Школа молекулярной и теоретической биологии

IV сезон

БИОАЛГОРИТМИКА

Расширенный конспект

М. А. Ройтберг, Ильдар Хисамбеев

Преподаватели: Влад Белоусов, Белла Дасаева

Слушатели: Михаил Большелапов, Анна Лазарева, Алена Потапенко, Conrad Cardona

Пущино 9-17 августа 2015 г.

Часть 1.
СЛОЖНОСТЬ АЛГОРИТМОВ и СТРУКТУРЫ ДАННЫХ

9.08 Занятие 1.

1. Введение.

Замечание. Слушатели уже имеют (небольшой) опыт программирования. Цель этого раздела – систематизировать то, что они знают и ввести терминологию, которая будет использоваться в дальнейшем. Конец замечания.

1.1.<u>Модель вычислительного устройства</u> (иногда говорят «*модель вычислений*»): *процессор,* т.е. устройство для выполнения *элементарных операций, + память.*

Память состоит их ячеек. В ячейке могут храниться целые числа, символы и другие объекты (уточним позже по потребности).
Ячейка может иметь имя (*переменная*). То, что лежит в ячейке – *значение* переменной.
Зная имя, доступ к переменной (чтобы узнать ее значение или записать новое значение) осуществляется за фиксированное время.
Группа идущих подряд ячеек может рассматриваться, как *массив* – блок перенумерованных ячеек. У массива есть имя. Зная имя массива и номер элемента в массиве (*индекс*) доступ к этому элементу осуществляется за фиксированное время.

Допустимые операции:
- операции над числами (арифметические операции, сравнения);
- операции с памятью (присваивание, чтение).

Это – не полное описание модели вычислительного устройства. Будем уточнять по потребности.

Для простоты вместо «модель вычислительного устройства» будем говорить «*компьютер*».

<u>1.2. Алгоритм</u> – описание последовательности операций (*шагов)* компьютера, которые он должен сделать для решения определенной задачи.

При постановке задачи, которую выполняет алгоритм, будем явно описывать «**дано**» (входные данные) и «**надо**» (выходные данные).

1.3.<u>Псевдокод для записи алгоритмов</u>. Соглашения:

- Похож на Паскаль, но со вставками на естественном языке.
- Действие, описанное неформально, будем выделять *курсивом* и предварять ключевым словом **DO:**
- Опускаем (иногда) «инженерные» детали: объявление переменных, типизацию, обработку ошибок, etc.
- Блоки кода структурируются операторными скобками (например, alg – end_alg) и отступами; иногда операторные скобки будем опускать.
- Элементы массива могут обозначаться нижним индексом или квадратными скобками. Индексы (как правило) нумеруются с нуля.

Пример: Алгоритм поиска максимума

```
max = A[0]
maxi = 0
for i in 1..n-1
    if A[i] > max
        max = A[i]
        maxi = i
```

<u>*1.4.* Время работы алгоритма.</u>

Время работы алгоритма (при определенных исходных данных) – суммарное количество выполненных операций (*шагов*).

Время работы алгоритма ***T*** зависит от размера входных данных ***N*** и от их конкретных значений.

Среди входных данных одного размера ***N*** рассматриваем (пока) худший случай (worst case scenario).

Зависимость T(N) оцениваем асимптотически, то есть нас интересует ее *порядок роста*: пренебрегаем константами и членами нестаршего порядка (пояснения ниже).

2. **Пример алгоритмической задачи – сортировка.**

2.1. Постановка задачи

Дан массив целых чисел A[1..N]. Расположить числа в массиве в порядке неубывания.

2.2. «Наивный» алгоритм сортировки.

Идея:

- найдем максимальный элемент и поместим его в конец (поменяем местами с последним элементом);
- в оставшемся массиве сделаем то же самое;
- будем повторять, пока все не отсортируем (не дойдем до массива длины 1)

```
alg NaiveSort
  for n in N..2
  утв Последние N-n элементов массива упорядочены по возрастанию
  утв Каждый из первых n элементов меньше (не больше) каждого из N-n последних
  assert Last N-n elements are incrementally ordered
  assert Each of first n elements is less (not greater) than each of last N-n elements
    max = A[0]
    maxi = 0
    for i in 1..n-1
      if A[i] > max
        max = A[i]
        maxi = i
      endif
      A[maxi] = A[n-1]
      A[n-1] = max
    end_for
  end_for
end_alg
```

Замечание 1. «утв» (англ. “assert”) – ключевое слово языка записи алгоритмов. После него пишется утверждение о данных при прохождении соответствующей строки.

Если утверждение о состоянии данных стоит в начале цикла, его обычно называют *инвариантом цикла.*

Утверждения о состоянии данных нужны для *доказательства корректности алгоритма.*

Их можно использовать при отладке программ.

Конец замечания.

Подсчитаем количество операций в худшем случае: $\frac{5}{2}N^2 + \frac{7}{2}N - 6$

Замечание 2. Напомним: нас интересует *порядок роста* функции T(N). Это (в данном случае) записывают так: $\boldsymbol{T(N) = O(N^2)}$

Это означает (примерно), что $T(N)/(N^2)$ стремится к какой-то константе, когда N стремится к бесконечности.

Говорят: «T(N) растет, как N^2»

3. Слияние упорядоченных массивов.

3.1. Можно ли сортировать быстрее? Вспомогательная задача: слияние упорядоченных массивов.

Дано: A[1..N], B[1..M] – массивы, упорядоченные по возрастанию.

Получить: C[1..N+M] – массив, содержащий все элементы массивов A и B, упорядоченные по возрастанию.

Идея 1. Сравниваем первый элемент массива B с последним элементом массива A

```
array A[1..M], B[1..N]
array C[1..M+N]
C[1..M] := A[1..M]
for k from 1 to N
утв Первые N+k-1 элементов массива C упорядочены по возрастанию
утв Они содержат все элементы массива A и k-1 первых элементов
     массива B
assert First N+k-1 elements of the array C are incrementally
       ordered.
assert They contain all elements of the array A and first k-1
       elements of the array B
  // find place p of B[k] in C
  p := N+k
  while B[k]<C[p] AND p > 0
    p:=p-1
  end_while
  DO: insert B[k] after C[p]
end_for
```

Посмотрим на время.

Для начала будем считать только количество сравнений TComp

В худшем случае TComp(M, N) ~ N*M

Если повезет: TComp(M, N) ~ N+M.

Замечание. Полное время включает не только сравнения, но и вставки элементов массива B в массив C. Вставка в середину массива – долгое дело. Нужно использовать не массивы, а другую *структуру данных,* например, – **списки**. Отложим разговор о списках и других структурах данных и будем пока считать только сравнения.

3.2 Что будет, если использовать схему «первый – первый», т.е. искать место для элементов массива B, начиная с начала массива A?

Напишите алгоритм по этой идее.

Продолжение следует

10.08 Занятие 2.

1. **Повторение**

2. **Алгоритм слияния «первый-первый» и сортировка на его основе.**

   2.1. Обсуждение.

   Что нужно изменить в алгоритме «первый-последний»?

   Идея алгоритма «первый-первый»:

   1) Сравниваем B[1] с A[1], A[2] и т.д., пока не найдем такое k, что A[k]>=B[1] (или массив A закончится).
   2) Вставляем B[1] передA[k] (или после всех элементов массива A).
   3) Берем B[2] и начинаем его сравнивать с элементами массива A, начиная с A[k] (или просто записываем после B[1], если B[1] больше всех элементов массива A)
   4) И т.д.

   Время работы (набросок). Каждый элемент массива B сравнивается со «старым» элементом массива A (то есть элементом, который уже сравнивался с предыдущими элементами массива B) не более одного раза. Таким образом, количество сравнений со «старыми» не более N (длина B). Количество сравнений с «новыми» (по определению) – не более M (длина A). Таким образом, общее количество сравнений – не более N+M. Ура!

   2.2. Слтяние 4-х массивов. Балансировка.

   Вопрос: Мы уже знаем, что для слияния двух массивов длины N/2 каждый нужно не более N/2 + N/2 = N сравнений. Сколько сравнений нужно для слияния 4-х массивов длины N/4?

   Решение.

Этап 1. Сначала сливаем 1-й и 2-й массивы – потребуется не более N/4 + N/4 = N/2 сравнений. Аналогично, сливаем 2-й и 4-й массивы – еще N/2 сравнений. Всего – N/2 + N/2 = N сравнений.

Этап 2. Теперь сливаем два новых массива, длина каждого из них – N/2. Потребуется не более N/2 + N/2 = N сравнений.

Итак, на каждом этапе – не более N сравнений. Всего – не более 2N сравнений.

Замечание. Важно, что мы разбили 4 массива на две пары, а не «приливали» массивы по одному. В последнем случае было бы вот что:

1) Слияние 1-го и 2-го: не более N/4 + N/4 = N/2 сравнений
2) «Приливание» 3-го: не более N/2 + N/4 = 3N/4 сравнений
3) «Приливание» 4-го: не более 3N/4 + N/4 = N сравнений

   Всего – N/2 +3N/4 + N = 2N +N/4 сравнений.

Во многих задачах полезно, чтобы обрабатываемые данные разбивались на части примерно одинакового размера. Такой прием называется *балансировкой.* Мы с ним еще встретимся.

