Занятие №3-4 Задача №1

Описание задания

Составить программу, которая для заданных значений $F(1), F(2), \ldots, F(d) F(1), F(2), \ldots, F(d)$ и числа d (где $2 < d \leq 10$) вычисляет n-й член последовательности $F(n)$ при условии $n > d$. Последовательность задаётся рекуррентным соотношением:

$$F(n) = F(n-1) + F(n-2) + \cdots + F(n-d). F(n) = F(n-1) + F(n-2) + \cdots + F(n-d)$$

Необходимо также проанализировать сложность предлагаемого решения и указать, как она изменяется по сравнению с «классическим» случаем, когда $d = 2$.

Вариант 1. Наивная рекурсия

Описание:

Функция вычисляет F(n) рекурсивно, вызывая себя для каждого из предыдущих d элементов. Если n не превышает d, возвращается соответствующее базовое значение.

Пошаговое выполнение для $n = 7$, $d = 3$, $base = [1, 1, 2]$:

- базовые значения заданы как $base = [1, 1, 2]$, то есть
  - $F(1) = 1$,
  - $F(2) = 1$,
  - $F(3) = 2$.

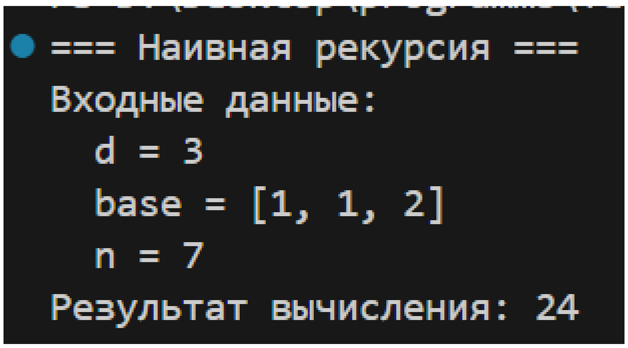
1. **Вызов:**
   Вызываем функцию с "$n = 7$", "$d = 3$" и "$base = [1, 1, 2]$".
   Так как "$7 > 3$", функция переходит к рекурсии.
2. **Вычисление $F(7)$:**
   Функция выполняет:
   "$F(7) = F(6) + F(5) + F(4)$".
3. **Вычисление $F(6)$:**
   Для $F(6)$ выполняется:
   "$F(6) = F(5) + F(4) + F(3)$".
   При этом "$F(3)$" – базовый случай, равный "2".
4. **Вычисление $F(5)$:**
   Аналогично, "$F(5) = F(4) + F(3) + F(2)$".
   Здесь "$F(3) = 2$" и "$F(2) = 1$" – базовые значения.
5. **Вычисление $F(4)$:**
   "$F(4) = F(3) + F(2) + F(1)$",
   где "$F(3) = 2$", "$F(2) = 1$" и "$F(1) = 1$".
   Таким образом, "$F(4) = 2 + 1 + 1 = 4$".
6. **Подстановка базовых случаев:**
   Теперь можно вычислить:
   - "$F(5) = F(4) + F(3) + F(2) = 4 + 2 + 1 = 7$",
   - "$F(6) = F(5) + F(4) + F(3) = 7 + 4 + 2 = 13$".
7. **Итоговое вычисление:**
   "$F(7) = F(6) + F(5) + F(4) = 13 + 7 + 4 = 24$".

