**Комбинаторные алгоритмы, теоретический минимум для первой лабы**

Очередь – абстрактный тип данных с дисциплиной доступа к элементам «первый пришел – первый вышел». Добавление элемента возможно только в конец очереди (inqueue), а выборка – только с начала очереди (dequeue), при этом элемент из очереди удаляется.

Стек – структура данных с дисциплиной доступа к элементам «первый пришел – последний вышел». Добавление и выборка элемента возможны только с начала стека.

Дэк (double ended queue) - структура данных, имеющая две дисциплины обслуживания сразу – “LIFO” и “FIFO”. То есть, элементы можно добавлять и удалять как в начало, так и в конец дека.

Моделирование очереди при помощи двух стеков: стеки, составленные концами, образуют очереди. Перечисляя элементы одного стека вглубь, а другого – наружу, мы перечисляем элементы с первого до последнего. Добавление элемента в очередь сводится к помещению элемента в стек, принятый началом очереди. Выборка элемента сводится к выборке элемента со стека, принятого концом очереди, при этом если такой стек оказывается пустым, то в него необходимо скопировать все элементы из другого стека, изменив порядок на противоположный, после чего осуществить выборку элемента.

Односвязный список – структура данных, составленная из однотипных элементов, связанных последовательно при помощи указателей.

Двусвязный список – структура данных, составленная из однотипных элементов, каждый из которых содержит указатель на следующий и на предыдущий элемент.

Асимптотические обозначения:

f = O(g) – функция растет не быстрее чем другая функция (f<=g с точностью до константы). Найдется такая константа с>0 и такое число n0>0 что 0<=f(n)<=cg(n) при всех n>n0

f = Ω(g) – функция растет не медленнее, чем другая функция (f>=g с точностью до константы). Найдется такая константа с>0 и такое число n0>0 что 0<=cg(n)<=f(n) при всех n>n0

f = θ(g) – функция растет так же как g ( у функций одинаковая скорость роста). Найдется такая константа с1>0, c2>0 и такое число n0>0 что 0<=c1g(n)<=f(n)<=c2g(n) при всех n>n0

f = o(g) – функция растет медленнее чем другая функция (f<g с точностью до константы). Найдется такая константа с>0 и такое число n0>0 что 0<=f(n)<cg(n) при всех n>n0

f = ω(g) - функция растет быстрее, чем другая функция (f>g с точностью до константы). Найдется такая константа с>0 и такое число n0>0 что 0<=cg(n)<f(n) при всех n>n0

Вычисление чисел Фиббоначи:

**Экспоненциальный рекурсивный алгоритм:**

Fib(n)

If n ==0 return 0

If n ==1 return 1

Return Fib(n-1) + Fib(n-2)

**Полиномиальный алгоритм:**

Fib(n)

If n==0 return 0

Int[] fibs = int[0..n]

Fibs[0] = 0

Fibs[1] = 1

For i=2 to n

Fibs[i] = fibs[i-1] + fibs[i-2]

Return fibs[n]

**Более детальный анализ показывает, что можно понизить сложность алгоритма до O(n)**

Количество размещений с повторениями равно количеству размещений: если у нас есть n элементов, и мы будем искать все размещения по m, то за первый можем взять любой (n), за второй – все, кроме взятого за первый (n-1) и так далее до m (n – m + 1). Видно, что полученное произведение является факториалом n за исключением чисел 1..n-m. Таким образом, количество перестановок с повторениями равно n!/(n-m)!.

Количество перестановок без повторов, очевидно, равно n!.

Количество перестановок с повторами равно n!/(m1!+m2!+…+mk!), где m1,m2,…,mk – количество повторов того или иного элемента.

Количество размещений строится из количества уникальных сочетаний, умноженного на количество перестановок, таким образом, количество сочетаний без повторов можно выразить формулой n!/((n-m)!\*m!). С повторами: (n+m-1)!/((n-1)!\*(m!))

Метод рекуррентных соотношений – сведение решения задачи, касающейся n элементов к аналогичной, охватывающей меньшее число элементов.

Если последовательность задается некоторым рекуррентным соотношением, то для нее верно уравнение:

an = b1an-1 + b2an-2 + … + bkap – линейное рекуррентное отношение порядка p

*Максимальная сложность (временная)* соответствует случаю, когда выбранные входные данные порождают наиболее долгое выполнение алгоритма.

*Средняя сложность (временная)*соответствует случаю, когда алгоритм применяется к *произвольным данным*.

Максимальная сложность (пространственная) соответствует случаю, когда алгоритму приходится использовать наибольшоий объем памяти.

Средняя сложность (пространственная) соответствует случаю, когда алгоритм использует произвольный объем памяти.

**Алгоритмы:**

1. *Пузырьковая*

Количество сравнений: (n2-n)/2 всегда одно и то же (n-1 = количество итераций внешнего цикла и n/2 = среднее количество итераций внутреннего цикла)

Количество обменов: в лучшем случае = 0 (массив отсортирован), в среднем и худшем случаях – величина порядка n2 (в худшем случае = (n2-n)/2)

Устойчивость – устойчивая

Симметрическая группа – группа всех перестановок

Инверсия – пара элементов перестановки, такая что меньшее число стоит правее большего

Таблица инверсий – последовательность b1, b2, b3, … , bn где bj – количество элементов слева от j, которые больше j.

Для получения обратной перестановки необходимо поменять местами области определения и области значения.

Восстановление перестановки по таблице инверсий: необходимо обойти все элементы таблицы инверсий и проверить три условия:

1. Равен ли элемент нулю, если да, то соответствующее значение пишем в начало перестановки
2. Больше ли значение в таблице перестановки чем длина строки, если да, то пишем соответствующее значение в конец перестановки
3. В противном случае пишем соответствующий элемент в данную позицию строки (с учетом того что индексация начинается с нуля)

**Алгоритмы сортировки**

**Сортировка методом пузырька**

for i in reversed(range(n)):

for j in range(1,i+1):

if a[j-1]>a[j]:

a[j-1], a[j] = a[j], a[j-1]

**Сортировка вставками**

for i in range(1,n):

key = a[i]

j = i – 1

while j >= 0 and a[j] > key:

a[j+1]=a[j]

j = j – 1

a[j+1] = key

**Сортировка бинарными вставками**

for i in range(1,n):

if a[i-1]>a[i]:

left = 0

right = i – 1

repeat

mid = (left + right) div 2

if a[mid] > a[i]:

right = mid - 1

else:

left = mid + 1

until left > right

for j in reverse(range(left+1,i))

a[j] = a[j – 1]

a[left] = a[i]

**Сортировка выбором**

for i in range(0,n):

imin = i

for j in range(i+1,n):

if a[j]<a[imin]:

imin = j

t = a[j]

a[j] = a[imin]

a[imin] = a[j]

**Сортировка выбором из дерева**

Если у нас есть N элементов, то необходимо выполнить N-1 сравнений, чтобы найти максимальный элемент, затем – N/4 чтобы выделить предпоследний и так далее.