2.2. Сортировка слиянием (набросок)

1) Пусть исходно было 2^t упорядоченных массивов длины $N/2^t$ каждый. Тогда для получения из них слиянием одного массива длиной N понадобится t этапов, а на каждом этапе – не более N сравнений. Таким образом, всего за t этапов – не более t*N сравнений.

2) Возьмем t таким, чтобы 2^t было не меньше N. Например, пусть t – это наименьшее целое число, которое не меньше, чем $\log_2 N$ (такое число будем обозначать lob(N), буква b – от слова binary ☺).

Тогда в исходных массивах будет не более одного элемента, то есть эти массивы уже будут упорядоченными! Значит, массив длины N можно упорядочить, выполнив, не более N*lob(N) сравнений.

ВНИМАНИЕ! Мы учитывали только сравнения и не учитывали время, необходимое на

другие операции (например, вставку элемента внутрь массива, разбиение массива на более мелкие массивы). Чтобы разобраться с этим нужно записать наши алгоритмы (алгоритм слияния двух упорядоченных массивов и алгоритм сортировки разбиением массива на части и слиянием их) более формально. Этим и займемся. Начнем с алгоритма слияния.

3. Реализации алгоритма слияния.

3.1.Наивная реализация.

Реализация №1. Массив A переписываем в новый массив размера M+N. Результат – в этом новом массиве (см. рис.1).

```
1. array  A[1..M], B[1..N]
2. array  C[1..M+N]
3. C[1..M] := A[1..M]
4. p:=1 // указатель на элемент массива C, с которым нужно сравнивать
5. for k from 1 to N
6. assert First N+k-1 elements of the array C are incrementally
          ordered.
7. assert They contain all elements of the array A and first k-1
          elements of the array B
8.   // find place p of B[k] in C
9.   p := p + 1  // номер элемента, с которым нужно сравнивать,
                 // увеличен на 1 из-за вставки элемента перед ним
10.  while B[k] > C[p] AND p < N+k
11.      p:=p+1
12.  end_while
13.  DO: insert B[k] before C[p]
14.end_for
```

Рис.1.

Утверждение. Количество сравнений ~ N+M

Объяснение. После каждого выполнению сравнения в строке 10 значение переменной p увеличивается на 1. Если неравенство истинно, то p увеличивается в строке 11. Если ложно – в строке 9. Поэтому проверка неравенства выполняется столько раз, сколько разных значений принимала переменная p (в начале p=1). Максимальное возможное значение p = N+M. Это и дает оценку. Конец объяснения.

Замечание. Время работы будет ~ N*M (в худшем случае). Потому, что кроме сравнений, нужно еще выполнять вставки элементов внутрь массива, а для этого придется «сдвигать» (т.е. переписывать на новое место) все, что было правее места вставки. Конец замечания. Конец замечания.

Таким образом, у этой реализации – два недостатка:

1) Используется дополнительный массив
2) Нужно много времени на переписывание при вставке.

   Каждый из этих недостатков в отдельности можно исправить (см. пп. 2.3, 2.4). А оба вместе – не получится, если хранить данные в массивах: массивы не приспособлены для вставки элементов «внутрь». Что делать – см. п.3.

3.2. Реализация без использования дополнительной памяти.

Реализация №2. Массивы A и B представлены, как фрагменты одного массива C: массив A занимает NA позиций, начиная с позиции Start; массив B занимает NB следующих позиций. Результат – в той же области массива C. При вставке элемента происходит перезапись элементов справа от места вставки на новое место (сдвиг на одну позицию вправо)

```
algorithm  MergeArray(array C[1..L], int Start, NA, NB)
  // array  C[1..L]
```

```
  // int Start, NA, NB :
  //    массив A – C[Start..Start+NA-1]
  //    массив B – C[Start+NA..Start+NA+NB-1]
  p:= Start-1 // (p+1) - указатель на элемент массива C,
          // с которым нужно сравнивать очередной элемент массива B
          //
  for k from Start+NA to Start +NA + NB-1
  assert Elements of the array C[Start..Start+NA-1+k-1
        are incrementally ordered.
  assert They contain all elements of the array A and first k-1
         elements of the array B
   assert Elememts C[Start..Start+NA-1+k..Start+NA-1+NB] contain last
         NB-k+1 elements of the array B
      // find place p of B[k] in C
    p := p + 1  // номер элемента, с которым нужно сравнивать,
                // увеличен на 1 из-за вставки элемента перед ним
    while C[Start+NA-1+k] > C[p] AND p < Start+NA-1+k
      p:=p+1
     end_while
     DO: insert C[Start+NA-1+k] before C[p]and
         remove C[Start+NA-1+k] from the array
   end_for
end_alg
```

Рис.2.

```
algorithm  MergeArrayNew(array C[1..L], int Start, NA, NB; array
D[1..L])
  // array  C[1..L]
  // int Start, NA, NB :
  //    массив A – C[Start..Start+NA-1]
  //    массив B – C[Start+NA..Start+NA+NB-1]
  kA:= 0 // количество обработанных элементов массива A
  kB:= 0 // количество обработанных элементов массива B
  //     1. Слить массивы там, где они «перемежаются»
  while kA < NA and kB < NB
  утв первые kA элементов массива A и первые kB элементов массива B
      совместно отсортированы и записаны в позиции массива D,
начиная
      с позиции Start
      pA:=Start+kA//позиция первого необработанного элемента массива
A
      pB:=Start+kB//позиция первого необработанного элемента массива
D
      pD:=Start+kA+kB//куда писать в массиве D
      // Найти наименьший элемент в оставшейся части массивов A и B
      //   и записать его в массив D
      if C[B] < C[A] then
        D[pD]:= C[pB]
        kB:=kB+1
      else
        D[pD]:= C[pA]
        kA:=kA+1
      endif
   end_while
   //   2.Дописать в конец остаток одного из массивов (если нужно)
   while kA<NA
```

```
    pA:=Start+kA//позиция первого необработанного элемента массива
A
    pD:=Start+kA+NB//куда писать в массиве D
    D[pD]:= C[pA]
    kA:=kA+1
  end_while
  while kB<NB
    pB:=Start+kB//позиция первого необработанного элемента массива
A
    pD:=Start+NA+kB//куда писать в массиве D
    D[pD]:= C[pB]
    kB:=kB+1
  end_while
end_alg
```

Рис.3.

<u>3.3.Реализация без вставок (все пишем на новое место).</u>

Реализация №3. Массивы A и B представлены, как фрагменты одного массива C: массив A занимает NA позиций, начиная с позиции Start; массив B занимает NB следующих позиций.. Результат – в аналогичной области нового массива D. Все элементы сразу записываются в новую область в порядке возрастания, поэтому вставок и сдвигов не нужно

3.4.Сортировка слиянием для массивов

Постановка задачи:

Дано: массив C[1..N]

Получить: массив C[1..N]; все элементы – в возрастающем порядке.

Алгоритм приведен на рис.4. Для слияния массивов на очередном этапе использован алгоритм `MergeArray` (см. рис.2).

Замечание. Можно было бы использовать и алгоритм с записью результата в новый массив (алгоритм `MergeArrayNew`, см. рис.3). Тогда нужно или после каждого этапа переписывать данные из массива D в массив C (и тратить на это время) или по очереди на разных этапах в качестве сортируемого массива использовать то массив C, то массив D. В обоих случаях общее время работы будет ~N*lob(N). Конец замечания.

```
1. algorithm MergeSortArray
2.   array  C[1..N]
3.   int k=1  // Текущий размер «блока». В обозначениях п.2.2., k =
   2^t
4.   while k < N
     // Очередной этап (см. п.2.2)
     утв Массив C разбит на блоки длиной k (последний блок может быть
         короче). Каждый блок упорядочен по неубыванию.
5.     s :=1  // начало очередной пары блоков
6.     while s < N
           // слить два блока длины k
           // последний блок может быть короче
7.       NA:=k    // длина 1-го блока
8.       if (s+NA>N) break // остался один блок; ничего не делаем
            // есть 2-й блок: вычисляем его длину
9.       if (s+2*k-1 <=N)
10.      NB=k
11.   else
12.      NB = N-s-k+1
13.   end_if
14.      MergeArray(C, s, NA, NB)
```

```
15.         s := s+NA+NB
16.       end_while
17.       k :=  2*k
18.     end_while
19.   end_alg
```
Рис.4.

Оценим количество сравнений
При выполнении вызова MergeArray (строка 14) выполняется NA+NB сравнений.
Значит, при одном проходе цикла “while s<N” (строка 6) выполняется всего не более N сравнений.
Так как при каждом выполнении цикла “while k<N” (строка 4) значение k увеличивается в 2 раза, то тело этого цикла (включая тело цикла “while s<N”) выполняется $[\log_2 N]+1$ раз.
Итого: общее количество сравнений $\sim N*\log_2 N$

Замечание. Важно, что на каждом этапе (прохождении цикла“while k<N”) у нас все блоки примерно одной длины. Это и позволяет сделать общее количество этапов $\sim \log_2 N$
Итак, используя массивы, можно либо отсортировать набор чисел за время ~N*lob(N) с использованием дополнительной памяти размера ~N, либо не использовать дополнительной памяти, но при этом время работы будет $\sim N^2$. Посмотрим, можно ли отсортировать набор чисел за время ~ N*lob(N) без использования дополнительной памяти.
.

