Ускорение поиска - идея

- При поиске образца в отсортированном массиве строк *S* можно экономить на некоторых сравнениях символов
- На каждой итерации бинарного поиска уточняется диапазон, внутри которого может находиться искомый элемент массива
- Все строки внутри диапазона могут иметь общий префикс с искомой подстрокой
- У строк вне диапазона такого общего префикса гарантированно нет (в необработанной части образца)
- Цель: по возможности вычислять очередной диапазон за константное время

Ускорение поиска – базовое свойство

- Обозначим l – левую границу выделенного диапазона, r – правую, P – искомый образец
- Введем функцию *lcp* (longest common prefix), которая получает пару строк, возвращая длину их наибольшего общего префикса
- Предположим, что известны длины общих префиксов образца с границами диапазона: $ll = lcp(P, \mathrm{S}[l])$ и $lr = lcp(P, S[r])$
- *Лемма*. Для любой строки $S[m]$ внутри диапазона $[l, r]$ значение $lm = lcp(P, S[m]) \geq \min(ll, lr)$. (В случае $lm < \min(ll, lr)$ нарушится упорядоченность)
- Это свойство применяется для уменьшения числа сравнений символов
- Т.к. общий префикс образца и строки внутри диапазона не меньше чем $mlr = min(ll, lr)$, то при сравнении символы в количестве mlr можно пропустить
- Сравнения образца достаточно начинать лишь с позиции mlr.

Иллюстрация базового свойства

Вычисление функции *lcp*

- При бинарном поиске образец посимвольно сравнивается со строкой в середине $m = (l + r) / 2$ очередного диапазона
- Сравнение прекращается при несовпадении очередной пары символов
- Длина совпадающей части P и $S[m]$ стала известной – $lcp(P, S[m])$
- В зависимости от характера несовпадения, позиция m замещает одну из границ l или r
- Вместе с переопределением границы сохраняется и $lcp(P, S[m])$ для этой границы