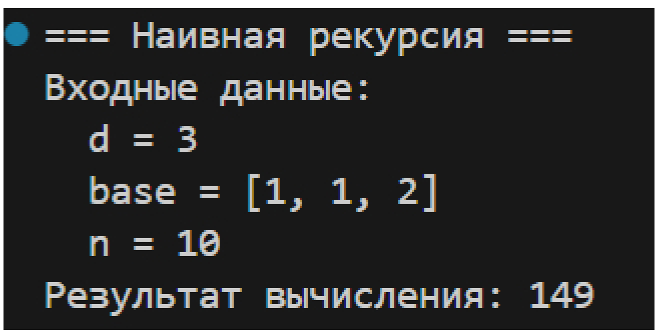
Код на Python

```
def F_naive(n, d, base):
    """
    Вычисляет n-й член последовательности наивной рекурсией.

    Параметры:
      n    - искомый номер элемента (n > d)
      d    - параметр рекурсии (количество суммируемых предыдущих элементов)
      base - список базовых значений [F(1), F(2), ..., F(d)]

    Сложность:
      Время: экспоненциальное ~ O(d^(n-d))
      Память: O(n) (глубина рекурсии)
    """
    if n <= d:
        return base[n - 1]  # базовые случаи
    result = 0
    for i in range(1, d + 1):
```

```python
            result += F_naive(n - i, d, base)
    return result

# Пример использования:
if __name__ == "__main__":
    d = 3
    base = [1, 1, 2]  # Заданные базовые значения
    n = 7

    # Сначала выводим все входные данные
    print("=== Наивная рекурсия ===")
    print("Входные данные:")
    print("  d =", d)
    print("  base =", base)
    print("  n =", n)

    # Вычисляем и выводим результат
    result = F_naive(n, d, base)
    print("Результат вычисления:", result)
```

Скриншоты выполнения кода





Временная сложность:

При использовании наивой рекурсии функция для каждого $n > d$ вызывает d рекурсивных вызовов, что формирует дерево рекурсии, где на каждом уровне число вызовов умножается на d. Глубина этого дерева приблизительно равна "$n - d$", поэтому общее число вызовов оценивается как "$O(d^{(n-d)})$". Это означает, что количество операций растёт экспоненциально с увеличением n, например, для $d = 3$ временная сложность записывается как "$O(3^{(n-3)})$".

Сложность по памяти:

Память в этом варианте расходуется в первую очередь за счёт стека вызовов рекурсии. Глубина рекурсии составляет приблизительно "$n - d$" уровней, что асимптотически оценивается как "$O(n)$". Таким образом, память используется пропорционально значению n.

Плюсы:

- Реализация очень проста и интуитивно понятна.
- Не требуется дополнительная структура данных для хранения результатов.

Минусы:

- **Низкая эффективность:** Из-за отсутствия кэширования функция выполняет множество повторных вычислений.
- **Экспоненциальный рост вызовов:** При увеличении "n" количество вызовов растёт как "$O(d^{(n-d)})$", что делает решение непригодным для больших значений n.
- **Переполнение стека:** Глубина рекурсии может привести к проблемам с памятью при больших значениях n.

Вариант 2. Рекурсия с мемоизацией (Top-Down Dynamic Programming)

Описание:

Аналогичен наивой рекурсии, но с сохранением (кэшированием) уже вычисленных значений в структуре "$memo$". Это позволяет избежать повторного вычисления одних и тех же значений.

Пошаговое выполнение для $n = 7$, $d = 3$, $base = [1, 1, 2]$:

- базовые значения заданы как $base = [1, 1, 2]$, то есть
  - $F(1) = 1$,
  - $F(2) = 1$,
  - $F(3) = 2$.

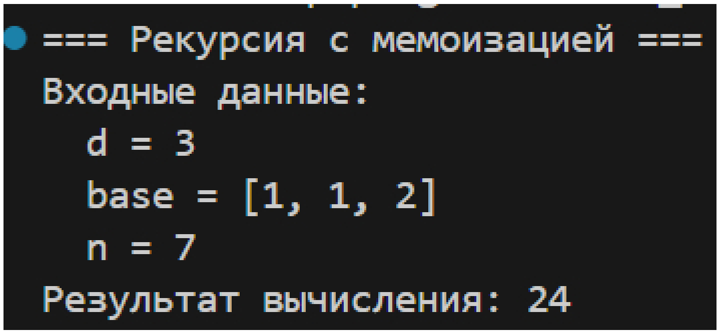
1. **Вызов:**
   Вызываем функцию "$F\_memo(7, 3, [1, 1, 2])$".
   Так как "$7 > 3$", функция не попадает в базовый случай.
2. **Проверка кэша:**
   При каждом вызове проверяется, есть ли значение "$F(n)$" в "$memo$". Если нет, то происходит вычисление.
3. **Вычисление $F(7)$:**
   "$F(7) = F(6) + F(5) + F(4)$".
   Функция вызывает "$F\_memo(6, 3, base)$", "$F\_memo(5, 3, base)$" и "$F\_memo(4, 3, base)$".
4. **Вычисление $F(4)$:**
   "$F(4) = F(3) + F(2) + F(1)$".
   Все эти значения – базовые: "$F(3) = 2$", "$F(2) = 1$", "$F(1) = 1$",
   поэтому "$F(4) = 2 + 1 + 1 = 4$". Результат сохраняется в "$memo[4] = 4$".
5. **Вычисление $F(5)$:**
   "$F(5) = F(4) + F(3) + F(2)$".
   Уже вычислено: "$F(4) = 4$", плюс базовые "$F(3) = 2$" и "$F(2) = 1$",
   итого "$F(5) = 4 + 2 + 1 = 7$". Сохраняем "$memo[5] = 7$".
6. **Вычисление $F(6)$:**
   "$F(6) = F(5) + F(4) + F(3)$".
   Из кэша: "$F(5) = 7$", "$F(4) = 4$", а "$F(3) = 2$" – базовое,
   таким образом "$F(6) = 7 + 4 + 2 = 13$". Сохраняем "$memo[6] = 13$".
7. **Итоговое вычисление $F(7)$:**
   Подставляем:
   "$F(7) = 13 + 7 + 4 = 24$". Сохраняем "$memo[7] = 24$".