4. Структуры данных. Сетевые структуры.

Структура данных – общее понятие для чего-то, где хранится определенное множество данных. В наших примерах данные – это целые числа, но это не обязательно. Для конкретной структуры данных известно, как устроены «единицы хранения» (ячейки) и как можно осуществлять доступ к ним (извлекать данные и записывать данные). Разные структуры данных позволяют удобно осуществлять разные операции над множествами. Примеры операций: получить элемент (GET), найти максимальный элемент множества (MAX), выяснить, есть ли в множестве заданное число (FIND); упорядочить элементы множества (SORT); добавить элемент (INSERT), удалить элемент (DELETE).

До сих пор мы умели хранить данные в отдельных ячейках (переменных, у каждой – свое имя) или в массивах. В массиве у каждого элемента тоже есть свое имя (пример: A[127]), но эти имена – стандартные (имя элемента сострит из имени массива и индекса элемента массива). Короткая (не зависящая от размера массива) программа позволяет перебрать все элементы массива, перебирая все индексы.

Пример структуры данных 1. Массив.

Каждая ячейка имеет *индекс* – целое число из определенного интервала. Доступ к ячейке осуществляется указанием имени массива и индекса ячейки.

Если в массиве N элементов, то

- операции MAX и FIND выполняются за время ~N;
- операция SORT выполняется за время $\sim N^2$ (без использования дополнительной памяти);
- если нам важен порядок расположения элементов в массиве, то операции вставки элемента в массив и удаления элемента из массива выполняются за время ~ N (нужно сдвигать элементы после места вставки/удаления)

Наборы (множества) чисел (и других объектов) не обязательно хранить в массивах, в подряд идущих перенумерованных ячейках. Другой способ хранения – сетевые структуры и простейшая из них – *(односторонний) список.*
В сетевых структурах единица хранения (*запись*) имеет сложную структуры – она разбита на отдельные ячейки (они называются *полями*); все записи в структуре имеют одинаковую

структуру. Ячейки (поля) бывают двух типов: в ячейках первого типа хранятся данные (например, числа). В ячейках второго типа хранятся *ссылки* (иначе - *указатели*) – адреса других записей. Используя ссылки, можно переходить от одной записи к другой и получать доступ к записанным в них данных.

!!! В отличие от элементов массива, записи не имеют своих имен и получить доступ к определенной записи можно только добравшись до нее по ссылкам. Но во многих случаях сетевые структуры удобнее массивов. Например, в них можно легко вставлять новые записи – достаточно изменить ссылки у соседей новой записи.

Замечание. Мы используем слова «указатель» (pointer) и «ссылка» (reference) как синонимы. В современном программировании это не совсем так, но нам эта разница сейчас не важна. И то, и другое здесь будет обозначать адрес определенной записи в сетевой структуре. Основным термином у нас будет «указатель».

Простейшая сетевая структура – *список*. К ней и перейдем.

Продолжение следует.

11.08 Занятие 3.

1. **Списки.**

Список (иногда говорят – «*односторонний список*») - простейшая сетевая структура данных. В списке запись состоит из двух ячеек (*полей*) – как «доминошка». В основной ячейке доминошки хранятся данные, в дополнительной — указатель на другую доминошку – *следующий элемент* списка. Напомним: указатель содержит адрес в памяти, где расположена нужная запись и позволяет легко ее найти («*перейти к следующему элементу списка*»
В последней «доминошке» эта ячейка содержит признак конца списка. Существует специальная доминошка — заголовок списка, которая не хранит никаких данных, ее указатель указывает на 1-й элемент списка, содержащий данные.

!!! В списке нет циклов, т.е. начав двигаться по ссылкам от заголовка, мы пройдем все элементы и придем к *последнему* элементу, у которого нет следующего за ним элемента.

Замечание. Бывают списки, которые содержат не одну, а несколько ссылок. Например, в записи может быть ссылка не только на следующий, но и на предыдущий элемент списка (такие списки называют *двусторонними)*.
Кроме того, бывают списки, у которых запись содержит не одно число, а два или больше. Например, если у нас есть список точек на плоскости, то запись будет хранить два числа – координаты точки. Мы будем под списком (если не оговорено противное) понимать односторонний список, каждая запись которого хранит одно число. Конец замечания.

У списка, как и у массива есть свое имя. Но элементы списка не имеют своих имен (например, индексов, как в массиве). Для работы со списком есть отдельный указатель (*главный указатель списка, MainPointer*), который указывает на *текущий* элемент списка. В начальный момент указатель списка указывает на заголовок списка. Получить/записать значение можно только для текущего элемента списка. Используя поля ссылок в элементах списка, можно добраться до нужного элемента.

Со списком можно выполнить следующие операции (перед описанием операции указано ее сокращенное название; A – элемент массива, z – данное, хранящееся в элементе списка):

- Value(A)
  - значение текущего элемента (используется в правой части операторов присваивания и т.п.);
- PutValue(A, z)
  - записать в текущий элемент списка A значение z;
- ToNext(A)
  - перейти к следующему элементу списка A (то есть, объявить текущим следующий элемент списка; если текущий элемент списка был последним, то это действие выполнить нельзя);
- ToStart(A)
  - объявить заголовок текущим элементом (встать в начало списка);
- HasNext(A)
  - проверить, что текущий элемент списка A – не последний (возвращается значение ДА, если в списке A после текущего есть еще элементы, НЕТ, если текущий элемент – последний);
- InsertAfter(A)
  - вставить после текущего новый элемент с заданным значением (текущим элементом становится вставленный элемент);
- DeleteAfter(A)
  - удалить элемент, стоящий после текущего элемента (если текущий элемент – последний элемент списка, то операцию выполнить нельзя).

Каждую из этих операций можно выполнить за фиксированное (не зависящее от размера списка) время.

Замечание. Последняя операция позволяет удалить элемент, следующий за текущим, а не сам текущий элемент. При удалении текущего элемента нужно будет изменить указатель *предшествующего* элемента. А доступа к предыдущему элементу в односторонних элементах нет. Чтобы удалять текущий элемент, нужно использовать двусторонние списки.

Замечание. В «классических» списках напрямую с указателями мы работать не можем. Но иногда это оказывается полезным. Поговорим об этом позже. Конец замечания.

Если в списке N элементов, то
- операции MAX и НАЙТИ выполняются за время ~N;
- операция SORT выполняется за время ~N*logN (докажем позже);

2. Задача склейки списков. Расширенные возможности работы со списками

2.1. Задача склейки списков

Постановка задачи.
Дано: Список A, список B
Получить: Список A, состоящий из исходного списка, к которому в конце присоединен список B. Текущим элементом списка A является его заголовок.
Замечание. (Как всегда) в начале работы алгоритма текущий элемент списка – его заголовок.

Решение.
```
alg Сцепление (список A, список B)
assert Текущими элементами списков A и B являются их заголовки
assert После выполнения алгоритма к концу списка A добавлены все
      элементы списка B в том порядке, в котором они шли в списке B
             // Сделать текущим последний элемент списка A
  while HasNext(A)
    ToNext (A)
  end_while
             // Вставлять по одному элементы списка B
  while HasNext(B)
    ToNext(B)
    InsertAfter(A, Value(B))//новый текущий элемент – тот,
                              //    который вставлен
  end_while
  ToStart(A)
end_alg
```
Рис.1.

Время работы: ~ length(A) +length(B).
Как видим, это время – не лучше, чем при перезаписи массивов.
Можно ли быстрее?

2.2. Расширение возможностей: списки с прямым доступом к концу и явным использованием указателей (pointer)

Чтобы описать более быстрый алгоритм сцепления списков, понадобятся новые возможности работы со списками, которые позволяют более свободно обращаться с его элементами.

1) Прямой доступ к концу списка

Разрешим за одну операцию делать текущим последний элемент списка.

Для этого будем помнить, например, в заголовке, еще и указатель на последний элемент. Списки с возможностью прыжка в конец – это немного более сложная структура, чем тот список, с которого мы начинали.

Кроме того, будем считать, что за одну операцию можно определить длину списка (для этого это значение нужно помнить и корректировать каждый раз при удалении или добавлении элементов)

Обозначение:

ToEnd(A) - сделать текущим элементом списка A его последний элемент

Length(A) – возвращает длину списка A

2) Указатели.

До сих пор мы могли извлекать из ячейки списка только данное, которое там хранится (например, число). Ссылку на следующий элемент можно было использовать только для перехода к этому элементу (функция ToNext())

Разрешим использовать указатели так же, как числа – читать их значения и присваивать их. Только складывать и умножать указатели не будем ☺

Новые функции работы со списками:

- MainPointer(A) – значение указателя на текущий элемент списка A
- PointerNext(A) - значение указателя на следующий элемент в текущем элементе списка A (при вызове функции возвращается данное, записанное во вспомогательной ячейке текущего элемента списка (текущей «доминошки»); если текущий элемент – последний элемент списка, то возвращается специальное значение, которое обычно обозначается NULL
- SetPointerNext(A, r) - установить указатель на следующий элемент в текущем элементе списка A равным указателю r (часто r – это указатель, взятый из другого элемента списка или главный указатель другого списка)

Новый алгоритм сцепления списков

```
r:=PointerNext(B)
ToEnd(A)
SetPointerNext(A, r)
```

Можно записать и короче:

```
ToEnd(A)
SetPointerNext(A, PointerNext(B))
```

Время работы – не зависит от длин списков. Ура!