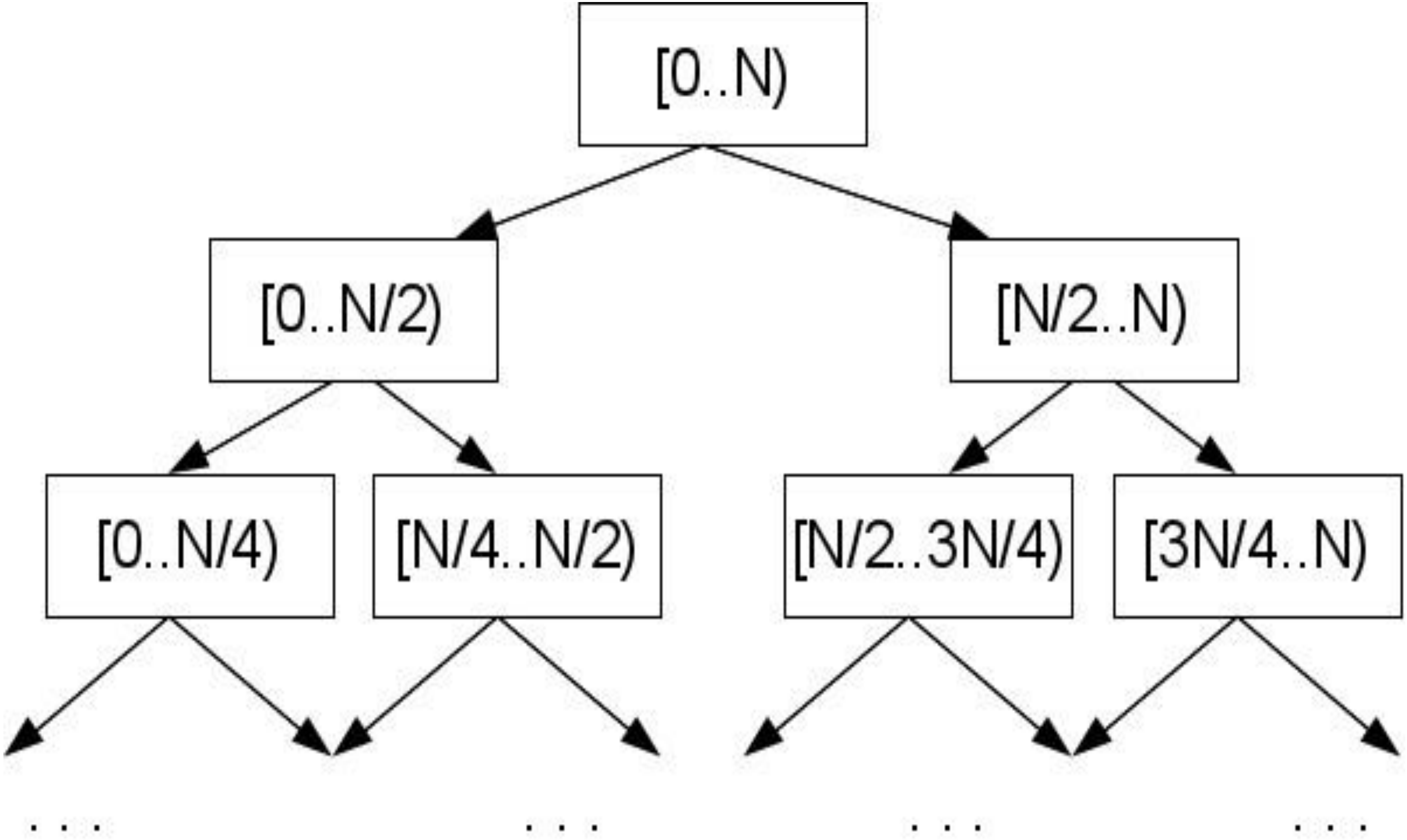
Об оценке сложности

- На основе *lcp* оптимизируется посимвольное сравнение при бинарном поиске образца в упорядоченном массиве строк
- Оценка в худшем случае от этого не снижается: если, например, искомый элемент расположен на краю массива, а соседний имеет с ним нулевой *lcp*, то придется проверять каждый символ $\lg n$ раз
- Однако эксперименты свидетельствуют о том, что для естественных текстов время поиска существенно сокращается по сравнению с $O(m\lg n)$
- При этом улучшение оценки сложности оптимизированного алгоритма может иметь лишь вероятностный характер

Повышение эффективности поиска

- Позиции l, r границ участка массива определяют центр m этого участка
- Все такие «триады», возникающие в бинарном поиске, представляют подмножества набора позиций в количестве $n - 2$ (исключены края массива)
- Предположим, что на стадии препроцессинга для всевозможных триад (l, m, r) найдены величины вида $ml = lcp(S[l], S[m])$ и $mr = lcp(S[m], S[r])$
- Цель: при бинарном поиске по возможности вычислять очередной диапазон за константное время

