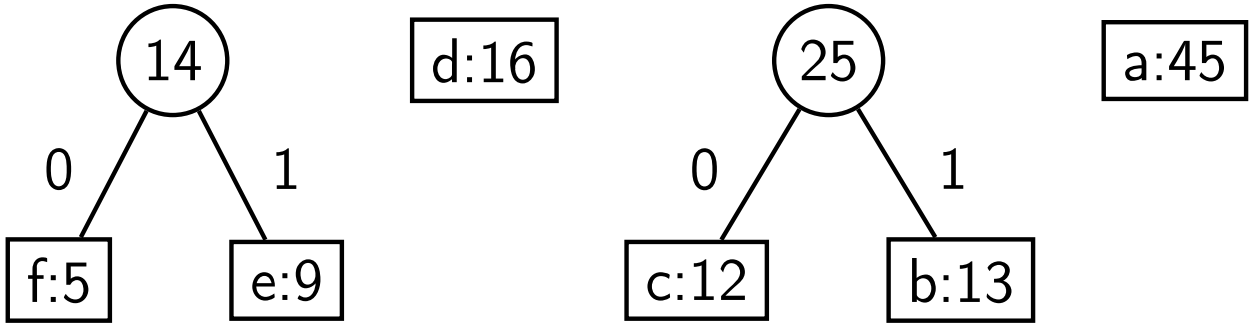


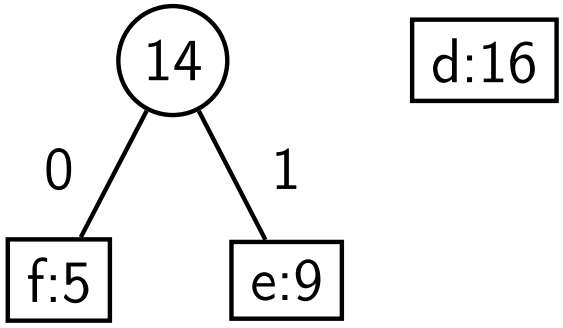
d:16 a:45


f:5
e:9
c:12
b:13
d:16
a:45

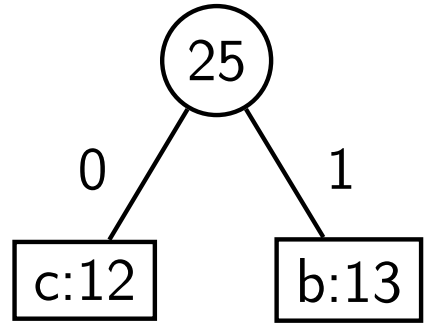

c:12
b:13
14
0
1
f:5
e:9
d:16
a:45


14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45

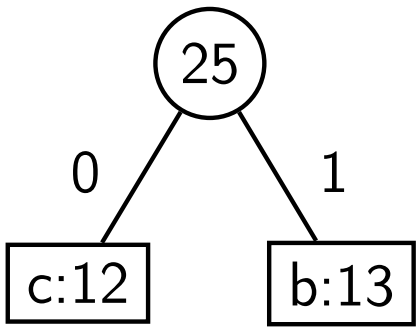

14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45

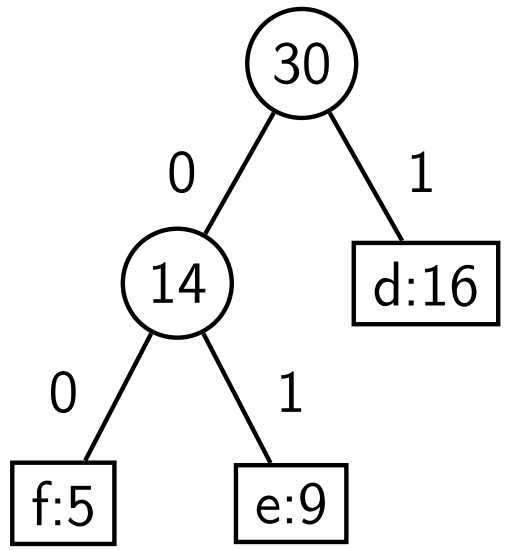


d:16

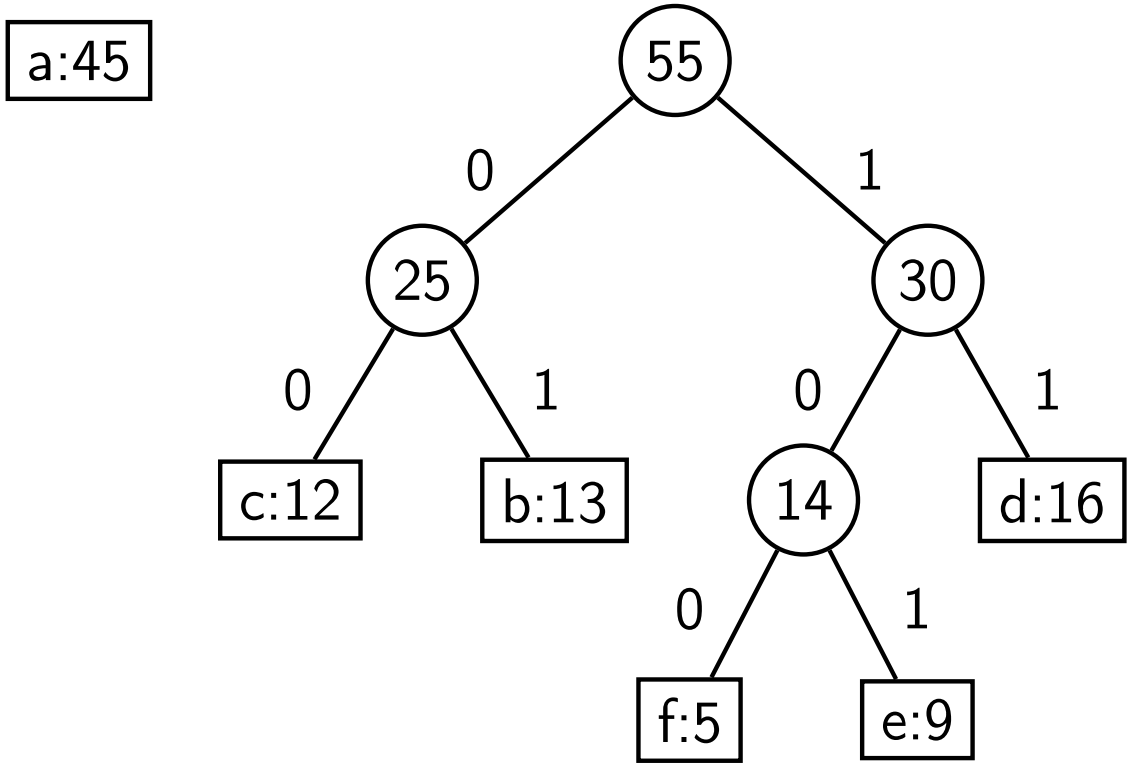


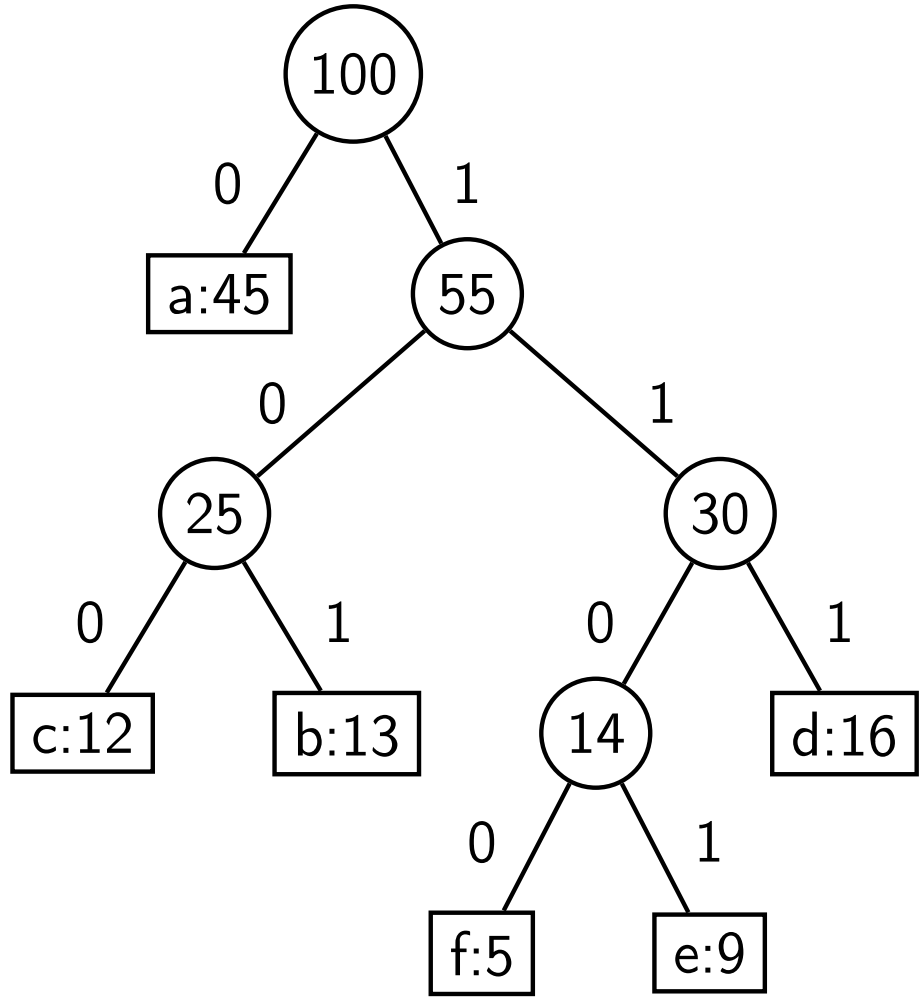
a:45





a:45


a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9