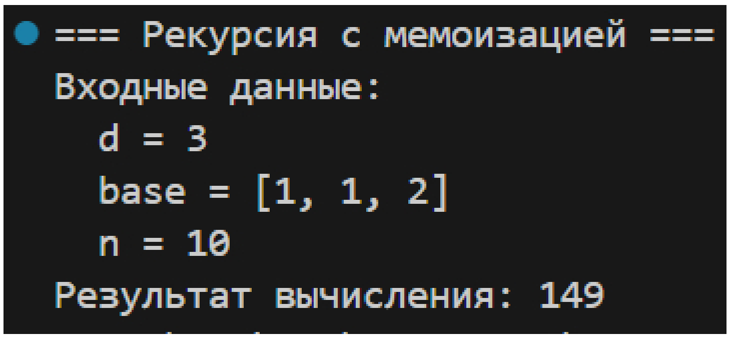
Код на Python

```python
def F_memo(n, d, base, memo=None):
    """
    Вычисляет n-й член последовательности с использованием мемоизации.

    Параметры:
      n    - искомый номер элемента (n > d)
      d    - параметр рекурсии
      base - список базовых значений
      memo - словарь для кэширования результатов (по умолчанию None)

    Сложность:
      Время: O(n * d)
      Память: O(n)
    """
    if memo is None:
        memo = {}
    if n <= d:
        return base[n - 1]
    if n in memo:
        return memo[n]
    result = 0
    for i in range(1, d + 1):
        result += F_memo(n - i, d, base, memo)
    memo[n] = result
    return result

# Пример использования:
if __name__ == "__main__":
    d = 3
    base = [1, 1, 2]
    n = 7

    # Сначала выводим все входные данные
    print("=== Рекурсия с мемоизацией ===")
    print("Входные данные:")
    print("  d =", d)
    print("  base =", base)
    print("  n =", n)
```

```
    # Вычисляем и выводим результат
    result = F_memo(n, d, base)
    print("Результат вычисления:", result)
```

Скриншоты выполнения кода





Временная сложность:

При использовании мемоизации каждое значение $F(k)$ для k от 1 до n вычисляется только один раз. Для вычисления $F(k)$ требуется суммировать d предыдущих значений, что даёт затраты на уровне каждого k равные "$O(d)$". Поскольку таких значений примерно "n" (точнее, $n - d$ вычислений сверх базовых), общая временная сложность равна "$O(n \times d)$". Это существенно лучше экспоненциального роста наивной рекурсии.

Сложность по памяти:

В данном подходе используется кэш (например, словарь "$memo$") для хранения результатов вычислений для каждого k, что требует памяти пропорционально числу вычисляемых значений, то есть "$O(n)$". Помимо этого, стек рекурсии может достигать глубины $O(n)$ в худшем случае, что также влияет на расход памяти.

Плюсы:

- **Эффективность:** Значительно уменьшается количество повторных вычислений, так как каждое значение вычисляется один раз.
- **Простота реализации:** Логика остаётся похожей на наивую рекурсию, но с добавлением кэша.