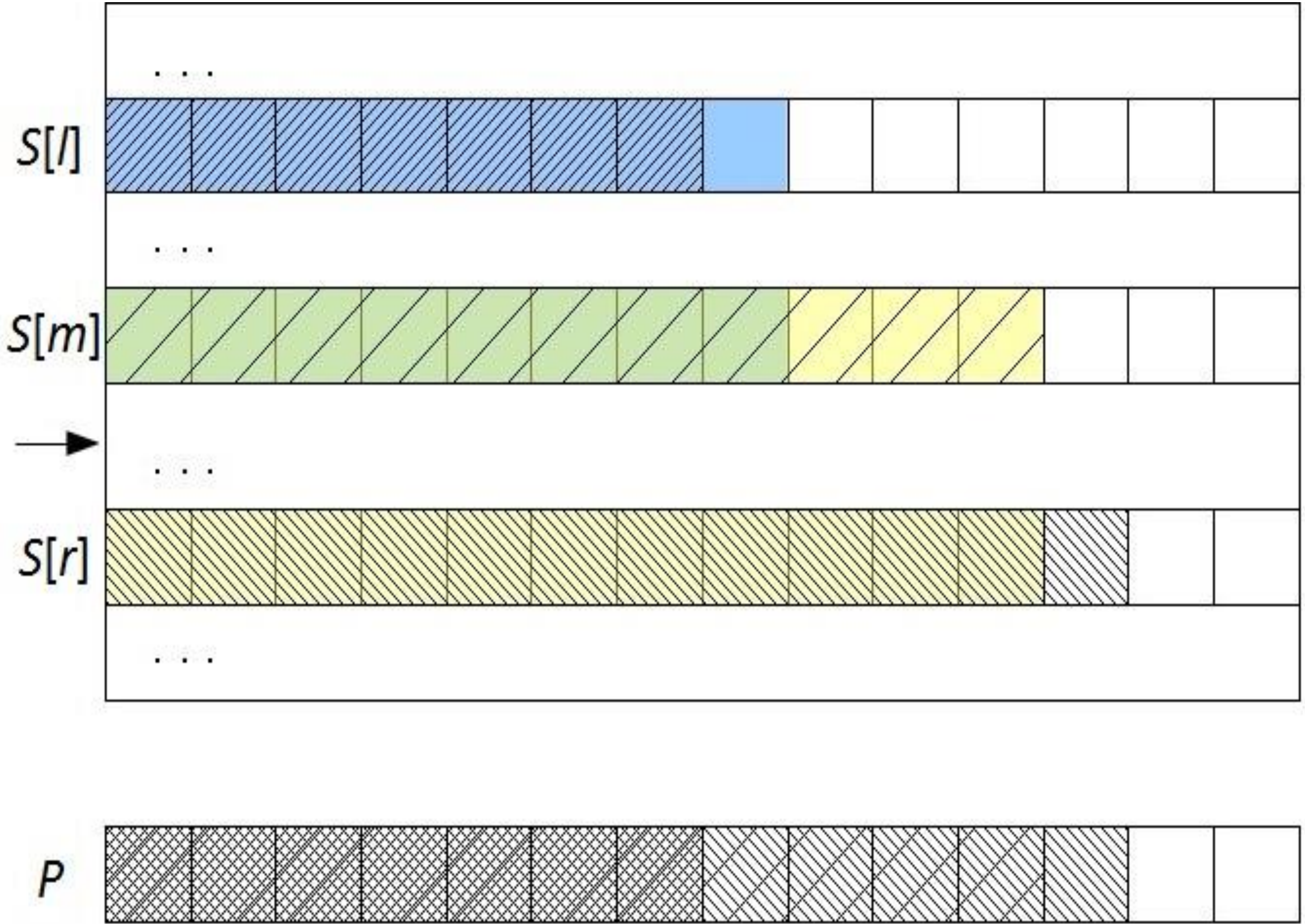
Триады при бинарном поиске



Процесс бинарного поиска

- Для очередной триады рассмотрим одну из границ, например, левую
- Если $ll = lcp(P, S[l]) < ml = lcp(S[l], S[m])$, то левая граница совпадает с серединой (и любым элементом между ними) больше, чем образец
- Тогда гарантированно $lcp(P, S[m]) = ll$
- При этом отрицательный результат сравнения образца заранее известен для всего участка от l до m
- В этом случае искать образец в другой половине, переопределив l и используя ll
- Если $ll > ml$, то середина совпадает с левой границей меньше, чем искомый образец
- Тогда есть шанс найти его на этом участке – от левой до средней границы, в альтернативном – нет
- Аналогичные рассуждения актуальны, если исследовать правую границу триады вместо левой

Бинарный поиск – иллюстрация шага



Бинарный поиск – случай двух равенств

- *При ll < ml* и *lr < mr* искомого образца в массиве нет; вариант двух неравенств *ll > ml* и *lr > mr* невозможен, поскольку противоречит *лемме*
- Не рассмотрен случай: *ll = ml* и *lr = mr* (если одно из этих равенств неверно, результат получается согласно предыдущим рассуждениям для неравенств)
- Эти два соотношения означают следующее:
  - образец совпадает с левой границей настолько, насколько она совпадает с серединой
  - образец совпадает с правой границей настолько же, насколько она совпадает с серединой
- То есть *lcp*(*P*, *S*[*m*]) не меньше чем *max*(*ll*, *lr*)
- Для точного нахождения *lcp*(*P*, *S*[*m*]) и пути бинарного поиска потребуется сравнение символов *P* и *S*[*m*] начиная с позиции *max*(*ll*, *lr*)
- Это единственная ситуация, когда сравниваются отдельные символы