100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Обоснование жадного выбора

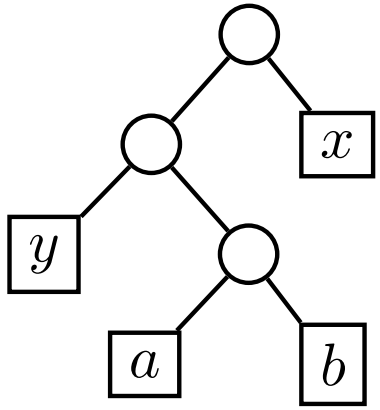
Теорема

Пусть C — алфавит, каждый символ $c \in C$ которого встречается с частотой $f[c]$. Пусть x и y — два символа алфавита C с самыми низкими частотами. Тогда для алфавита C существует оптимальный префиксный код, кодовые слова символов x и y в котором имеют одинаковую длину и отличаются лишь последним битом.

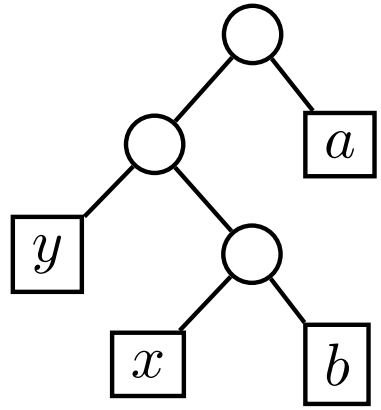
Доказательство.

- T — дерево оптимального префиксного кода.
- a и b — символы, находящиеся на максимальной глубине с общим родительским узлом. Предположим, что $f[a] \leq f[b]$ и $f[x] \leq f[y]$. Тогда и $f[x] \leq f[a]$ и $f[y] \leq f[b]$.

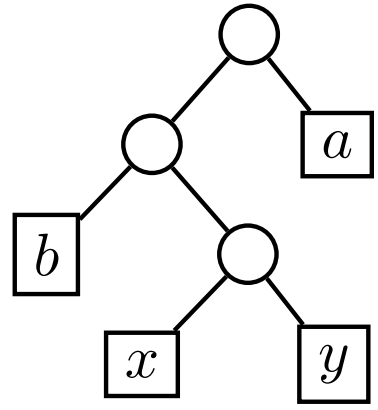
T:



T':



T'':



Доказательство.

- T' — перестановка a и x, а T'' – перестановка b и y. Тогда:

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) = \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T'}(x) - f[a] d_{T'}(a) = \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) = \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

- Аналогично, $B(T') - B(T'') \geq 0$ и $B(T'') \leq B(T)$. Но должно так же $B(T) \leq B(T'')$, следовательно $B(T'') = B(T)$.