Минусы:

- **Дополнительная память:** Необходим кэш для хранения промежуточных результатов, что может быть критичным при очень больших значениях n.
- **Стек вызовов:** Рекурсивная реализация всё ещё использует стек, что может привести к переполнению при очень больших n.

Вариант 3. Итеративное динамическое программирование (Bottom-Up)

Описание:

Создаем массив (или список) "dp" длины "n" для хранения значений от 1 до n. Сначала в него записываются базовые значения, затем последовательно вычисляем каждое $F(i)$ как сумму предыдущих d значений (для каждого "i" от "$d + 1$" до "n" вычисляется значение по формуле " $dp[i] = dp[i - d] + dp[i - d + 1] + \cdots + dp[i - 1]$").

Пошаговое выполнение для $n = 7$, $d = 3$, $base = [1, 1, 2]$:

- базовые значения заданы как $base = [1, 1, 2]$, то есть
  - $F(1) = 1$,
  - $F(2) = 1$,
  - $F(3) = 2$.

1. **Инициализация:**
   Создаётся массив "dp" размером "7".
   Записываем:
   - "$dp[1] = F(1) = 1$",
   - "$dp[2] = F(2) = 1$",

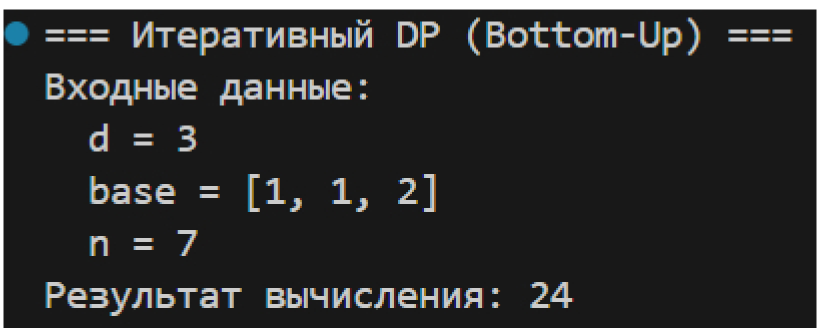
- "$dp[3] = F(3) = 2$".

2. **Вычисление $F(4)$:**
   "$dp[4] = dp[1] + dp[2] + dp[3] = 1 + 1 + 2 = 4$".
3. **Вычисление $F(5)$:**
   "$dp[5] = dp[2] + dp[3] + dp[4] = 1 + 2 + 4 = 7$".
4. **Вычисление $F(6)$:**
   "$dp[6] = dp[3] + dp[4] + dp[5] = 2 + 4 + 7 = 13$".
5. **Вычисление $F(7)$:**
   "$dp[7] = dp[4] + dp[5] + dp[6] = 4 + 7 + 13 = 24$".

Код на Python

```python
def F_bottom_up(n, d, base):
    """
    Вычисляет n-й член последовательности итеративно (Bottom-Up).

    Параметры:
      n    - искомый номер элемента (n > d)
      d    - параметр рекурсии
      base - список базовых значений

    Сложность:
      Время: O(n * d)
      Память: O(n)
    """
    dp = [0] * n  # инициализируем массив для значений от 1 до n
    # Заполнение базовых случаев
    for i in range(d):
        dp[i] = base[i]
    # Вычисление последующих значений
    for i in range(d, n):
        dp[i] = sum(dp[i - d:i])
    return dp[n - 1]

# Пример использования:
if __name__ == "__main__":
    d = 3
    base = [1, 1, 2]
    n = 7

    # Сначала выводим все входные данные
    print("=== Итеративный DP (Bottom-Up) ===")
    print("Входные данные:")
    print("  d =", d)
    print("  base =", base)
    print("  n =", n)

    # Вычисляем и выводим результат
    result = F_bottom_up(n, d, base)
    print("Результат вычисления:", result)
```

Скриншоты выполнения кода



```
=== Итеративный DP (Bottom-Up) ===
Входные данные:
  d = 3
  base = [1, 1, 2]
  n = 7
Результат вычисления: 24
```

```
=== Итеративный DP (Bottom-Up) ===
Входные данные:
  d = 3
  base = [1, 1, 2]
  n = 10
Результат вычисления: 149
```

Временная сложность:

Итеративное динамическое программирование строит массив "dp" от 1 до n, где для каждого индекса i (начиная с $i = d + 1$) вычисляется значение как сумма предыдущих d элементов. Для каждого i требуется выполнить суммирование "d" чисел, что даёт затраты "$O(d)$" на итерацию. Так как таких итераций примерно "$n - d$", общая временная сложность составит "$O((n - d) \times d)$", что асимптотически равносильно "$O(n \times d)$".