Упражнение 1. Написать алгоритм вставки списка B после текущего элемента списка A

Замечание. Упомянем еще одну важную функцию, которую используют при работе со списками (и другими сетевыми структурами). Это функция создания нового элемента

- NewElement(z, r) – создать новый элемент с данным z и указателем на следующий элемент r.

Эта функция возвращает указатель на созданный элемент и часто используется в операторе присваивания вида

```
rnew:= NewElement(z, r)
```

Конец замечания.

3. Алгоритм MergeSort на списках

Чтобы записать алгоритм MergeSort на списках нам потребуется еще расширить набор разрешенных операций над списками. В алгоритме MergeArray (см. Занятие 2, рис.2) мы представляли сливаемые массивы, как части одного массива C. Затем в алгоритме

сортировки массива MrgeSortArray (см. Занятие 2, рис.4) мы вызывали этот MergeArray для слияния частей сортируемого массива. Как работать с частями списка, как с отдельными списками?

Сейчас для данного списка А мы можем работать с только текущим элементом списка (см. список операций в п.1), то есть с элементом, на который указывает основной указатель списка MainPointer(A), Расширим список операций так, чтобы иметь возможность так же работать с любым элементом списка, на который указывает заданный указатель r.

Обозначения:

Value(r) – значение «доминошки», на которую указывает указатель r (сравни с функцией Value(A); теперь вместо Value(A) мы могли бы написать Value(MainPointer(A));

SetValue(r, z) – установить данное (т.е. значение в основной ячейке «доминошки») в элементе списка, на который указывает указатель r равным z;

PointerNext(r) - значение указателя на следующий элемент в элементе списка, на который указывает указатель r (при вызове функции возвращается данное, записанное во вспомогательной ячейке этого элемента списка; если r указывает на последний элемент списка, возвращает специальное значение, которое обычно обозначается NULL:

SetPointerNext(r, t) - установить указатель на следующий элемент в элементе списка, на который указывает указатель r, равным t;

HasNext(r) - проверить, есть ли элементы в списке А после элемента, на который указывает указатель r (ДА, если есть; НЕТ, если текущий элемент – последний)

Упражнение 2. Написать алгоритм MoveNext(A, r1, r2), который удаляет из списка А элемент, следующий за элементом, на который указывает указатель r2 и вставляет этот элемент после элемента, на который указывает указатель r1.

Решение.

```
alg Move (список A, указатель r1, r2)
  r_del:= PointerNext(r2) // элемент, который нужно удалить
  rnew:=  PointerNext(r2) // элемент, на который он ссылается
  SetPointerNext(r2, rnew)// указатель в элементе r2 перенаправлен,
                       //  т.е. следующий за r2  элемент списка удален
  r_ins := PointerNext(r1)// элемент, перед которым будет вставлен
                          //     элемент r_del
  SetPointerNext(r1, r_del)// меняем ссылки в r_ins и r_del –
                           //   вставляем r_del после r_ins
  SetPointerNext(r_del, r_ins)
end_alg
```

Рис.2

Теперь мы можем написать алгоритм сортировки набора чисел, которые представлены односторонним списком, см. рис 3А (основной алгоритм) и рис. 3В, 3С (вспомогательные алгоритмы). В алгоритме на рис. 3С используется функция Move.

```
algo MergeSortList(C)
  k:=1
  while k<N
    // MergeStageList(C, k) – выполнение этапа сортировки,
    //     соответствующего блокам размера k
    MergeStageList(C, k)
    k:=2*k
  end_while
end_alg
```

Рис.3А Основной алгоритм сортировки списка слиянием фрагментов

```
alg MergeStageList(C, k)
// Этап сортировки разбиением на блоки и
слиянием,
// соответствующий блокам
// ДАНО: Список разбит на блоки по k чисел в
каждом
//      (в последнем блоке может быть меньше k
элементов).
//      Все блоки упорядочены
//      Главный указатель списка С указывает на
его заголовок
// НАДО: Список разбит на блоки по 2k чисел в
каждом
//      (в последнем блоке может быть меньше k
элементов).
//      Все блоки упорядочены
//      Главный указатель списка С указывает на
его заголовок
  while HasNext(C)
  assert Главный указатель указывает на элемент с
номером (2k)*t
         от начала списка (заголовок имеет № 0);
         t – количество предшествующих выполнений
цикла.
  assert  Каждый из блоков элементов списка с 1-го
по k-й;
          с (k+1)-го по 2k-й, …, с (2k*(t-1)+1)-го по 2kt-й
          упорядочен по неубыванию
    r1:= C
    // Установить указатель r2 на элемент,
предшествующий началу
    //  следующего k-блока
    m:=0
    while m<k and HasNext(C)
      ToNext(C)
      m:=m+1
    end_while
    if NOT(HasNext (C) then // нет второго k-блока
       toStart(C)
       break  // Обработка списка С закончена
    endif
    r2 := MainPointer(C)
    Merge(C, r1, r2, k)
  end_while
end_alg
```

Рис.3В. Реализация этапа сортировки слиянием (см.занятие 2, п.2.2)

```
alg Merge(C, r1, r2, k)
// r1, r2 – указатели на элементы списка  C
// указатель k находится на k элементов после
 указателя r1
Given: Фрагмент списка C длины k, начиная от Next(r1)
 упорядочен по
      возрастанию
      Фрагмент списка C, начиная от Next(r2), длины k
(или до конца списка)
      упорядочен по возрастанию
Needed: Фрагмент списка, начиная от Next(r1), длины 2k
(или до конца списка)
        упорядочен по возрастанию
      Главный указатель списка C указывает на
 последний элемент
        упорядоченного блока
 NA:=0 //количество обработанных элементов,
 начиная с Next(r2) (“список B”)
 NB:=0//количество «пройденных» элементов,начиная
 с Next(r2)(“список A»)
 curA := r1  // элемент ПЕРЕД тем элементом списка A,
 с которым нужно
             // сравнивать текущий элемент списка B
 curB := r2   //элемент ПЕРЕД обрабатываемым
 элементом списка B,
 while NB<k  // Если в блоке B просмотрено k
 элементов,
            //   прекращаем работу
    CurAPrev:= CurA
    CurA:= PointerNext(CurA)
    CurBPrev:= CurB
    CurB:= PointerNext(CurB)
    x:= Value(CurB)
    while NA<k and  x > Value(CurA)
      CurAPrev:= CurA
      CurA := PointerNext(CurA)
      NA:=NA+1
    end_while
    Move (C, CurAPrev, CurBPrev)
    NB:=NB+1;
    if NOT HasNext(CurB) then
      break
    endif
  end_while
  MainPointer(C):= PointerNext(CurBPrev) // указатель на
                           // последний элемент блока B
 end_alg
```
Рис.3C Слияние двух соседних k-блоков в списке

4. Бинарные деревья поиска.

4.1. Определения.

Замечание. Операция FIND как для массивов, так и для списков, требует просмотра всего списка (или массива). Если известно, что данные упорядочены, то это помогает, но все равно (в среднем) нужно просмотреть список (массив) до середины.

Новая структура – *сбалансированное бинарное дерево.*
Дерево (точнее – *бинарное дерево*) - это сетевая структура, каждая запись которой хранит две ссылки – на *левого сына* и на *правого сына.* Записи называются *узлами* дерева. У некоторых записей может не быть одного из сыновей или обоих сыновей (в последнем случае узел называется *листом*). Узел, который ни для кого не является сыном, называется *корнем* дерева, при этом (1) из корня есть путь в любой узел; (2) этот путь – единственный.

Поддерево (с корнем в узле V) – это множество всех узлов, в которые можно попасть из узда V, включая сам узел V. *Левое (правое) поддерево для узла W* – это поддерево с корнем в левом (правом) сыне узла W. Если у вершины W нет какого-то сыны, то говорят, что соответствующее дерево – *пустое.*

Высотой дерево называется наибольшее количество ссылок на пути от корня к листу. Высота дерева T обозначается H(T).

Пример. На рис.4 изображено бинарное дерево, узлы изображены кружками. В каждом кружке изображено число, которое хранится в этом узле. В корне дерева хранится число 8. Его левый сын – узел с числом 3, правый сын – узел с числом 10. Левое поддерево корня содержит узлы с числами 3, 1, 6, 4, 7; правое поддерево корня содержит узлы с числами 10, 14, 17. У записи с числом 14 нет правого сына, а у записей с числами 1, 4, 7, 13 вообще нет сыновей (это листья). Высота дерева – 3. Конец примера.

Бинарное дерево *поиска (БДП)* – это такое бинарное дерево, что
1) в каждом узле хранится число, причем все числа – разные;
2) в левом поддереве произвольного узла W все числа меньше, чем число в узле W; в правом поддереве произвольного узла W все числа больше, чем число в узле W.