Оптимальная подструктура

Теорема

- $x, y \in C$ — *символы с минимальными частотами.*
- $C' = C - \{x, y\} \cup \{z\}$, $f[z] = f[x] + f[y]$.
- T' — *дерево оптимального префиксного кода* C'.
- *Тогда дерево* T, *полученное из* T' *путём замены листа* z *внутренним узлом с сыновьями* x *и* y, *представляет оптимальный префиксный код для* C.

Доказательство.

- $c \in C - \{x, y\}: \ d_T(c) = d_{T'}(c)$, т.е. $f[c]d_T(c) = f(c)d_{T'}(c)$,
- $d_T(x) = d_T(y) = d_{T'}(z) + 1$, тогда:

$$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = \\ = f[z]d_{T'}(z) + (f[x] + f[y]).$$

- Следовательно, $B(T) = B(T') + f[z] + f[y]$ или $B(T') = B(T) - f[x] - f[y]$.
- Предположим, что T — не оптимальное. Тогда существует оптимальное дерево T'', такое что $B(T'') < B(T)$. x и y можно считать сыновьями одного узла и T''' получено из T'' заменой x и y листом z с частотой $f[z] = f[x] + f[y]$. Тогда:

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

что противоречит тому, что T' — оптимальное дерево для C'.

Раздел

Задача

- Даны две строки, длиной n и m, нужно найти наибольшую общую подпоследовательность.
- Методом динамического программирования можно разработать алгоритм $\Theta(nm)$.
- Жадные алгоритмы позволяют свести к $O(r \log n)$, где r в некоторых условиях достаточно мало, чтобы оценка была лучше $\Theta(nm)$.
- Основная идея — сведение к задаче о наибольшей возрастающей подпоследовательности.

Возрастающая подпоследовательность

Определение

Π — *список из* n *целых чисел. Возрастащая подпоследовательность* Π — *подпоследовательность* Π, *значения которой строго возрастают слева направо.*

Возрастающая подпоследовательность

Определение

Π — *список из* n *целых чисел. Возрастащая подпоследовательность* Π — *подпоследовательность* Π, *значения которой строго возрастают слева направо.*

Например, $\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$. Тогда возрастающие подпоследовательности:

- $\{3, 4, 6, 8, 10\}$
- $\{5, 9, 10\}$

Невозрастающая подпоследовательность

Определение

Невозрастающая подпоследовательность Π — *подпоследовательность* Π, *в которой числа не возрастают слева направо.*

Невозрастающая подпоследовательность

Определение

Невозрастающая подпоследовательность Π — *подпоследовательность* Π, *в которой числа не возрастают слева направо.*

Например, $\{8, 5, 5, 3, 1, 1\}$ — невозрастающая подпоследовательность для $\{4, 8, 3, 9, 5, 2, 5, 3, 10, 1, 9, 1, 6\}$.

Покрытие

Определение

Покрытие Π *— набор невозрастающих подпоследовательностей* Π*, которые содержат все элементы* Π*.*

Покрытие

Определение

Покрытие Π *— набор невозрастающих подпоследовательностей* Π*, которые содержат все элементы* Π*.*

Например, для $\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$ одно из покрытий будет состоять из:

- $\{5, 3, 2, 1\}$
- $\{4\}$
- $\{9, 6\}$
- $\{8, 7\}$
- $\{10\}$

Размер покрытия

Определение

Размер покрытия — число невозрастающих подпоследовательностей в нём, а наименьшее покрытие — покрытие минимального размера.

Размер покрытия

Определение

Размер покрытия — число невозрастающих подпоследовательностей в нём, а наименьшее покрытие — покрытие минимального размера.

Теорема

Если I — возрастающая подпоследовательность Π, длина которой равна размеру некоторого покрытия C последовательности Π, то I является наибольшей возрастающей подпоследовательностью Π, а C — наименьшим покрытием Π.

Доказательство.

1. I не может содержать более одного числа из невозрастающей подпоследовательности Π, таким образом $|I| \leq |C|$ для любых I и C.
2. Допустим, $|I| = |C|$. Тогда I — наибольшая возрастающая подпоследовательность, т.к. $\nexists |I| > |C|$. И, наоборот, не существует $|C'| < |C|$, т.к. тогда $|I| \neq |C|$.

