**Оптимизация**

*Оптимизация* – это изменение системы для повышения её эффективности.

Цель оптимизации: получение системы оптимальной с точки зрения используемого объема памяти и затрачиваемого в ходе работы время.

Варианты оптимизации:

1. Уменьшение времени, требуемого для работы. Может потребоваться больший объем памяти.
2. Уменьшение требуемого объема памяти при использовании более медленных алгоритмов.

Основные принципы оптимизации:

1. Код должен быть аккуратным, модульным и легко читаемым для более легкого редактирования.
2. В результате оптимизации должен быть прирост производительности ПО. Хорошо если прирост 10-30%
3. Оптимизация и последующая отладка должны занимать не более 10-15% времени разработки.

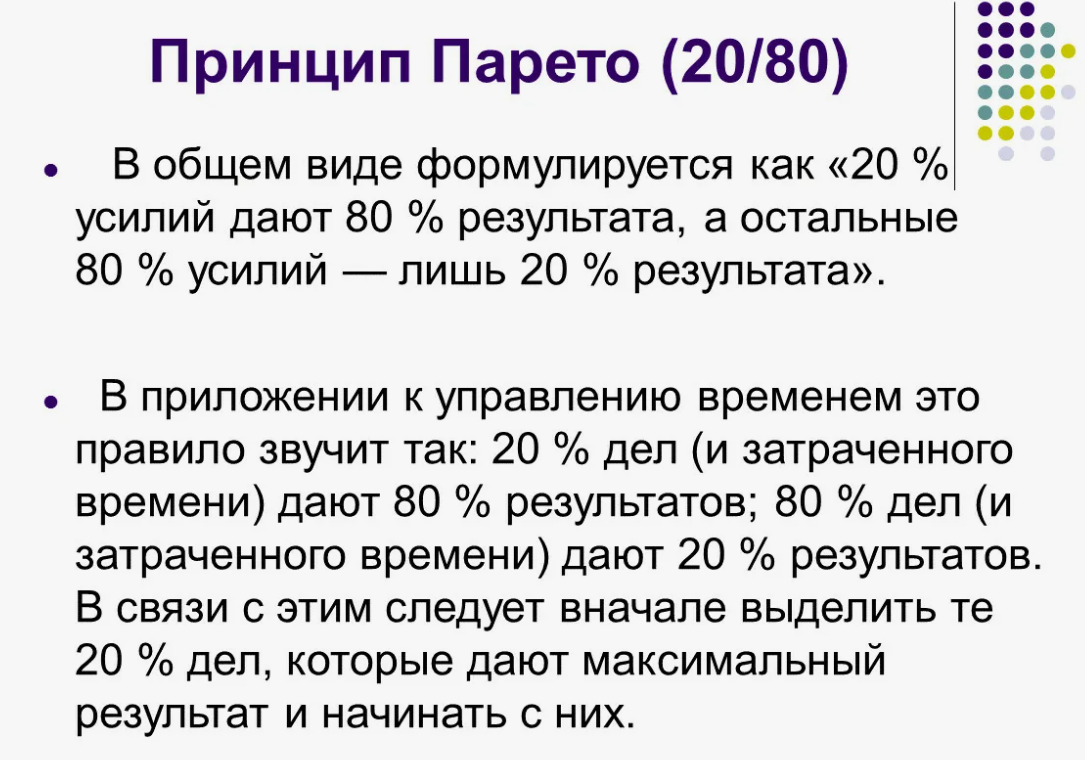
Рекомендации:

1. Перед началом оптимизации обязательно сохранить копию исходного рабочего кода.
2. Оптимизировать следует в том случае если выявлена нехватка ресурсов.
3. Для оптимизации требуется найти «бутылочное горлышко».

*«Бутылочное горлышко»* – это критическая часть кода, которая является основным потребителем необходимого ресурса.

1. При выполнении оптимизации следует помнить о принципе Парето.

Изменение или улучшение примерно 20% кода иногда улучшает результат на 80%.



Для поиска узких мест и утечек памяти вводятся специализированные отладочные средства и профилировщики.