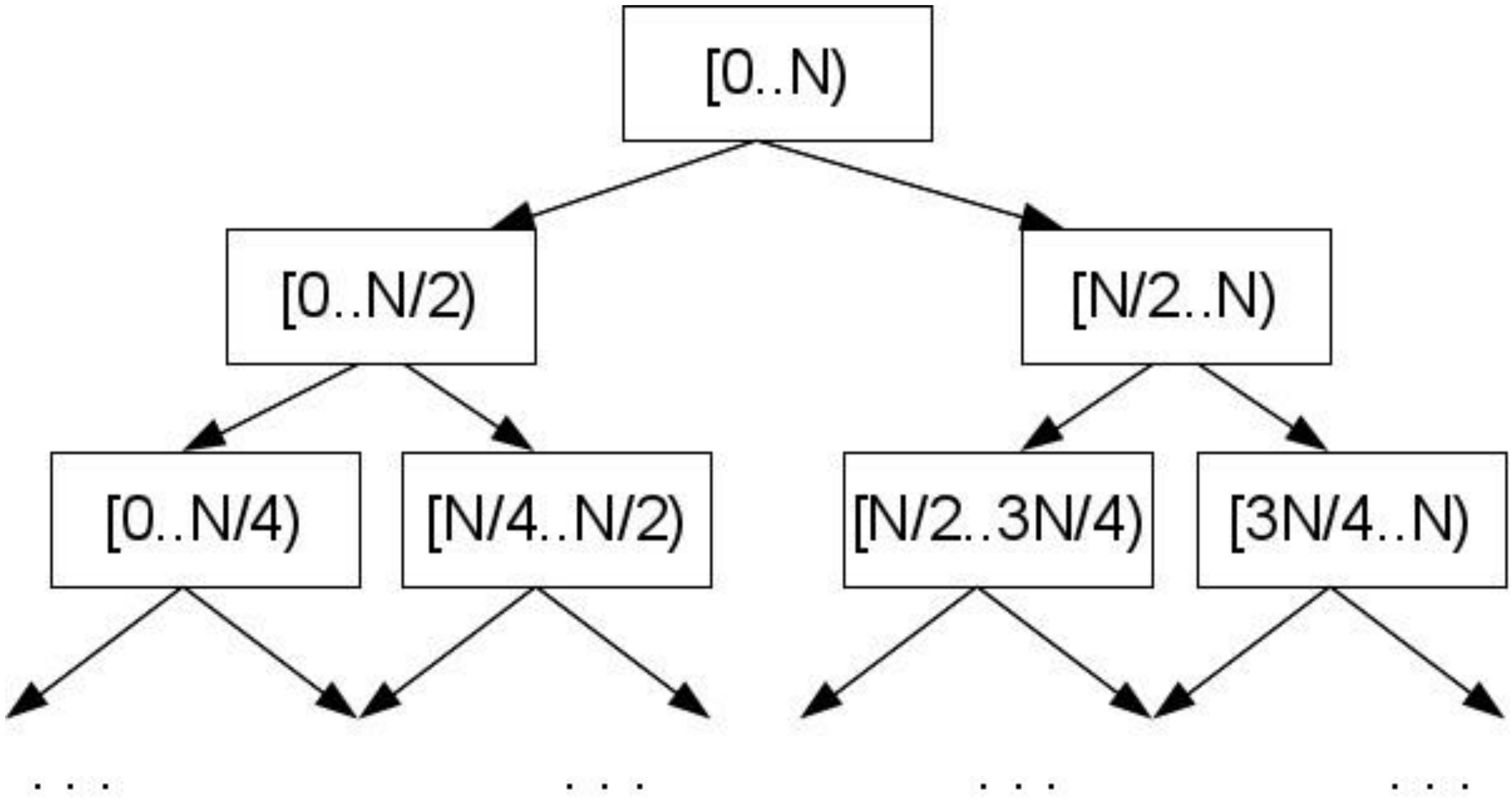
Бинарный поиск – оценка сложности

- Значительная часть шагов бинарного поиска осуществляется на основе константных сравнений величин *lcp* между собой
- При монотонном сужении рассматриваемого диапазона величина *max(ll, lr)* не уменьшается
- Символы сравниваются не более одного раза на каждый символ *P* (выбирается позиция *max*(*ll*, *lr*) и из-за ее монотонности в *P* нет возвратов)
- Итого оценка сложности алгоритма поиска = $O(\lg n + |P|)$
- Кроме того, требуются заранее подготовленные значения *lcp* для суффиксов текста (триад)