Пример. Деревья на рис. 4 и 5 - это бинарные деревья поиска.

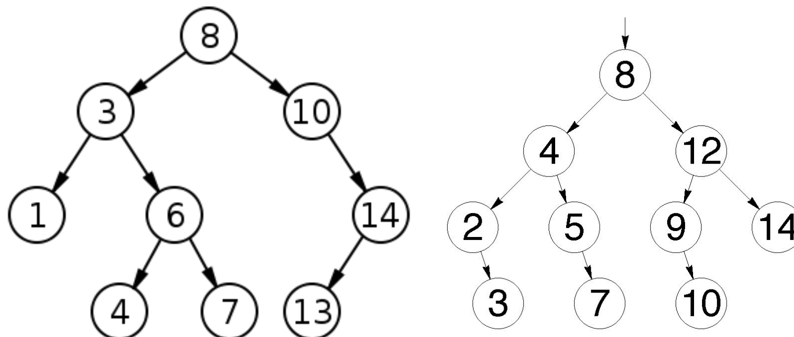

Рис.4 НЕ сбалансированное БДП   Рис.5. Сбалансированное БДП

Утверждение. Ответить на вопрос FIND(x), т.е. хранится ли в БДП T число x, можно за время ~ H(T).

Дерево называется *сбалансированным,* если для любого его узла W количество узлов в его левом и правом дереве отличается не более, чем на 1.

Пример. Дерево на рис.5 – сбалансированное, а на рис.4 – нет.

Утверждение. Если количество узлов в сбалансированном БДП равно N,то его высота не превосходит $\log_2(N)$

Доказательство – не приводим.

<u>4.2. Построение сбалансированного бинарного дерева.</u>
Если все элементы известны заранее, то строить легко.
Берем средний элемент, приписываем его к корневой вершине. Если количество элементов четное – берем больший из двух средних.
Далее точно так же строим левое и правое поддерево.

Если элементы поступают по одному в произвольном порядке, то придется на ходу перестраивать уже построенное дерево – делать балансировку. Это можно сделать за время ~ N, где N – текущее количество элементов в дереве. Про это не будем.
Вместо этого разберем другую структуру – дерево отрезков
Продолжение следует.

12.08 Занятие 4.
1. Повторение
2. Деревья отрезков как деревья поиска.

2.1. Полное дерево отрезков.

Определение. *Полное дерево отрезков высоты k* – это полное бинарное дерево (см. рис.1), узлам которого соответствуют отрезки, причем

1) корень дерева помечен отрезком $[1, 2^k]$;
2) если узел помечен отрезком $[a+1, a+2^r]$, где $r>0$, то у узла есть два сына, причем левый сын помечен отрезком $[a+1, a+2^{r-1}]$, а правый сын – отрезком $[a+2^{r-1}+1, a+2^r]$
3) если узел помечен отрезком [a, a], то этот узел не имеет сыновей (является *листом)*

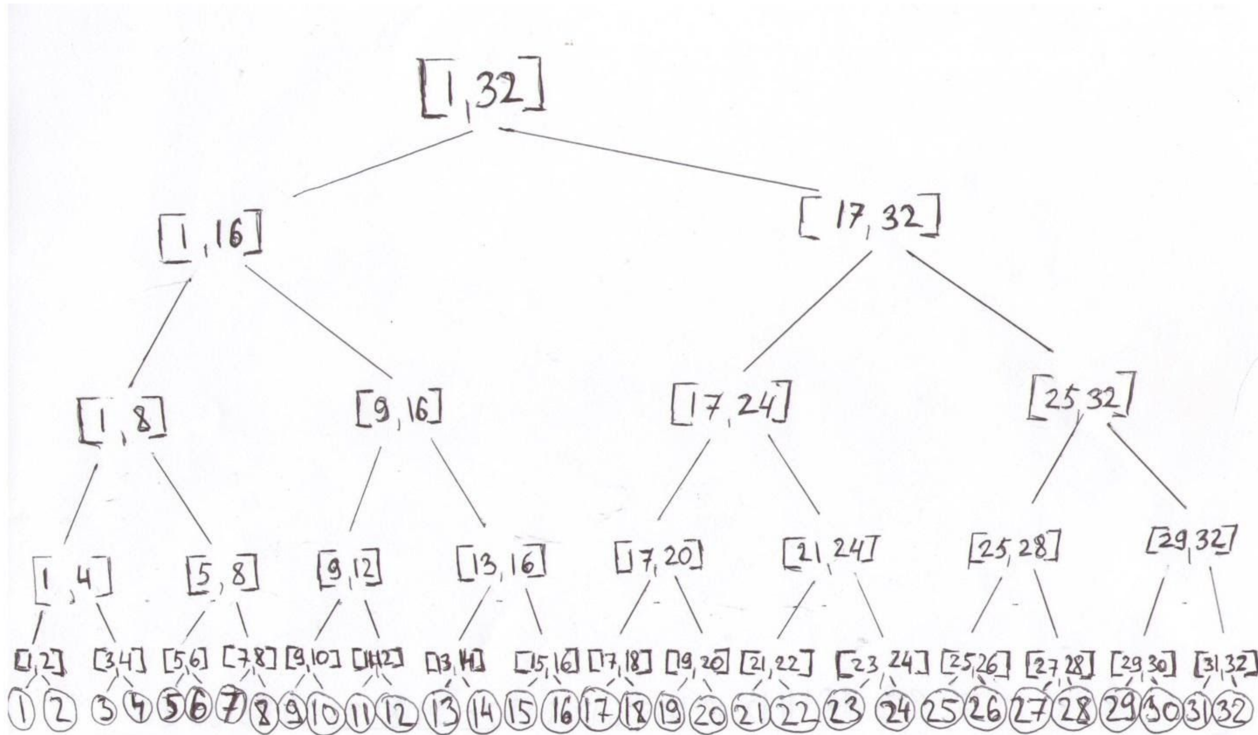

Рис.1. Полное дерево отрезков высоты 5.

Отрезки, которыми помечены узлы полного дерева отрезков, будем называть *бинарными* отрезками.

2.2. Дерево отрезков данного множества.

Пусть M – множество натуральных чисел, причем все элементы M не превосходят 2^k (k – некоторое натуральное число).

Дерево отрезков высоты k для множества M – это полное дерево высоты *k.* Из которого удалены все узлы, соответствующие отрезкам, в которых нет элементов множества *M* (см. рис.2).

Замечание. Так как число *s* такое, что $1\le s \le 2^k$, принадлежит k+1 бинарным отрезкам длины не более 2^k, то дерево отрезков высоты k для множества M содержит не более |M|*k узлов (|M| - количество узлов в множестве M).

2.3.Добавление и удаление узлов в дерево отрезков.

Утверждение. Пусть T - дерево отрезков высоты k для множества M; TIns - дерево отрезков высоты k для множества M+{x} (x не лежит в M); TDel - дерево отрезков высоты k для множества M-{z} (z лежит в M). Тогда как дерево Tins, так и дерево TDel могут быть получены из дерева T за время ~ k.

Доказательство (набросок).

(1) Удаление. Удаляем лист, соответствующий числу *z.* Далее двигаемся вверх по дереву T и

удаляем узлы, соответствующие отрезкам в которых нет элементов множества M, отличных от z.
(2) Вставка. Идем от корня вниз. Обнаружив уровень, на котором нет узла, помеченного отрезком, содержащим *x,* добавляем эту вершину. Далее добавляем вершину для числа *x* на каждом следующем уровне.
Конец доказательства.
Замечание. Если появилось число за пределами диапазона $[1.\ 2^k]$, то можно надстроить дерево. Время надстройки логартфмически зависит от того, насколько добавляемый элемент превосходит 2^k.

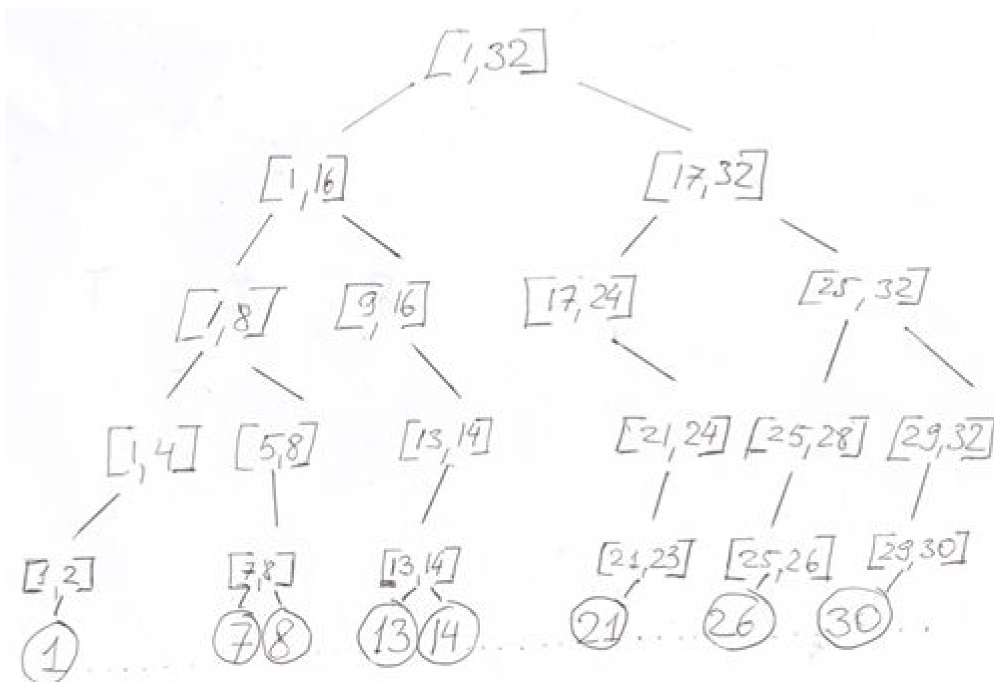

Рис.2. Дерево отрезков высоты 5 для множества {1, 7, 8, 13, 14, 21, 26, 30}

2.4.<u>Обогащенные деревья отрезков</u>.