$\square$

Наивный алгоритм построения покрытия

```
1  for i ← 1 to |Π|
2      j ← 1
3      while Π[i] > Last(C[j]) and j ≤ |C|
4          j ← j + 1
5      if j > |C|
6          Alloc-New-List(C)
7      Push(C[j], Π[i])
```

Пример выполнения алгоритма

$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$

C :

Пример выполнения алгоритма

$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$

5

C :

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C : 5
3

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C : 5 4 3

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C : 5 4 9
3

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9
3		6

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9
3		6
2		

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9
3		6
2		
1		

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9	8
3		6	
2			
1			

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9	8
3		6	7
2			
1			

Пример выполнения алгоритма

$$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$$

C :

5	4	9	8	10
3		6	7	
2				
1				

Пример выполнения алгоритма

$\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$

C :

5	4	9	8	10
3		6	7	
2				
1				

Назовём покрытие, полученное в результате работы алгоритма, «жадным».

Жадное покрытие Π можно построить за время $O(n^2)$.

Оптимальность работы алгоритма

Теорема

Существует возрастающая подпоследовательность I из Π, содержащая ровно по одному числу из каждой подпоследовательности жадного покрытия C. Следовательно, I является наибольшей возможной а C — наименьшим возможным.

Доказательство.

- Допустим, во время работы алгоритма x добавляется в $C[i]$.
- Следовательно, в этот момент времени сверху $C[i-1]$ находится $y < x$, которое располагается в Π раньше, чем x.
- Тем самым, $\{y, x\}$ образуют возрастающую подпоследовательность в Π.
- Аналогичное рассуждение можно применить и для y, найдя соответствующий z, дающий возрастающую подпоследовательность $\{z, y, x\}$.
- И т.д., до $C[1]$.
- Взяв в качестве исходного x любое число из $C[|C|]$, получим I.

Построение I

1 $i \leftarrow |C|$, $I \leftarrow \{\textsc{Last}(C[|C|])\}$.
2 **while** $i > 1$
3 $\quad j \leftarrow 1$
4 $\quad$ **while** $C[i-1][j] < x$
5 $\quad\quad j \leftarrow j + 1$
6 $\quad x \leftarrow C[i-1][j]$, $i \leftarrow i - 1$
7 $\quad I \leftarrow \{x\} \cup I$

Построение I

1 $i \leftarrow |C|$, $I \leftarrow \{\textsc{Last}(C[|C|])\}$.
2 **while** $i > 1$
3 $\quad j \leftarrow 1$
4 $\quad$ **while** $C[i-1][j] < x$
5 $\quad\quad j \leftarrow j + 1$
6 $\quad x \leftarrow C[i-1][j]$, $i \leftarrow i - 1$
7 $\quad I \leftarrow \{x\} \cup I$

Так как ни одно число не просматривается дважды, время работы — $O(n)$. Можно сократить время работы до $O(|C|)$, если запоминать при добавлении x в $C[i]$ размер $C[i-1]$ (т.е., какое число находится сверху $C[i-1]$ на этот момент).

Быстрое построение жадного покрытия

L — список, содержащий последние числа $C[i]$ в ходе выполнения алгоритма.

Теорема

В любой момент работы алгоритма значения в списке L упорядочены по возрастанию.

Доказательство.

- Допустим, утверждение выполняется до $k-1$-ой итерации. $\Pi[k] = x$, добавляется в $C[i]$.
- $w = \textsc{Last}(C[i-1])$, $y = \textsc{Last}(C[i])$, $z = \textsc{Last}[C+1]$ (если эти числа существуют)
- $w < x \leq y$ по правилам работы алгоритма.
- $y < z$ по индуктивному предположению.
- Следовательно, $x < z$, т.е. $w < x < z$, т.е. новый список L остаётся упорядоченным.

$\square$

Скорость построения покрытия

Можно заменить поиск нужной невозрастающей подпоследовательности $C[i]$ с линейного на бинарный. Отсюда, учитывая $n = |\Pi|$:

Теорема

Жадное покрытие можно построить за время $O(n \log n)$. Поэтому наибольшая возрастающая подпоследовательность и наименьшее покрытие Π могут быть получены за время $O(n \log n)$.