О препроцессинге функции *lcp*

- Подробное изучение этого вопроса не входит в настоящий курс. Он достаточно подробно освещен в *научной* литературе
- Существенную роль играет упорядоченность строк массива суффиксов
- Одна из схем – нахождении *lcp* между парами соседних элементов упорядоченного массива, но перебираются суффиксы в порядке их следования в исходной строке
- Полезно соответствие между массивом рангов и суффиксным массивом
- На последующих шагах диапазоны удваиваются и вычисляются их *lcp* (продвижение снизу вверх по дереву диапазонов)
- Вычисление *lcp* по результатам предыдущего шага (для объединения двух соседних диапазонов) сводится к задаче поиска минимума на отрезке (*range minimum query – RMQ*)
- Она может быть решена на стадии препроцессинга для всех необходимых пар за линейное время

Дерево диапазонов



Односторонние данные

- Если известны только левые значения *ml* (или только правые *mr*), то алгоритм остается работоспособным
- Нет возможности заранее решить, что вхождение отсутствует (*ll* < *ml* и *lr* < *mr*), однако оценка сложности алгоритма не повышается
- Пусть на очередном шаге имеется только *ml*. Если *ll* < *ml*, то сдвигаем *ll* на середину и продолжаем поиск в другой половине триады
- Если *ll* > *ml*, то ищем в этой половине
- Наконец, если *ll* = *ml*, то сравниваем серединный элемент с искомой строкой начиная с *ll*-ой позиции
- Возвратов по *P* не будет и в этой ситуации
- Таким образом, можно обойтись значениями *lcp* только для левых (или только для правых) границ без повышения сложности алгоритма

Построение суффиксного массива

- Известен ряд алгоритмов, характеризующихся оценкой вычислительной сложности, требованиями к памяти, а также и трудностями понимания
- Наиболее простой и наивный способ – прямая сортировка массива
- Лучшая сортировка массива без ограничения значений элементов при константной функции сравнения займет $O(n\lg n)$
- Поскольку сравниваются строки длиной порядка n, то оценка – $O(n^2\lg n)$
- Существуют усовершенствованные алгоритмы, работающие за $O(n\lg n)$, а также и $O(n)$

Использование суффиксного дерева

- За $O(n)$ построим суффиксное дерево
- Обойдя его в глубину, получим за время $O(n)$ суффиксный массив
- Дуги каждой вершины необходимо перебирать в лексикографическом порядке их начальных символов
- Дуги строго упорядочены, поскольку их начальные символы различны
- Однако суффиксное дерево критикуется за расходуемый объем памяти
- Поэтому не всегда целесообразно его присутствие в памяти одновременно с формируемым массивом
- Такой способ построения суффиксного массива оправдан, если предполагается его длительное использование, а построение выполняется относительно редко

Непосредственный алгоритм

- Далее будет представлен метод построения суффиксного массива сложностью $O(n\lg n)$
- На практике такой алгоритм вполне может оказаться удовлетворительным
- В основе лежит линейный алгоритм поразрядной сортировки
- Поэтому рассмотрим его основные положения

Сортировка подсчетом – идея

- Этот алгоритм сортировки применим для последовательности, *n* элементов которой представляют собой целые неотрицательные числа
- Они должны содержаться в ограниченном диапазоне (меньше заранее известного числа *K*)
- Если $K=O(n)$, то алгоритм сортировки подсчетом работает за время $O(n)$
- Идея: для каждого элемента *x* подсчитать, сколько есть элементов меньше *x*, и записать *x* в выходной массив в соответствии с этим числом
- Если, например, 17 элементов меньше *x*, то в выходном массиве *x* должен быть записан на место с индексом 17
- Если в последовательности могут быть равные числа, схему необходимо модифицировать, чтобы не записать несколько чисел на одно место

Сортировка подсчетом – алгоритм

- A[0..n-1] – исх. массив, B[0..n-1] – результат, C[0..K-1] – вспомогательный

Counting-Sort(A, B, K, n)

```
{
  for  (i = 0; i < K; ++i)  C[i] = 0;
  for  (j = 0; j < n; ++j)  C[A[j]] = C[A[j]] + 1;   //  C[i] - кол-во элементов == i
  for  (i = 1; i < K; ++i)  C[i] += C[i – 1];   //  C[i] - кол-во элементов <= i
  for  (j = n - 1; j >= 0; --j)
  {
    B[C[A[j]] - 1] = A[j]; // -1: сам эл-т тоже не превосходит себя
    C[A[j]]  =  C[A[j]] - 1; // Для учета равных элементов
  }
}
```