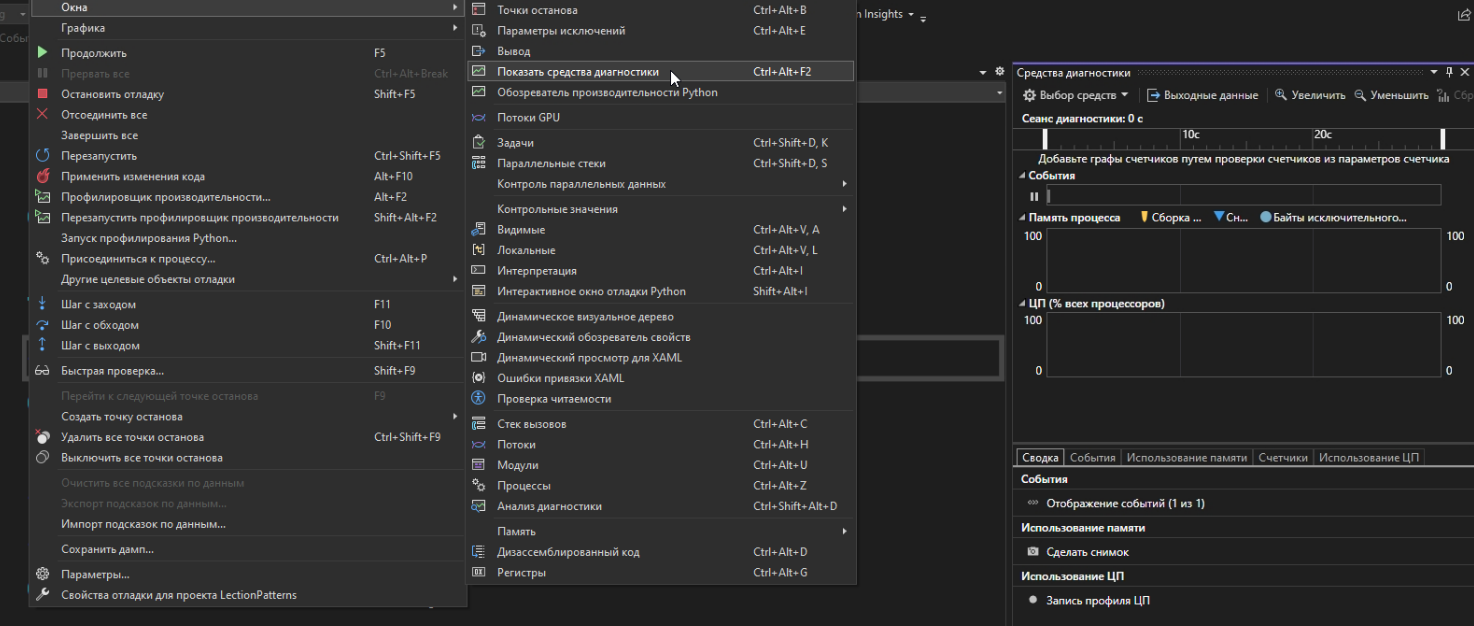
*Профилированные* – это сбор характеристик программы (время выполнения, объем использованной памяти, количество условных переходов и т.д.). Обычно используется совместно с оптимизацией.

*Profiler* (профилировщик) – специальный отладочный инструмент, используемый для анализа работы программы и поиска узких мест и утечек.

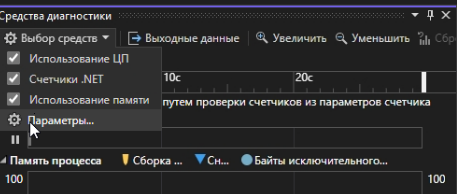
Пример профилировщика:

В VS существует собственный профайлер

Открыть средства диагностики (Окна=> Показать средства диагностики)



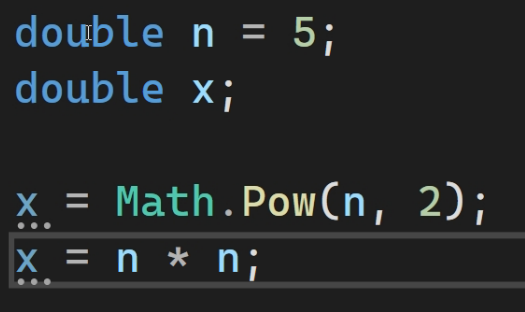
В параметрах можно ограничить доступные средства (Выбор средств => Параметры)