Связь с наибольшей общей подпоследовательностью

- Пусть заданы строки S_1 и S_2 (длиной m и n соответственно) над алфавитом $|\Sigma|$. Обозначим через $r(i)$ количество вхождений $S_1[i]$ в S_2.
- Обозначим за r:

$$r = \sum_{i=1}^{m} r(i).$$

- Например, для:
  - $S_1 = abacx$
  - $S_2 = baabca$

  имеем $r(1) = 3$, $r(2) = 2$, $r(3) = 3$, $r(4) = 1$, $r(5) = 0$. Т.е., $r = 9$.
- Решим задачу $LCS(S_1, S_2)$ за время $O(r \log n)$ $(n < m)$.
- Для больших алфавитов $r \lll nm$.

Сведение LCS к LIS

- Для всех x, встречающихся в S_1, создадим список позиций этого символа в S_2 и запишем его в убывающем порядке.
- $S_1 = abacx$, $S_2 = baabca$; для a: $\{6, 3, 2\}$, для b: $\{4, 1\}$.
- Построим список $\Pi(S_1, S_2)$ длиной r следующим способом: каждое вхождение символа в S_1 заменим списком позиций.
- Например, $\Pi(abacx, baabca) = \{6, 3, 2, 4, 1, 6, 3, 2, 5\}$.
- $I = \{3, 4, 5\}$, т.е. $LCS(S_1, S_2) = abc$, позиции символов в S_1 $\{1, 2, 4\}$ и в S_2 $\{3, 4, 5\}$.

Сведение LCS к LIS

Теорема

Каждая возрастающая подпоследовательность I *в* $\Pi(S_1, S_2)$ *определяет общую подпоследовательность* S_1 *и* S_2 *той же длины, и наоборот. Таким образом, наибольшая общая подпоследовательность* S_1 *и* S_2 *соответствует наибольшей возрастающей подпоследовательности в списке* $\Pi(S_1, S_2)$

Сведение LCS к LIS

Теорема

Каждая возрастающая подпоследовательность I в $\Pi(S_1, S_2)$ определяет общую подпоследовательность S_1 и S_2 той же длины, и наоборот. Таким образом, наибольшая общая подпоследовательность S_1 и S_2 соответствует наибольшей возрастающей подпоследовательности в списке $\Pi(S_1, S_2)$

Теорема

Задача о наибольшей общей подпоследовательности S_1 и S_2 может быть решена за время $O(r \log n)$, где $r = |\Pi(S_1, S_2)|$.

Эффективность алгоритма

- Если $\sigma = |\Sigma|$, то предполагая равномерное распределение символов можно считать что каждый символ встретится в короткой строке n/σ раз.
- Тем самым, $r_i = n/\sigma$ и $r = nm/\sigma$.
- В задачах с небольшим размером алфавита выигрыш спорен (хотя всё зависит от реальных частот символов).
- В задачах, где алфавит очень велик (или вообще не фиксирован и растёт с размером текста) выигрыш может оказаться очень большим. Пример — утилита diff.

LCS для более чем двух строк

- Есть три строки, $S_1 = abacx$, $S_2 = baabca$ и $S_3 = babbac$.
- Для каждого символа из S_1 составляется список пар позиций этого символа в S_2 и S_3, упорядоченный по лексикографическому убыванию.
- Т.е., список для символа a будет (6,5), (6,2), (3,5), (3,2), (2,5), (2,2).
- Списки записываются в порядке, определяемом строкой S_1: получается последовательность пар $\Pi(S_1, S_2, S_3)$.
- Возрастающая подпоследовательность пар — такая подпоследовательность, первые и вторые числа которой образуют возрастающую подпоследовательность (независимо друг от друга).

LCS для трёх строк

Теорема

Любая возрастающая подпоследовательность в $\Pi(S_1, S_2, S_3)$ *определяет общую подпоследовательность* S_1, S_2, S_3 *той же длины, и наоборот. Поэтому наибольшая общая подпоследовательность* S_1, S_2, S_3 *соответствует наибольшей возрастающей подпоследовательности в* $\Pi(S_1, S_2, S_3)$.