Сложность по памяти:

Для хранения всех промежуточных значений используется массив (или список) длиной "n". Следовательно, пространственная сложность равна "$O(n)$", поскольку необходимо хранить результат для каждого индекса от 1 до n.

Плюсы:

- **Отсутствие рекурсии:** Нет накладных расходов на вызовы функции, что исключает риск переполнения стека.
- **Простота:** Логика последовательного заполнения массива проста для понимания и реализации.

Минусы:

- **Память:** Если нас интересует только последний элемент, хранить весь массив может быть не оптимально.
- **Дублирование вычислений:** Для каждого нового элемента требуется суммировать d предыдущих значений, что при больших d увеличивает константу времени.

Вариант 4. Итеративное решение со скользящим окном

Описание:

Вместо хранения всего массива достаточно хранить последние "d" значений в виде списка (скользящего окна) и текущую сумму этих значений. На каждом шаге вычисляется новое значение как "$next\_value = current\_sum$", после чего обновляется окно: удаляется самое старое значение и добавляется "next_value", а сумма пересчитывается.

Пошаговое выполнение для $n = 7$, $d = 3$, $base = [1, 1, 2]$:

- базовые значения заданы как $base = [1, 1, 2]$, то есть
  - $F(1) = 1$,
  - $F(2) = 1$,
  - $F(3) = 2$.

1. **Инициализация:**
   Копируем базовые значения в окно:
   "$window = [1, 1, 2]$".
   Вычисляем начальную сумму:
   "$current\_sum = 1 + 1 + 2 = 4$".
2. **Если $n \leq d$:**
   Для "$n = 7$" условие "$7 \leq 3$" не выполняется, переходим к итерациям.
3. **Первый шаг итерации (вычисление $F(4)$):**
   - "$next\_value = current\_sum = 4$".
   - Обновляем окно: удаляем первый элемент "1" и добавляем "4", получаем "$window = [1, 2, 4]$".
   - Обновляем сумму:
     "$current\_sum = previous\_sum - removed + next\_value = 4 - 1 + 4 = 7$".
4. **Второй шаг итерации (вычисление $F(5)$):**
   - "$next\_value = current\_sum = 7$".
   - Обновляем окно: удаляем первый элемент "1" из "$window = [1, 2, 4]$", получаем "$[2, 4]$", добавляем "7" → "$window = [2, 4, 7]$".
   - Обновляем сумму:
     "$current\_sum = 7 - 1 + 7 = 13$".
5. **Третий шаг итерации (вычисление $F(6)$):**

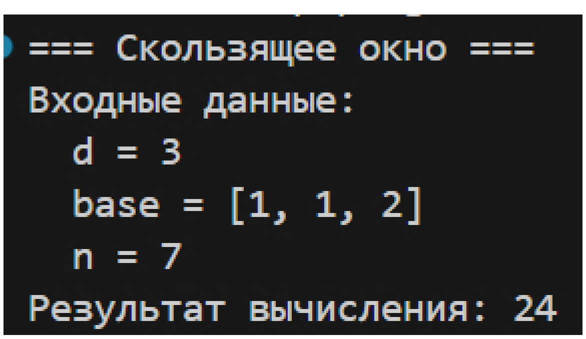
- "$next\_value = current\_sum = 13$".
- Обновляем окно: удаляем первый элемент "2" из "$window = [2, 4, 7]$", получаем "$[4, 7]$", добавляем "13" → "$window = [4, 7, 13]$".
- Обновляем сумму: "$current\_sum = 13 - 2 + 13 = 24$".

