Part I
Роль алгоритмов в вычислениях

1 Алгоритмы

Алгоритм - любая корректно определенная вычислительная процедура на вход которой подается некоторый набор величин и результатом выполнения которой является выходной набор значений
Алгоритм корректен, если для каждого ввода результатом его работы является корректный вывод

2 Структуры данных

Структура данных - способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию

3 Сложные задачи

Задачи, для которых неизвестны эффективные методы решения называются **NP-полные**
Они представляют интерес:

1. Не доказано, что эффективного алгоритма не существует
2. Если эффективный алгоритм существует хотя бы для одной NP-полной задачи, то его можно сформулировать и для остальных
3. Некоторые NP-полные задачи похожи на задачи, для которых известны эффективные алгоритмы

4 Алгоритмы как технология

4.1 Эффективность

Алгоритмы часто различаются по эффективности, различия могут быть значительнее, чем те, что вызваны разной мощностью аппаратного обеспечения
Пример:
Сортировка вставкой: требует время, которое оценивается как $c_1 n^2$, где c_1 - константа, не зависящая от n. Таким образом, время работы этого алгоритма пропорционально n^2.
Сортировка слиянием: требует время, приблизительно равное $c_2 nlgn$.
Для этих двух методов зависимые множители относятся как $\frac{lgn}{n}$. Для маленьких n сортировка вставкой работает быстрее, но для больших n проявляется преимущество сортировки слиянием.

Part II
Приступаем к изучению

5 Сортировка вставкой

Вход: Последовательность из n чисел $< a_1, a_2, \cdots, a_n >$
Выход: Перестановка $< a'_1, a'_2, \cdots, a'_n >$ входной последовательности таким образом, что $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
Сортируемые числа известны под названием *ключи (keys)*
Псевдокод:

```
Insertion_Sort(A):
    for j = 2 to length[A]:
        key = A[j]
        # Insert A[j] into sorted sequence A[1:j-1]
        i = j - 1
        while i > 0 and A[i] > key:
            do A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

5.1 Инварианты цикла и корректность сортировки вставкой

В начале каждой итерации цикла for подмассив A[1:j-1] содержит те элементы, которые были в нем с самого начала, но расположенные в отсортированном порядке. Это свойство элементов A[1:j-1] называется **инвариантом цикла**
Инварианты цикла позволяют понять, что алгоритм работает корректно. Необходимо показать, что инварианты циклов обладают следующими тремя свойствами:

1. **Инициализация:** Они справедливы перед первой инициализацией цикла
2. **Сохранение:** Если они истинны перед очередной итерацией цикла, то остаются истинны и после нее
3. **Завершение:** По заврщении цикла инварианты позволяют убедиться в правильности алгоритма

Если выполняются первые два свойства, то инварианты **остаются истинными перед каждой очередной итерацией цикла**
Доказательства
Инициализация:
j = 2 → подмножество элементов A[1:j-1] состоит из одного элемента A[1], сохраняющего исходное значение. Более того, в этом подмножестве элементы рассортированы. Инвариант соблюдается
Сохранение:
В теле внешного цикла происходит сдвиг элементов A[j - 1], A[j - 2], A[j - 3], ... на одну позицию вправо до тех пор, пока не освободится подходящее место для элемента A[j]
Завершение:

При сортировке методом включений внешний цикл завершается, когда j превышает n, т.е. когда j = n+1. Подставим в формулировку инварианта цикла значение n+1, получим утверждение: в подмножестве элементов A[1:n] находятся те же элементы, что и были в нем до начала работы алгоритма, но расположенные в отсортированном порядке. Подмножество A[1:n] и есть массив A. Таким образом, весь массив отсортирован.

6 Анализ алгоритмов

Анализ алгоритма заключается в том, чтобы предсказать требуемые для его выполнения ресурсы. Чаще всего определяется время выполнения.
Путем анализа нескольких алгоритмов можно выбрать наиболее эффективный
Модель ресурсов:
В качестве технологии реализации принята модель обобщенной однопроцессороной машины с *памятью с произвольным доступом* (RAM). Команды процессора выполняются последовательно, одновременно выполняемые операции отсутствуют.
Команды процессора:

1. Арифметические операции
2. Операции перемещения данных
3. Управляющие (условное и безусловное ветвление, вызов подпрограммы, возврат из нее)

Для выполнения каждной такой инструкции требуется определенный фиксированный промежуток времени
Типы данных:

1. Целочисленный тип данных
2. Тип чисел с плавающей точкой

Существует верхний предел размера слова данных
Например, если обрабатываются входные данные с максимальным значением n, обычно предполагается, что целые числа представлены $clgn$ битами, где c - произвольная константа $\infty \geq c \geq 1$.
Вычисление $2k$ рассматривается как элементарная операция, если k достаточно малое число

6.1 Анализ алгоритма Insertion Sort

Для анализа необходимо использовать **размер входных данных**. Для некоторых задач это может быть *количество входных элементов*, для других - *общее количество бит*
Время работы измеряется в количестве элементарных операций, которые необходимо выполнить.
Предположим, что для выполнения строки псевдкода требуется фиксированное время. Одна и та же строка i выполняется за время c_i
t_j - количество проверок условия в цикле while. При нормальной работе циклов условие проверяется на один раз больше, чем выполняется тело цикла

