0.1.1 Определение алгоритма. Примеры простых алгоритмов: вычисление числа Фибоначчи, проверка числа на простоту, быстрое возведение в степень.

Алгоритм – это формально описанная вычислительная процедура, получающая исходные данные (input), называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход (output).

Эффективность алгоритма определяется:

- Временем работы,
- Объемом дополнительно используемой памяти
- Другими характеристики. Например, количеством операций сравнения или количеством обращений к диску.

Вычисление числа Фибоначчи

1, 1, 2, 3, 5, 8, 13, 21, 34…

```
// Рекурсивный алгоритм.
int Fib(int n)
{
	if(n==0 || n==1)
		return 1;
	return Fib(n-1) + Fib(n-2);
}
```

Формула Бине

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

```
// Нерекурсивный алгоритм.
int Fib(int n)
{
	if(n == 0)
		return 1;
	int prev = 1;
	int current = 1;
	for(int i=2; i<=n; ++i)
	{
		int temp = current;
		current += prev;
		prev = temp;
	}
		return current;
}
```

Время работы T(n) = O(n)– количество итераций в цикле.

Проверка числа на простоту

```
bool IsPrime( int n )
{
      if( n == 1 )
            return false;
      for(int i=2;i*i <=n; ++i)
            if( n % i == 0 )
                  return false;
      return true;
}
```

Время работы T(n) = O(n).
Объем доп. памяти M(n) = O(1).

Быстрое возведение в степень

```
function Power(value, pow: int): int
  int result = 1
  while (pow > 0)
    if pow mod 2 == 1
      result *= value
    value *= value
    pow /= 2;
  return result;
```

Пусть $m = (m_k m_{k-1} \dots m_1 m_0)_2$ — двоичное представление степени *n*. Тогда $n = m_k \cdot 2^k + m_{k-1} \cdot 2^{k-1} + \dots + m_1 \cdot 2 + m_0$, где $m_k = 1, m_i \in \{0,1\}$ и $x^n = x^{((\dots((m_k \cdot 2 + m_{k-1}) \cdot 2 + m_{k-2}) \cdot 2 + \dots) \cdot 2 + m_1) \cdot 2 + m_0}$.

T(n) = O(log n),
M(n) = O(1)

0.1.2. Асимптотические обозначения (O - нотации), работа с ними.

Опр Пусть f, g - функции N -> N. N = {1, 2, …}

Тогда f = **O(g)**, если ∃ C, N0: ∀ n ∈ N, n>=N0 -> f(n)<=C*g(n)

Утв f = **O(g)**<=> ∃ C: ∀ n ∈ N -> f(n)<=C*g(n)

Док-во

<= Очевидно

=> Пусть f(n)<=C*g(n), ∀ n>N0. Определим C' = max(C, f(1)/g(1), f(2)/g(2) …, f(N0)/g(N0)). Тогда

- ∀ n>N0 f(n)<=C*g(n)<=C'*g(n)
- ∀ n<=N0 C'>=f(n)/g(n) -> f(n) <= C'*g(n)

n^2 = O(n^3)

Опр Пусть f, g - функции N -> N. N = {1, 2, …}

Тогда f = **Ω(g)**, если ∃ C, N0: ∀ n ∈ N, n>=N0 -> f(n)>=C*g(n)

Утв f = **Ω(g)** <=> ∃ C: ∀ n ∈ N -> f(n)>=C*g(n)

Замечание f = **Ω(g)** **<=>** **g** = **O(f)**

n^3 = **Ω**(n^2)

Опр f = **Θ(g)**, если ∃ C1, C2, N0: ∀ n ∈ N, n>=N0 -> C1*g(n)<=f(n)<=C2*g(n)

Замечание f = **Θ(g)** **<=>** **f** = **O(g)** **и** **f** = **Ω(g)**

4n^2+7n = **Θ**(n^2)
0<=log2n<=n (n ∈ N)
n^2<=4n^2+7n<=100*n^2

n = **Θ**(n+log2n)
0<=log2n<=n (n ∈ N)
1/2(n+log2n)<=n<=n+log2n

0.1.3. Определение структуры данных, абстрактного типа данных (интерфейса).

Структура данных — это способ организации информации для более эффективного использования.

Формально АТД может быть определён как множество объектов, определяемое списком компонентов (операций, применимых к этим объектам, и их свойств). Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

Зачастую реализует некоторый АТД (интерфейс)
Типичные операции:

- добавление данных
- изменение данных
- удаление данных
- поиск данных

Примеры АТД

- Список
- Стек
- Очередь
- Ассоциативный массив
- Очередь с приоритетом

0.1.4. Массив. Линейный поиск. Бинарный поиск.

Массив – набор однотипных компонентов (элементов), расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу (индексам). Традиционно индексирование элементов массивов начинают с 0.

Размерность массива – количество индексов, необходимое для однозначного доступа к элементу массива.

Линейный поиск.

Задача. Проверить, есть ли заданный элемент в массиве.

Решение. Последовательно проверяем все элементы массива, пока не найдем заданный элемент, либо пока не закончится массив.

```
// Проверка наличия элемента.
bool HasElement( const double* arr, int count, double element )
{
	for( int i = 0; i < count; ++i )
		if( arr[i] == element )
			return true;
	return false;
}
```

Время работы в худшем случае T(n) = O(n), где n – количество элементов в массиве.

Бинарный поиск.

Определение. Упорядоченный по возрастанию массив – массив A, элементы которого сравнимы, и для любых индексов k и l, k < l: A[k] ≤ A[l]

Задача. Проверить, есть ли заданный элемент в упорядоченном массиве. Если он есть, вернуть позицию его первого вхождения. Если его нет, вернуть -1.