Пусть M – множество натуральных чисел. Будем считать, что все элементы M принадлежат отрезку $[1, 2^k]$, где k – некоторое натуральное число.

2.4.1. Задача о сокровищах на дороге (сумма). Для каждого из элементов множества M задано целое число («вес мешка сокровищ под столбом»). Нужно уметь для каждого допустимого отрезка, лежащего внутри отрезка $[1, 2^k]$, назвать сумму весов сокровищ на этом отрезке.

Замечание. Количество возможных запросов ~ $(2^k)^2$.

Идея решения – использовать дерево отрезков высоты k для множества M с дополнительными пометками (*обогащенное* дерево отрезков высоты k для множества M). Это дерево будем обозначать *TSegmSum(M, k).* Исходное дерево отрезков высоты k для множества M будем обозначать *TSegm(M, k).*

Дерево *TSegmSum(M, k)* получается из дерева *TSegm(M, k)* дописыванием в каждом узле сумму весов сокровищ на соответствующем отрезке.

Замечание. Дерево *TSegmSum(M, k)* строится из дерева *TSegm(M, k)* движением от листьев к корню за время ~ количества узлов в дереве *TSegm(M, k).*

2.4.2. Решение задачи о сокровищах на дороге (сумма).

Пусть нужно узнать сумму сокровищ для отрезка [a, b]. Двигаясь по дереву *TSegmSum(M, k)* от корня к листьям строим разбиение [a, b] на непересекающиеся бинарные отрезки так, чтобы разбиение было минимальным, то есть так, что никакие два бинарных отрезка, входящих в разбиение, нельзя объединить в бОльший бинарный отрезок. Сопоставим исходный тестовый отрезок [a, b] корню дерева *TSegmSum(M, k)*. Далее будем переходить от узла дерева к его сыновьям. При этом тестовый отрезок будет «дробиться» на

более мелкие отрезки – по одному на каждому сына узла. А именно, тестовый отрезок узла-сына – это пересечение тестового отрезка узла-отца и бинарного отрезка узла-сына. Если оказалось, что тестовый отрезок узла-сына совпадает с бинарным отрезком этого узла, то бинарный отрезок включается в искомое разбиение. В противном случае к узлу сыну и его тестовому отрезку применяется та же процедура.

Утверждение. Если узел не является корнем, то не более одного из его сыновей получает тестовый отрезок, не совпадающий с бинарным отрезком этого узла.
Доказательство - разберем на примере отрезка [5. 19] и обогащенного дерева, изображенного на рис. 3.

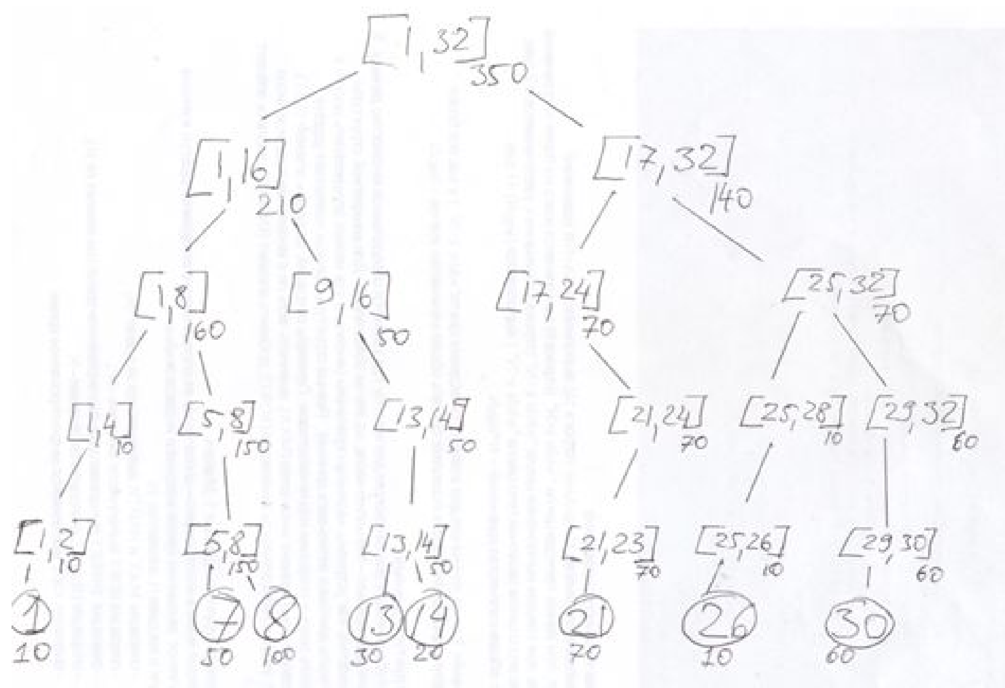

Рис.3. Обогащенное дерево для дерева отрезков, изображенного на рис.2.

Уровень 1. Узел: [1, 32]. Тестовый отрезок T(1, 32) = [5, 19]. Сыновья: [1, 16], [17. 32].
Новые тестовые отрезки: T(1, 16) = [5, 16]; T(17, 32) = [17, 19].
Текущее разбиение: [5, 16] + [17, 19]. Оба тестовые отрезка – не бинарные Пока искомых бинарных отрезков нет.
Текущая сумма для найденных бинарных отрезков S=0.
Уровень 2.
А. Узел: [1, 16]. Тестовый отрезок T(1, 16) = [5, 16]. Сыновья: [1, 8], [9. 16].
Новые тестовые отрезки: T(1, 16) = [5, 8]; **T(9, 16) = [9, 16]. Жирным выделен тестовый отрезок, совпадающий с бинарным.**
Б. Узел: [17, 32]. Тестовый отрезок T(17, 32) = [17, 19]. Сыновья: [17, 24], [25. 32].
Новые тестовые отрезки: T(17, 24) = [17, 19]; T(25, 32) – пустой.
Текущее разбиение: [5, 8] + ***[9, 16]*** + [17, 19]. Есть два небинарных отрезка (по краям) и один бинарный, входящий в искомое разбиение (выделен жирным).
Текущая сумма для найденных бинарных отрезков S= 0+ S(9, 16) = 0+50 = 50
Уровень 3.
А. Узел: [1, 8]. Тестовый отрезок T(1, 8) = [5, 8]. Сыновья: [1, 4], [5. 8].
Новые тестовые отрезки: T(1, 4) - пустой; **T(5, 8) = [5, 8].**
Б. Узел: [17, 24]. Тестовый отрезок T(17, 24) = [17, 19]. Сын: [21. 24] (на отрезке [17, 20] элементов множества M нет).
Новый тестовый отрезки: T(21, 24) – пустой.
Текущее разбиение: ***[5, 8]*** + **[9, 16]** . Небинарных отрезков нет – это и есть искомое разбиение.

Новый бинарный отрезок - ***[5, 8]*** (выделен курсивом).
Сумма для найденных бинарных отрезков S= 50+ S(5, 8) = 50+150 = 200
Ответ: 200.
Конец примера.

Следствие. В разбиении будет не более 2*k бинарных отрезков, найти эти отрезки можно за время ~k. Поэтому общее время получения ответа будет ~k.

2.4.3. Задача о сокровищах на дороге (максимум). Для каждого из заданных чисел задано целое число («вес мешка сокровищ под столбом»). Нужно уметь для каждого допустимого сегмента (=отрезка) назвать максимальное сокровище на этом отрезке.
Эта задача решается так же, как задача о сумме (см. п. 2.4.1-2.4.2) , но в узлах обогащенного дерева нужно писать не сумму весов сокровищ соответствующего бинарного отрезка, а вес максимального из этих сокровищ.

2.4.4. Обновление обогащенного дерева сегментов.
Утверждение. Пусть T = *TSegmSum(M, k)* - обогащенное дерево отрезков высоты k для множества M; TIns – обогащенное дерево отрезков высоты k для множества M+{x} (x не лежит в M); TDel – обогащенное дерево отрезков высоты k для множества M-{z} (z лежит в M). Тогда как дерево Tins, так и дерево TDel могут быть получены из дерева T за время ~ k.
Доказательство – аналогично доказательству утверждения из п. 2.3. Единственное отличие – нужно пересчитывать суммы в тех узлах, которые мы проходим.