```
Insertion_Sort(A):
    for j = 2 to length[A]                    #c1n
        key = A[j]                            #c2(n-1)
        i = j - 1                             #c3(n-1)
        while i > 0 and A[i] > key:           #c4(sum_{j=2}^n t_j)
            A[i + 1] = A[i]                   #c5(sum_{j=2}^n (t_j - 1))
            i = i - 1                         #c6(sum_{j=2}^n (t_j - 1))
        A[i + 1] = key                        #c7(n-1)
```

Время работы алгоритма - это сумма времени промежутков времени, необходимых для выполнения каждой входящей в его состав исполняемой инструкции

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Если массив уже отсортирован, то $t_j = 1$
Если массив отсортирован в обратном порядке, то это наихудший случай. Каждый элемент необходимо сравнивать со всеми элементами уже отсортированного множества. Так что $\forall j : t_j = j$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1)$$

$$T(n) = (\frac{c_4 + c_5 + c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4 + c_5 + c_6}{2} + c_7)n - (c_2 + c_3 + c_5 + c_7)$$

Это время работы можно записать как $an^2 + bn + c$, где константы зависят от c_i. Таким образом, это квадратичная функция n^2

6.2 Наихудшее и среднее время работы

В основном уделяется внимание **наихудшему времени работы**

1. Наихудший случай - верхний предел
2. В некоторых алгоритмах наихудший случай встречается довольно часто
3. Характер поведения усредненного времени работы часто ничем не лучше поведения для наихудшего случая

6.3 Порядок возрастания

Во внимание будет приниматься **только главный член формулы**. **Постоянные множители будут игнорироваться**, так как для оценки вычислительной эффективности алгоритма с большими входными данными они менее важны, чем порядок роста
Обычно один алгоритм считается **эффективнее** другого, если **время его работы в наихудшем случае имеет более низкий порядок роста**

Part III
Разработка алгоритмов

В алгоритме, работающем по методу вставок, применяется **инкрементный подход**: располагая отсортированным подмассивом A[1:j-1], мы помещаем очередной элемент A[j] туда, где он должен находится, в результате чего получаем отсортированный подмассив A[1:j]

7 Метод декомпозиции

Сложная задача разбивается на несколько более простых, которые подобн-ы исходной задаче, но имеют меньший объем. Далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получить решение исходной задачи
Разделяй и властвуй

1. **Разделение** задачи на несколько подзадач
2. **Покорение** - рекурсивное решение этих подзадач. Когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно
3. **Комбинирование** решений исходной задачи из решений вспомогательной задачи

7.1 Алгоритм сортировки слиянием

1. **Разделение:** Сортируемая последовательность разбивается на две меньшие последовательности, каждая из которых содержит n/2 элементов
2. **Покорение:** Сортировка обеих вспомогательных последовательностей методом слияния
3. **Комбинирование:** Слияние двух отсортированных последовательностей для получения конечного результата

Рекурсия достигает своего нижнего пределга, когда длина сортируемой последовательности становится равно 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной.
Основная операция, которая производится в процессе сортировки слияниям - это **объединение двух отсортированных последовательностей**.

Это делается с помощью вспомогательной функции Merge(A, p, q, r), где A - массив, p, q, r - индексы, такие что $p \leq q < r$.
В этой процедуре предполагается, что элементы подмассивов A[p:q] и A[q + 1:r] упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы массива A[p:r]
Для выполнения процедуры Merge требуется время $\Theta(n)$, где n = r - p + 1.
Аналогия с картами:
Из двух младших карт выбрать наиболее младшую, извлечь ее из соответствующей стопки и поместить в выходную стопку. Этот шаг повторяется до тех пор, пока в одной стопке не окажется 0 карт. После этого все оставшиеся в другой стопке карты необходимо поместить в выходную стопку.
Дополнительная идея: Используем **сигнальную карту** В ходе каждого шага не приходится проверять является ли каждая из двух стопок пустой. Не существует карт, достоинство которых больше сигнальной карты. Процесс продолжается до тех пор, пока карты в обеих стопках не окажутся сигнальными. В выходной стопке должна содержаться r - p + 1 карта → на этом значении **процесс можно остановить**
Псевдокод