Сортировка подсчетом – комментарии

- После инициализации в $C[i]$ записывается количество элементов массива A, равных i
- С учетом монотонности i, нахождением частичных сумм в $C[i]$ вычисляется количество элементов, не превосходящих i
- Наконец каждый элемент помещается на нужное место в массиве B
- Если все n элементов различны, то число A[j] должно стоять на месте с индексом $C[A[\mathrm{j}]] - 1$, так как ровно столько элементов не превосходят A[j]
- На случай повторений в A, после записи A[j] в B величина $C[A[\mathrm{j}]]$ уменьшается. При следующей встрече с числом = A[j], оно будет записано на одну позицию левее
- Алгоритм состоит из одноуровневых циклов, каждый из которых повторяет вычисления K или n раз
- Таким образом, его сложность равна $O(K+n)$. Если $K=O(n)$, то время работы составит $O(n)$

Сортировка подсчетом – устойчивость

- Алгоритм обладает важным свойством *устойчивости*
- Если во входном массиве содержатся равные числа, то в выходном массиве они располагаются в том же порядке
- Это свойство обеспечивается обратным просмотром *A* в заключительном цикле
- Оно несущественно, если есть только сами числа. Если с числами хранятся дополнительные данные, может оказаться важным
- Свойство устойчивости алгоритма сортировки применяется в реальных задачах
- Имеет важное значение для описываемого далее алгоритма поразрядной сортировки

Поразрядная сортировка – введение

- Сортировка подсчетом эффективна при не слишком большом *K*. Например, при $K=O(n)$ работает за линейное время по *n*
- Если диапазон чисел велик, могут быть проблемы с объемом памяти: используется вспомогательный массив *C* размера *K*
- Кроме того, этот массив обрабатывается, что соответствующим образом отражается и на вычислительной сложности алгоритма
- Идея сортировки подсчетом модифицирована с целью более адекватного применения для чисел широкого диапазона
- В алгоритме поразрядной сортировки существенным условие – ограничение разрядности чисел *l*

Пример

- Массив трехзначных (*l* = 3) чисел, могли бы быть и многозначными
- При различной разрядности могут быть дополнены незначащими нулями
- Последовательно сортируем их по одной цифре, начиная с младшей
- Важно, что каждая сортировка по очередной цифре *устойчива*

Исх.	По 1 цифре	По 2 цифре	По 3 цифре
635	102	102	102
124	124	108	108
578	235	124	124
102	578	635	578
108	108	578	635

Результат примера

- Числа в последнем полученном массиве отсортированы по старшей цифре
- Для чисел с одинаковой старшей цифрой в силу устойчивости сортировок сохраняется порядок следования по предыдущей цифре и т.д.
- Таким образом, после *l* стадий всегда получается отсортированный массив
- Формально корректность алгоритма доказывается индукцией по номеру разряда
- Это есть *поразрядная сортировка*, называемая еще *цифровой*
- Можно аналогично работать с числами высокой разрядности и даже со строками, если их алфавит «оцифрован»
- На каждой стадии – распределение элементов по *классам эквивалентности* (с одинаковым набором значений отсортированных разрядов)
- Это касается и сортировки массива строк, где в качестве «разрядов» фигурируют символы

Поразрядная сортировка – о реализации

- Нужно выбрать эффективный алгоритм, реализующий представленную идею
- Сортировку по каждой цифре предлагается выполнять подсчетом, то есть предыдущим алгоритмом
- Потребуются массив C длины k (верхняя граница цифры) для подсчета присутствия цифр на каждой стадии и массив B для перезаписи чисел
- При работе с длинными числами или строками можно перезаписывать не элементы массива, а указатели на них (или их индексы)
- Для n чисел с l знаками от 0 до $k – 1$ каждая стадия займет время $O(n + k)$
- Всего l стадий $\Rightarrow$ время работы поразрядной сортировки равно $O(ln + lk)$
- Если l фиксировано и $k=O(n)$, то поразрядная сортировка работает за линейное время