с небольшой сложностью (допускаем пропуск уровней, если цепь – УТОЧНИТЬ)
2.4.5. Вывод
Обогащенные деревья отрезков позволяют для заданного множества M и натурального число k, такого, что все элементы M не превосходят 2^k , решать задачи ДОБАВИТЬ, УДАЛИТЬ, СУММАпоОТРЕЗКУ, МАКСИМУМпоОТРЕЗКУ, НАЙТИ за время ~k.
Если элементы множества M «примерно равномерно» распределены на участке [1, 2^k], то можно считать, что k ~ log(|M|).
Замечание. Если это не так, например, один из элементов множества M намного больше остальных, то времени ~ log(|M|) можно достичь за счет небольшого изменения деревьев отрезков (т.н. «сжатие цепей»).

3. Поиск точных вхождений

3.1. Поиск вхождений одного слова: постановка задачи

Дан длинный текст (геном) длины L и короткое слово (сайт) P длины k; предполагается, что k <<L (читается «k много меньше L»).
Требуется найти все вхождения сайта в геноме.
Замечание. Слово, которое мы ищем принято называть *паттерном* (от англ. *pattern* – образец).
Наивный метод сравнения — «прикладывать» паттерн к геному, начиная с каждой позиции. Приложив паттерн к геному, побуквенно проверяем, совпадают ли буквы в паттерне и соответствующем месте генома. Если нашли несовпадение, сдвигаем начальную позицию в геноме на 1 и повторяем сравнение.
Недостаток метода: если, скажем первые 5 букв просматриваемого участка генома совпадают с началом паттерна, а 6-я буква нет, то при сдвиге начала просмотра генома на 1 мы 4 буквы будем анализировать повторно. Это приводит к оценке времени работы алгоритма поиска ~L*k.
Упражнение. Для произвольных L и k придумайте текст длины не менее L и паттерн длиной k так, чтобы при поиске паттерна в тексте с помощью описанного наивного алгоритма выполнялось L*(k-1)/2 сравнений букв.
Хорошо бы избежать повторного просмотра уже просмотренных букв – то есть получить

алгоритм со временем работы ~L.

3.2. Поиск вхождений одного слова: идея быстрого алгоритма

Длинный текст (геном) будем обозначать G, а паттерн – P. Букву в k-й позиции генома (паттерна) будем обозначать G[k] (соответственно, P[k]). Позиции будем нумеровать, начиная с 1.

Для примеров будем использовать паттерн P=ACAGACAT.

Пусть мы «прошли» s позиций генома и паттерн приложен, начиная с позиции s+1, где $s \geq 0$.

То есть буква G[s+1] сравнивается с буквой P[1]= A,

Пример 1. Текущая буква генома сравнивается с 1-й буквой паттерна, т.е. никакой фрагмент генома, заканчивающийся в позиции s, не является началом паттерна. В этом случае быднм говорить «обнаружено пустое начало паттерна» или «обнаружен пустой *префикс* паттерна»

Если G[s+1] ≠ P[1]= A, то мы не смотрели лишних букв и можно просто сдвинуть начало анализируемого фрагмента генома. Следующую букву - букву G[s+2] - мы снова должны сравнивать с 1-й буквой паттерна P[1]= A. Если G[s+1] = P[1]= A, то букву G[s+2] мы должны сравнивать со 2-й буквой паттерна P[2]= C. Как видим, в этом случае повторного просмотра букв нет. Сказанное будем изображать так (см. рис.4).

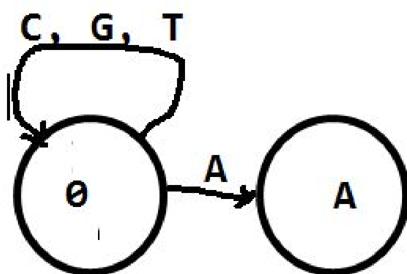

Рис.4

Каждый кружок обозначает, какой префикс в паттерне мы обнаружили (0 обозначает пустой префикс). Соответственно, следующая буква генома (в нашем случае) – буква G[s+2]) должна сравниваться с позицией в паттерне, которая следует за найденным префиксом. Стрелки, выходящие из кружка соответствуют возможным буквам на *текущей* позиции генома. Конец стрелки указывает префикс, который будет обнаружен после сравнения. В нашем примере после сравнению с буквами C.G,T у нас по-прежнему будет пустой паттерн, а после сравнения с буквой A мы обнаружим префикс длины 1.

Пример 2. Найден префикс длины 1 P[1:1] = A. Текущая буква генома сравнивается с 2-й буквой паттерна. Например, G[s+1] = P[1]= A и мы выполняем сравнение очередной буквы генома (это буква G[s+2]) с P[2].

Если G[s+2] = P[2]= C, то найденный префикс продлевается (теперь это префикс AC) и на следующем шаге мы продолжим сравнение с паттерном, т.е. будем сравнивать букву G[s+3] с буквой P[3]. Повторного анализа позиции G[s+2] нет.

Если G[s+2] = G или T, то ясно, что с позиции s+2 паттерн начинаться не может (паттерн начинается с A, а G[s+2] ≠ P[1]= A). Поэтому новое сравнение с 1-й позицией паттерна мы начнем с позиции s+3, при этом мы располагаем только пустым префиксом. Отметим, что мы снова избежали повторного анализа позиции s+2.

Пусть, наконец, G[s+2] = A≠ P[2]= C. В этом случае совпадение с паттерном нарушено и нам нужно сдвинуть начало исследуемого фрагмента генома в позицию s+2. Но мы уже знаем, что G[s+2]=A=P[1]! Поэтому можно сразу перейти к сравнению G[s+3] со ВТОРОЙ буквой паттерна P[2] – мы располагаем префиксом длины 1!. Схема сравнений изображена на рис. 5.

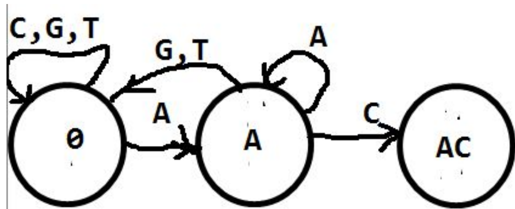

Рис.5

Как видим, во всех случаях, мы перешли к анализу позиции G[s+3], избежав повторного анализа позиции G[s+2]. При этом, с какой буквой паттерна сравнивается буква G[s+3], зависит от буквы G[s+2] (напомним, что мы рассматриваем случай G[s+1] =P[1] = A).
Если G[s+2] = G или T, то сравниваем букву G[s+3] с P[1]; если G[s+2] = A, то сравниваем G[s+3] с P[2]; если G[s+2] = C, то сравниваем G[s+3] с P[3].

Подобным образом можно рассмотреть случаи, когда текущая буква генома сравнивается с каждой из букв паттерна. Формальное правило выглядит так.
Пусть текущая буква генома G[t] = x сравнивается с (r+1)-й буквой паттерна. Это значит, что предшествующие r букв генома совпадают с r первыми буквами паттерна (у нас есть префиксP[1..r]). То есть в позиции t генома заканчивается слово v = P[1]…P[r]x. Пусть n – такое наибольшее число, что конец слова v длины n является началом (префиксом) паттерна (но не совпадает с паттерном целиком!). Это и будет новый текущий найденный префикс паттерна. Следующую букву генома – букву G[t+1] нужно сравнивать с (n+1)-й буквой паттерна – следующей буквой после найденного префикса.

Рассмотрим еще 2 примера.
Пример 3. Текущая буква генома G[t] = x сравнивается с 4-й буквой паттерна P[4] = G, то есть, найденный префикс – это префикс ACA (см. рис.6).

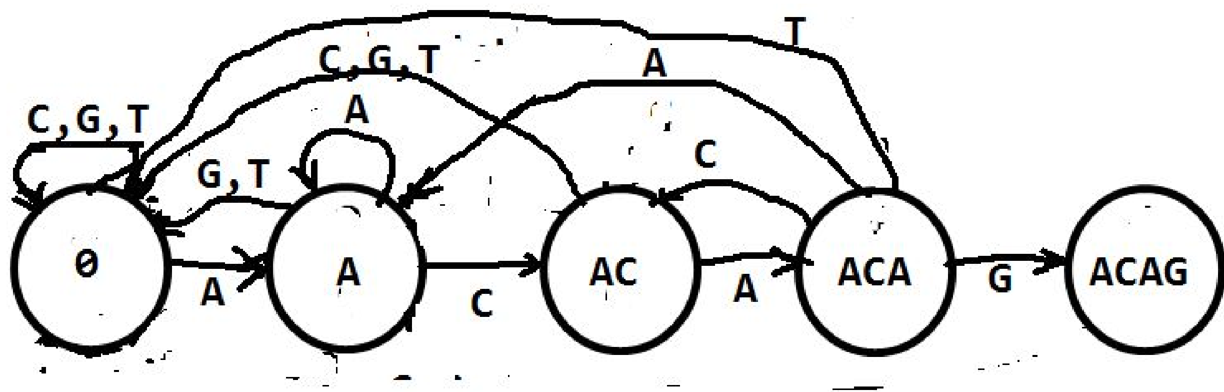

Рис.6

Если x = P[4] = G (в геноме найдено ACAG), то следующая буква генома G[t+1] сравнивается с P[5] (*n=4,* найденный префикс – ACAG, см. определение *n* в *формальном описании*).
Если x = P[4] = A (в геноме найдено ACAA), то *n=1,* найденный префикс – A и следующая буква генома G[t+1] сравнивается с P[2] =C.
Если x = P[4] = C (в геноме найдено ACAC), то *n=2, найденный префикс* – AC и следующая буква генома G[t+1] сравнивается с P[3] = A.
Если x = P[4] = T (в геноме найдено ACAT), то *n=0*, найденный префикс - пустой и следующая буква генома G[t+1] сравнивается с P[1] = A.