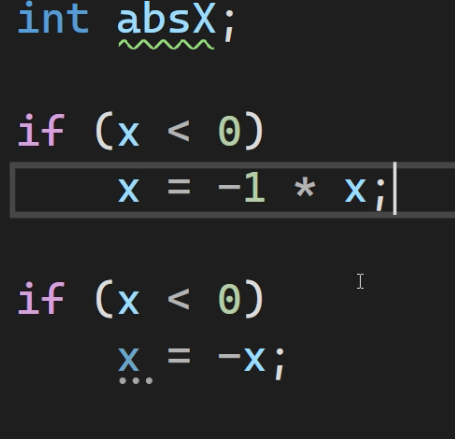


Принципы поиска узких мест:

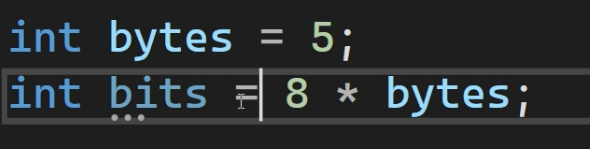
1. Не изменять ту часть программы, которая уже оптимально работает
2. Проанализировать блоки кода, которые часто повторяются в ходе работы программы. Часто повторяются циклы и подпрограммы

Пример правильного и неправильного кода:





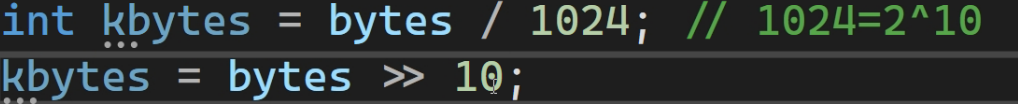
Плохой код:



Хороший:



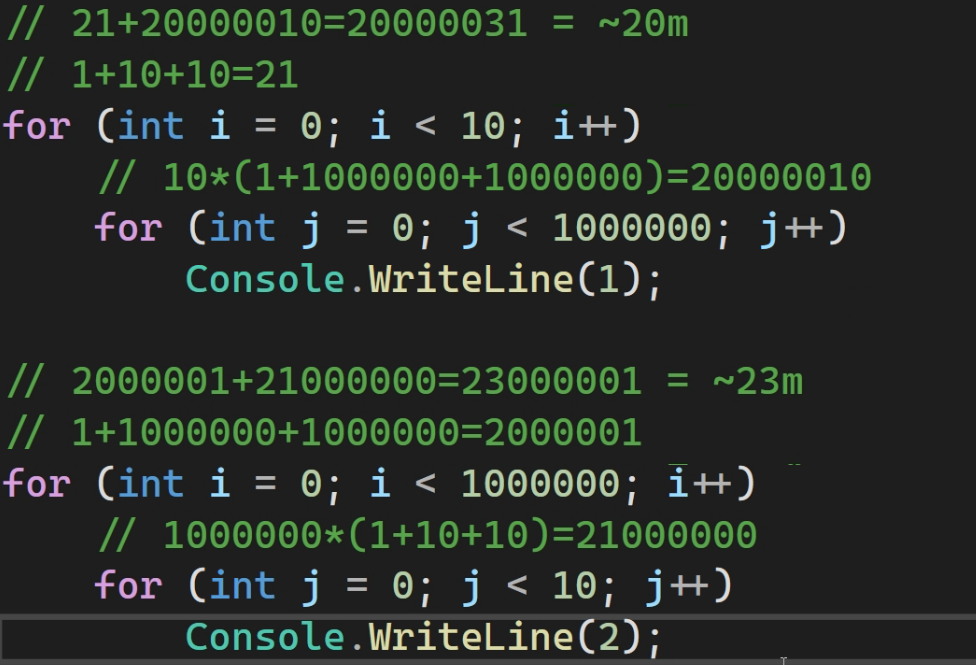
Сдвиг влево это замена умножений на число в какой-то степени



Правила оптимизации циклов:

1. Не выполнять в цикле то, что можно сделать за его пределами.
2. Не объявлять в цикле переменные.
3. Не вызывать методы
4. Избавление от передачи управления внутри цикла