```
Merge(A, p, q, r):
    n1 = q - p + 1                          #Calculate len A[p:q]
    n2 = r - q                              #Calculate len A[q+1:r]
    # Create arrays L[1:n1+1], R[1:n2+1]
    for i=1 to n1:                          #Copy A elements to L, R
        L[i] = A[p + i - 1]
    for j = 1 to n2:
        R[j] = A[q + j]
    L[n1 + 1] = ∞                  # Add signal values
    R[n2 + 1] = ∞
    i = 1
    j = 1
    fpr k = p to r:
        if L[i] ≤ R[j]:
            A[k] = L[i]
            i = i + 1
        else:
            A[k] = R[j]
            j = j + 1
```

Инвариант цикла

1. **Инициализация:** Перед первой итерацией цикла k = p, поэтому подмассив A[p:k-1] пуст. Он содержит k-p = 0 наименьших элементов массивов L, R. Поскольку i=j=1, элементы L[i], R[j] - наименьшие элементы массивов L и R, не скопированные обратно в массив A

2. **Сохранение:** Предположим, что L[i] ≤ R[j]. Тогда L[i] наименьший элемент еще не скопированный в массив A. Поскольку в массиве A[p:k-1] содержится k-p наименьших элементов, после копирования L[i] в A[k] в подмассиве A[p:k] будет содержаться k-p+1 наименьших элементов. В результате увеличения k в цикле for и i инварианта цикла сохраняется. Аналогично для L[i] ≥ R[j]

3. **Завершение:** Алгоритм завершается, когда $k = r + 1$. В соответствии с инвариантом цикла подмассив A[p:k-1], т.е. массив A[p:r] содержит k - p = r - p + 1 наименьших элементов массивов L[1:n1+1], R[1:n2+1] в отсортированном порядке. Все они, кроме сигнальных, скопированны в исходный массив.

Merge Sort(A, p, r):
Производит сортировку элементов в подмассиве A[p:r]. Если справедливо неравентсво $p \geq r$, в массиве содержится не больше 1 элемента и он является **уже отсортированным**
Псевдокод

```
Merge_Sort(A, p, q):
    if p < r:
        q = \lfloor (p + r)/2 \rfloor
        Merge_Sort(A, p, q)
        Merge_Sort(A, q + 1, r)
        Merge(A, p, q, r)
```

7.2 Анализ алгоритмов, построенных на принципе "разделяй и властвуй"

Время работы можно описать с помощью **рекуррентного уравнения**, которое выражает время решения задачи n через решения задач для меньших входных данных
Обозначим время решения задачи через $T(n)$

$$T(n) = \Theta(1), n \leq c$$

c - некоторая заранее известная константа
Если задача делится на a подзадач, объем каждой из которых равен 1/b от исходной задачи:
Для решения подзадачи потребуется время $T(n/b)$, для решения a задач: aT(n/b)
Предположим, что на разбитие задачи расходуется время $D(n)$, на переход к исходной задаче $C(n)$
Тогда:

$$T(n) = \begin{cases} \Theta(1), c \leq n \\ aT(n/b) + D(n) + C(n), c > n \end{cases}$$

7.3 Анализ алгоритма Merge Sort

1. **Разделение**: Определяет, где находится середина подмассива $D(n) = \Theta(1)$
2. **Властвование**: Рекуррентно решаются 2 подзадачи размер которых составляет 1/2 от исходной. $2T(n/2)$
3. **Комбинирование**: Процедура Merge. Работает за $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1), c = 1 \\ 2T(n/2) + \Theta(1) + \Theta(n) = 2T(n/2) + cn, n > 1 \end{cases}$$

Полностью раскрытое дерево имеет $lgn + 1$ уровень, а вклад каждого уровня - cn $\rightarrow$ cnlgn - cn $\rightarrow \Theta(nlgn)$