Пример 4. Текущая буква генома G[t] = x сравнивается с последней буквой паттерна P[8] = T, см. рис.7. Предшествующие 7 букв генома совпадают с соответствующими буквами

паттерна, то есть найденный префикс - это ACAGACA.

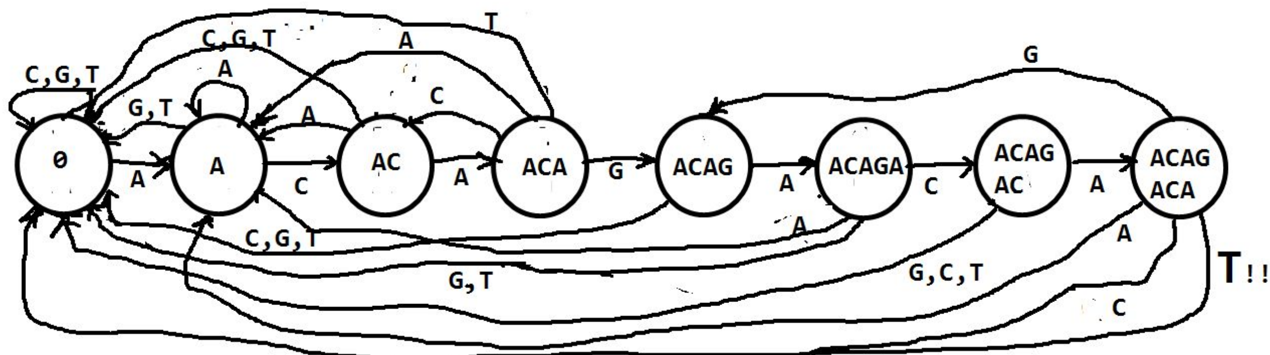

Рис.7

Если x = P[4] = T (в геноме найдено ACAGACAT), то найдено вхождение паттерна. Это обозначено восклицательными знаками на соответствующей стрелке, такую стрелку назовем *допускающей*. Следующая буква генома G[t+1] будет сравниваться с P[1] (*n=0,* см. определение *n* в *формальном описании*). Мы продолжаем поиск новых вхождений паттерна.
Если x = P[4] = A (в геноме найдено ACAGACAA), то *n=1* и следующая буква генома G[t+1] сравнивается с P[2] =C.
Если x = P[4] = C (в геноме найдено ACAGACAC), то *n=2* и следующая буква генома G[t+1] сравнивается с P[3] = A.
Если x = P[4] = G (в геноме найдено ACAG), то *n=4* и следующая буква генома G[t+1] сравнивается с P[5] = A.

Замечание. Рассмотрим паттерн ACAGACAA (он отличается от нашего паттерна тем, что последняя буква не T, а A). Для такого паттерна возможны перекрывающиеся вхождения. Например, в тексте F = ACAGACAACAGACAA есть 2 вхождения: F[1..8] и F[8..15]. Допускающая стрелка из 8-го узла с буквой A ведет в этом случае во второй узел (*n= 1,* см. определение *n* в *формальном описании*).

Замечание. Построенная нами конструкции известна в теории алгоритмов под названием *конечного автомата* (классическое определение немного отличается от нашего). Конечный автомат имеет конечный набор *состояний* (они у нас обозначены кружками). Состояния нашего автомата соответствуют началам (по-научному - *префиксам*) паттерна.
На вход автомата по очереди подаются буквы определенного текста (у нас это буквы генома). Получив букву, автомат переходит в новое состояние. Находясь в определенном состоянии и получив на вход определенную букву, автомат может выдать сигнал о том, что обнаружено «допустимое слово». Конечные автоматы используются во многих алгоритмах поиска паттернов

3.3. Паттерны, включающие более одного слова.
Часто сайты связывания в геноме бывают *вырожденными,* то есть связывание происходит в месте, где в геноме находится одна последовательность нуклеотидов (слово) из некоторого набора. Такой набор слов тоже называется *паттерном* (паттерны, которые рассматривались в пп. 3.1, 3.2 – это *простые* или *однословные* паттерны).
Описанная идея (построение конечного автомата, которому на вход по очереди подаются буквы генома) применима и для паттернов, которые содержат несколько слов.
В этом случае состояния соответствуют префиксам всех слов паттерна (если у нескольких слов паттерна есть общее начало, ему соответствует одно состояние).

Формальное правило построения автомата теперь выглядит так:
Рассмотрим слово w, которое является началом (префиксом) одного или нескольких слов паттерна. Пусть x – буква (в случае геномного алфавита – одна из букв A, C, G, T). Пусть

текущая буква генома $G[t] = x$ сравнивается с $(r+1)$-й буквой паттерна. Пусть n – такое наибольшее число, что конец слова wx длины n является началом одного из слов паттерна (но не совпадает с этим словом целиком!). Пусть z и есть это самое начало.
Тогда стрелка из состояния w, помеченная буквой x ведет в состояние z. Если при этом слово wx принадлежит паттерну, то стрелка является допускающей.

На рис.8 изображен автомат для поиска вхождений паттерна, состоящего из двух слов ACAGACAT и ACAACACA. Все стрелки, которые явно не показаны на рисунке, ведут в начальное состояние, которое соответствует пустому префиксу.

Замечание. Если убрать стрелки, ведущие назад, останется дерево. Это дерево называется *префиксным деревом*.

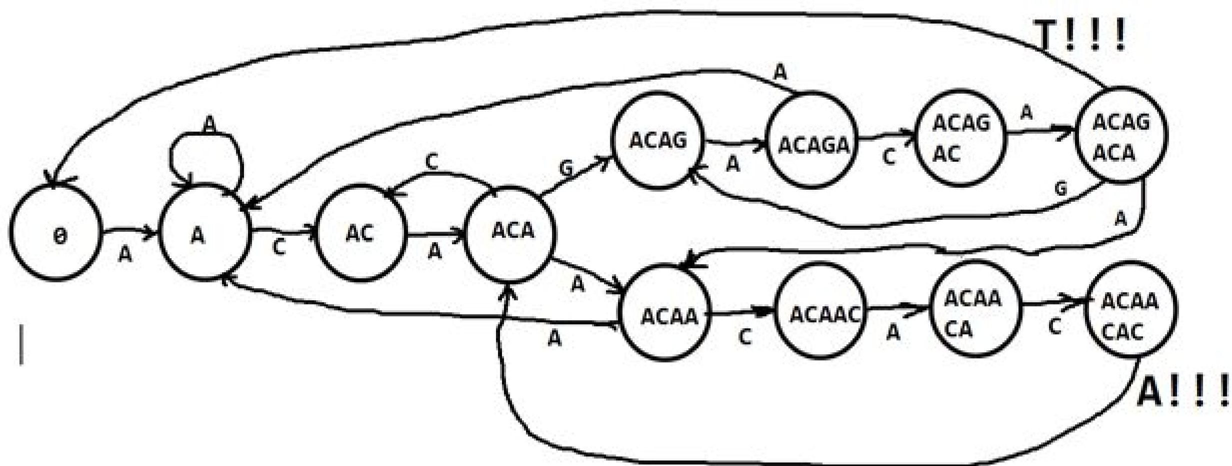

Рис.8. Автомат для поиска вхождений паттерна, состоящего из двух слов ACAGACAT и ACAACACA. Все стрелки, которые явно не показаны на рисунке, ведут в начальное состояние, которое соответствует пустому префиксу.

4. Подведение итогов.
За 4 занятия мы познакомились со следующими темами:
(Занятие 1)
- формальная запись алгоритма, время работы алгоритма,
- задача сортировки; наивный алгоритм сортировки; время его работы; потребность в улучшении

(Занятие 2)
- задача слияния массивов; алгоритм слияния массивов и алгоритм MergeSort с использованием массивов; количество сравнений в этих алгоритмах и общее время работы

(Занятие 3)
- списки; списки с доступом к концу; явное использование указателей в языках программирования;
- понятие структуры данных, допустимые операции для каждой из рассмотренных структур данных:
- алгоритмы сцепления списков и вставки списка в другой список;

(Занятие 4)
- задача поиска объекта (числа) в множестве; сбалансированные бинарные деревья поиска; время поиска при их использовании
- идея балансировки; примеры ее использования
- деревья отрезков (различные виды); время решения с их помощью задач НАЙТИ, ДОБАВИТЬ, УДАЛИТЬ, МАКСИМУМпоОТРЕЗКУ, СУММАпоОТРЕЗКУ;
- задача поиска всех вхождений паттерна в тексте (геноме) для случая простого паттерна (из одного слова) и паттерна, содержащего несколько слов; конечный автомат на основе префиксного дерева; решение задачи с помощью этого автомата.