Решение. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения. Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

```
// Бинарный поиск без рекурсии.
int BinSearch( double* arr, int count, double element )
{
	int first = 0;
	int last = count;
	while( first < last ) {
		int mid = ( first + last ) / 2;
		if( element <= arr[mid] )
			last = mid;
		else
			first = mid + 1;
	}
	return (first == count || arr[first] != element ) ? -1 : first;
}
```

```
// Возвращает позицию вставки элемента на отрезке [first, last).
int FindInsertionPoint( const double* arr, int first,
int last, double element )
{
	if(last - first == 1)
		return element <= arr[first] ? first : last;
	int mid = ( first + last ) / 2;
	if(element <= arr[mid])
		return FindInsertionPoint( arr, first, mid, element );
	else
		return FindInsertionPoint( arr, mid, last, element );
}

// Возвращает позицию элемента в упорядоченном массиве, если он есть.
// Возвращает -1, если его нет.
int BinarySearch( const double* arr, int count, double element )
{
	if(count == 0) return -1;
	int point = FindInsertionPoint( arr, 0, count, element );
	return ( point == count || arr[point] != element ) ? -1 : point;
}
```

Время работы T(n) = O(logn), где n – количество элементов в массиве.
Объем дополнительной памяти:
В нерекурсивном алгоритме M(n)=O(1).
В рекурсивном алгоритме M(n)=O(logn) , так как максимальная глубина рекурсии – log n.

Базовые структуры данных

1.1.1. Динамический массив.

1.1.2. Амортизационный анализ. Амортизированное (учетное) время добавления элемента в динамический массив.

1.1.3. Двусвязный и односвязный список. Операции. Объединение списков.

1.1.4. Стек.

1.1.5. Очередь.

1.1.6. Дек.

1.1.7. Хранение стека, очереди и дека в массиве. Циклическая очередь в массиве.

1.1.8. Хранение стека, очереди и дека в списке.

1.1.9. Поддержка минимума в стеке.

1.1.10. Представление очереди в виде двух стеков. Время извлечения элемента.

1.1.11. Поддержка минимума в очереди.

1.1.12. Двоичная куча. АТД "Очередь с приоритетом".

Тема 2. Сортировки и порядковые статистики.

3 лекции.

- **Формулировка задачи. Устойчивость, локальность.**
- **Квадратичные сортировки: сортировка вставками, выбором.**
- **Сортировка слиянием.**
- **Сортировка с помощью кучи.**
- **Слияние K отсортированных массивов с помощью кучи.**
- **Нижняя оценка времени работы для сортировок сравнением.**
- **Быстрая сортировка. Выбор опорного элемента. Доказательство среднего времени работы.**
- **Сортировка подсчетом. Карманная сортировка.**
- **Поразрядная сортировка.**
- **MSD, LSD. Сортировка строк.**
- **Поиск k-ой порядковой статистики методом QuickSelect.**
- **Поиск k-ой порядковой статистики за линейное время.**

Тема 3. Деревья поиска.

4 лекции.

- **Определение дерева, дерева с корнем. Высота дерева, родительские, дочерние узлы, листья. Количество ребер.**

- **Обходы в глубину. pre-order, post-order и in-order для бинарных деревьев.**
- **Обход в ширину.**
- **Дерево поиска.**
- **Поиск ключа, вставка, удаление.**
- **Необходимость балансировки. Три типа самобалансирующихся деревьев.**
- **Декартово дерево. Оценка средней высоты декартового дерева при случайных приоритетах (без доказательства).**
- **Построение за O(n), если ключи упорядочены.**
- **Основные операции над декартовым деревом.**
- **АВЛ-дерево. Вращения.**
- **Оценка высоты АВЛ-дерева.**
- **Операции вставки и удаления в АВЛ-дереве.**
- **Красно-черное дерево.**
- **Оценка высоты красно-черного дерева.**
- **Операции вставки и удаления в красно-черном дереве.**
- **Сплей-дерево. Операция Splay.**
- **Поиск, вставка, удаление в сплей-дереве.**
- **Учетная оценка операций в сплей-дереве = O(log n) без доказательства.**
- **B-деревья.**

Тема 4. Хеш-таблицы.

2 лекции.

- **Хеш-функции. Остаток от деления, мультипликативная.**
- **Деление многочленов - CRC.**
- **Обзор криптографических хеш-функций. CRC*, MD*, SHA*.**
- **Полиномиальная. Ее использование для строк. Метод Горнера для уменьшения количества операций умножения при ее вычислении.**
- **Хеш-таблицы. Понятие коллизии.**
- **Метод цепочек (открытое хеширование).**

- **Метод прямой адресации (закрытое хеширование).**
- **Линейное пробирование. Проблема кластеризации.**
- **Квадратичное пробирование.**
- **Двойное хеширование.**

Примечание. Остальные темы по хешам перенесены в 3 семестр, т.к. требуют знания теории вероятностей.

Тема 5. Жадные алгоритмы и Динамическое программирование.

1 лекция.

- **Общая идея жадных алгоритмов.**
- **Задача о рюкзаке.**
- **Общая идея последовательного вычисления зависимых величин. Идея введения подзадач (декомпозиции) для решения поставленной задачи. Восходящее ДП. Нисходящее ДП, кэширование результатов.**
- **Вычисление чисел Фибоначчи. Нахождение количества последовательностей нулей и единиц длины *n*, в которых не встречаются две идущие подряд единицы.**
- **Нахождение наибольшей возрастающей подпоследовательности за $O(N^2)$ и за $O(N \log N)$.**
- **Количество способов разложить число N на слагаемые.**
- **Количество способов разложить число N на различные слагаемые.**
- **Нахождение наибольшей общей подпоследовательности.**
- **Методы восстановления ответа в задачах динамического программирования.**
- **Расстояние Левенштейна.**