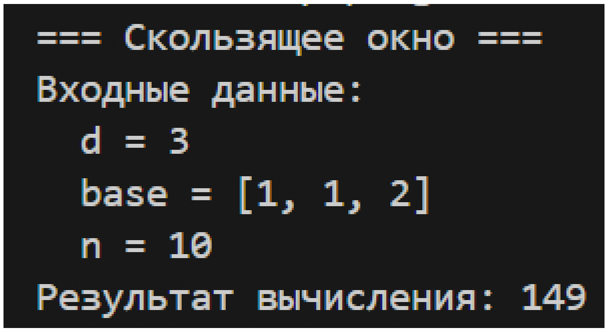
6. **Четвёртый шаг итерации (вычисление $F(7)$):**
   - "$next\_value = current\_sum = 24$".
   - Обновляем окно: удаляем первый элемент "4" из "$window = [4, 7, 13]$", получаем "$[7, 13]$", добавляем "24" → "$window = [7, 13, 24]$".
   - Обновляем сумму: "$current\_sum = 24 - 4 + 24 = 44$" (хотя для получения $F(7)$ достаточно последнего элемента окна).
7. **Итог:**
   После завершения итераций последнее значение в окне – это "$F(7) = 24$".

Код на Python

```python
def F_sliding_window(n, d, base):
    """
    Вычисляет n-й член последовательности с использованием скользящего окна.

    Параметры:
      n    - искомый номер элемента (n > d)
      d    - параметр рекурсии
      base - список базовых значений

    Сложность:
      Время: O(n)
      Память: O(d)
    """
    window = base.copy()  # начальное окно из базовых значений
    current_sum = sum(window)
    # Если n не превышает количество базовых значений, возвращаем соответствующий элемент
    if n <= d:
        return window[n - 1]
    # Вычисляем элементы от d+1 до n
    for i in range(d, n):
        next_value = current_sum
        # Обновляем окно: удаляем первый элемент и добавляем новое значение
        current_sum = current_sum - window.pop(0) + next_value
        window.append(next_value)
    return window[-1]

# Пример использования:
if __name__ == "__main__":
    d = 3
    base = [1, 1, 2]
    n = 7

    # Сначала выводим все входные данные
    print("=== Скользящее окно ===")
    print("Входные данные:")
    print("  d =", d)
    print("  base =", base)
    print("  n =", n)

    # Вычисляем и выводим результат
    result = F_sliding_window(n, d, base)
    print("Результат вычисления:", result)
```

Скриншоты выполнения кода





```
=== Скользящее окно ===
Входные данные:
  d = 3
  base = [1, 1, 2]
  n = 10
Результат вычисления: 149
```

Временная сложность:

В этом методе используется скользящее окно, которое хранит последние "d" значений, и поддерживается переменная "$current\_sum$". При каждой итерации новое значение вычисляется за счёт простой операции вычитания одного элемента и добавления нового (то есть за "$O(1)$" операций). Так как итераций выполняется примерно "$n - d$", общая временная сложность записывается как "$O(n)$", что делает метод очень эффективным.

Сложность по памяти:

Память требуется только для хранения скользящего окна, которое содержит ровно "d" элементов, поэтому пространственная сложность составляет "$O(d)$". При условии, что d обычно мало (например, $d \leq 10$), этот метод требует минимального объёма памяти по сравнению с предыдущими вариантами.

Плюсы:

- **Эффективность по времени:** Обновление суммы происходит за константное время, что значительно быстрее суммирования d элементов на каждой итерации.
- **Оптимальное использование памяти:** Вместо хранения всего массива используется лишь окно из "d" элементов, то есть память расходуется в размере "$O(d)$".

Минусы:

- **Ограниченность применения:** Этот метод удобен, если требуется только последний элемент последовательности. Если нужно восстановить всю последовательность, придется сохранять дополнительные данные.
- **Сложность логики:** Логика обновления окна требует аккуратного контроля за порядком удаления и добавления элементов, что может быть менее очевидным для начинающих.

Сравнение с классическим случаем d=2

В классическом случае с $d = 2$ все варианты алгоритмов работают быстрее и эффективнее по времени за счёт меньшего числа суммируемых элементов, а также используют чуть меньше памяти в методах, где важно хранение только последних элементов (например, метод со скользящим окном). Однако принципиальная разница в асимптотических оценках для итеративных решений заключается лишь в константном множителе, в то время как для наивой рекурсии разница становится критичной из-за экспоненциального роста числа вызовов.