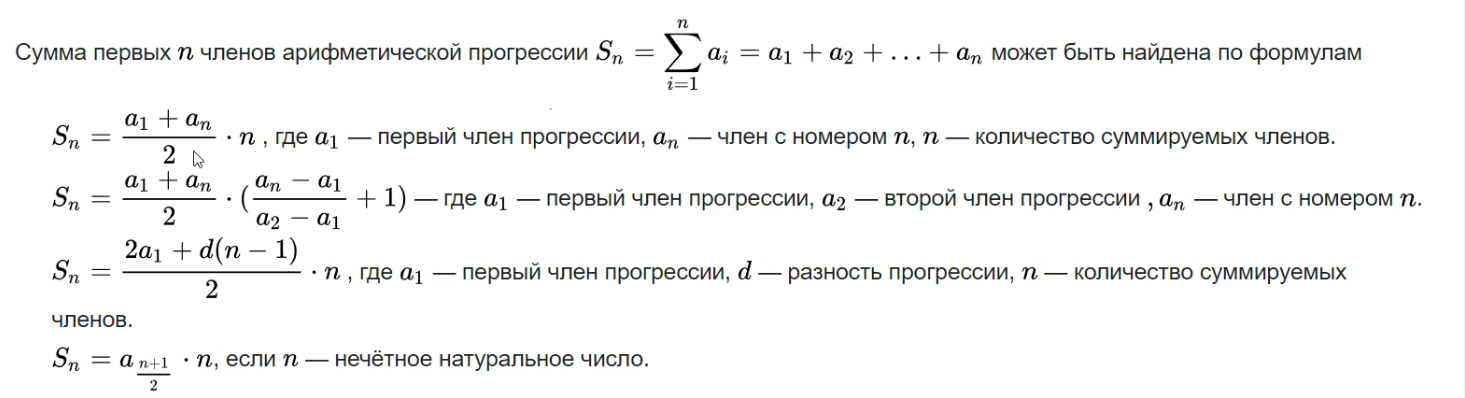
Пример подсчета количества операций при работе сложных циклов:



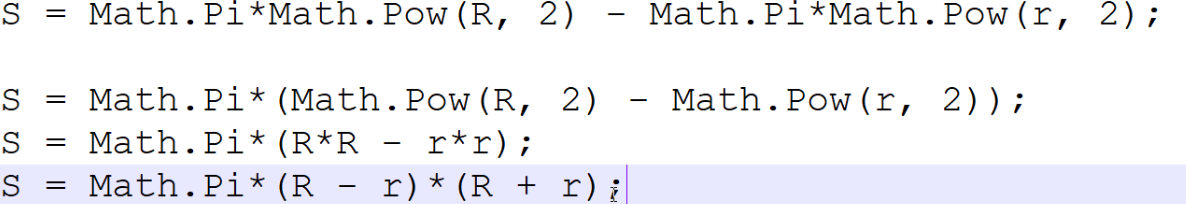
! Первым надо выполнять цикл с наименьшим количеством

Применение формул для оптимизации:

Пример 1. Для подсчета суммы чисел последовательности вместо цикла for использовать форму.

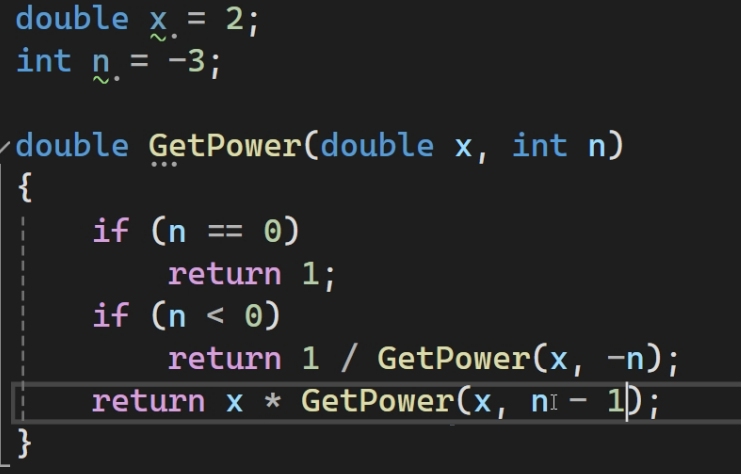


Пример 2. Улучшение формулы расчёта площади кольца.

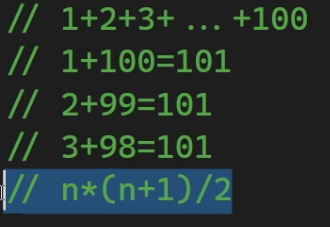


Пример 3. Возведение в степень.

x^n



Пример 4. Найти сумму первых n чисел (с 1 до n)

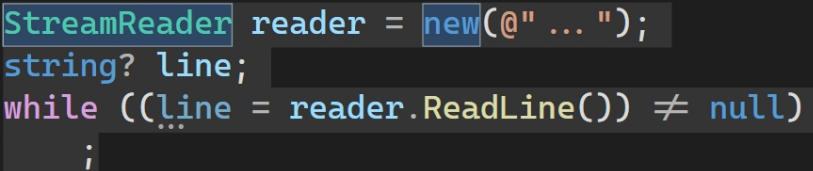


Пример 5. Чтение файлов построчно

 - чтение файла целиком. Это выигрыш во времени, но пока файл не прочитает целиком, то ничего нельзя сделать.

🗸🗴

*StreamReader* – класс для потокового чтения



Выигрыш в памяти и не нужно ждать, пока загрузится весь файл

If (условие == true); 🗴

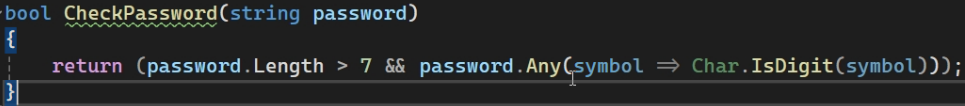
If (условие); 🗸

If (условие != false); 🗴

If (!условие); 🗸

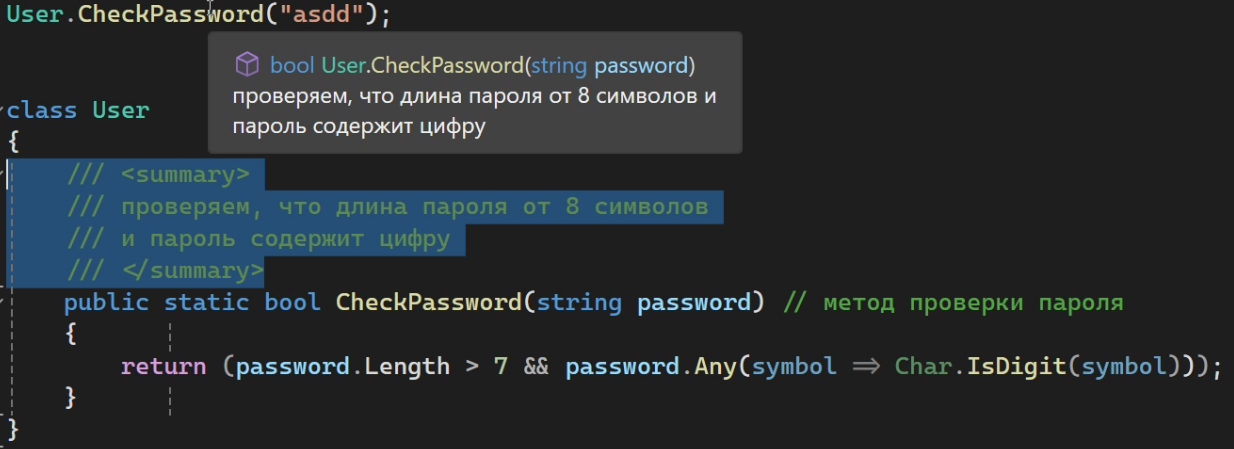
Исключение:

bool? true / false / null (условие == true) 🗸



Подготовка к оптимизации программы:

1. Проверить код на наличие ненужных или устаревших фрагментов.
2. Писать понятный и самодокументированный код, соблюдая code style языка программирования и команды разработки.
3. Снабжать непонятные блоки кода комментариями.



1. Выявить узкие или пробные места.
2. Применить компиляровщик.

*Code review* – это просмотр кода.

Методы оптимизации:

1. Настройка окружения.
2. Удаление ненужного кода.
3. *Меморизация* – это сохранение промежуточных результатов или результатов выполнения функции, чтобы не выполнять её повторно.
4. Распараллеливание.
5. Ленивое вычисление. Вычисления откладываются до необходимости получения результата. Большинство LINQ-методов выполняют ленивые вычисления.



1. *Метод приближения* – это замена точного алгоритма на неточный, но быстродействующий.
2. Использование сторонних языков. Например assembler-вставок.