# Задание на анализ кода

Дан исходный код двух функций. Необходимо проанализировать исходный код и написать, что он выполняет. Если код можно оптимизировать, то предложить свой вариант.

```
static void Func2(ref KeyValuePair<int, string>[] a, int key, string value)
{
  int pos;
  KeyValuePair<int, string> keyValuePair;

  if (a.Length == 0)
  {
    Array.Resize(ref a, 1);
    keyValuePair = new KeyValuePair<int, string>(key, value);
    a[0] = keyValuePair;
    return;
  }

  if (key < a[0].Key)
    pos = 0;
  else if (key > a[a.Length - 1].Key)
    pos = a.Length;
  else
    pos = Func1(a, 0, a.Length - 1, key);

  Array.Resize(ref a, a.Length + 1);
  for (int i = a.Length - 1; i > pos; i--)
    a[i] = a[i - 1];

  keyValuePair = new KeyValuePair<int, string>(key, value);
  a[pos] = keyValuePair;
}
```

## Анализ задачи

Скорее всего реализуется аналог SortedList'a на базе массива `KeyValuePair<int, string>` только без проверки на уникальность ключа.

Первая функция осуществляет бинарный поиск по массиву `KeyValuePair<int, string>`

Вторая функция аналог метода Add у SortedList'a.

## Анализ кода

- Название функций нечитаемое. Под замену.
- Нет XML комментариев.
- Нет смысла в ручной реализации бинарного поиска. У класса Array есть метод BinarySearch. Достаточно реализовать IComparer и передавать в Array.BinnarySearch().
- Если все же решили писать бинарный поиск руками, то желательно вместо рекурсии использовать цикл. Меньше аллокаций, свободнее память, меньше чисток мусора, выше производительность.
- Если все же решили писать бинарный поиск руками, то желательно делать эту функцию универсальной. Нужно писать проверки. Как минимум на массив == null и выход за пределы массива.
- Текущие названия аргументов первой функции могут ввести в заблуждение: low и high можно воспринять как значения границ поиска, а не индексов границ поиска.
- Если мы пишем свою коллекцию, то логичнее всего будет создать отдельный класс с удобным интерфейсом взаимодействия, а не передавать каждый раз массив в функцию.
- Если мы пишем свою коллекцию, значит мы будем добавлять в нее элементы. Вероятно что это делать мы будем очень часто. Текущая реализация бьет по производительности - при каждом добавлении элемента мы увеличиваем массив на 1, что влечет за собой полное копирование всех элементов в новый массив с увеличенным размером на 1. Кроме того мы чаще будем чистить мусор, ведь часто будем увеличивать массив на 1, а коллекции могут быть очень большими и это может стоить дорого. На мой взгляд нужно следовать примеру System.Collections: создавать по умолчанию с определенным размером, затем при превышении Capacity увеличивать размер массива в 2 раза. Но это возможно только только если мы будем использовать наш новый класс вместо массива `KeyValuePair<int, string>`, поскольку нам нужно будет еще хранить Count и Capacity.

- Нет смысла руками сдвигать массив. Для этого есть Array.CopyTo в этот же массив. Под капотом он unsafe с оптимизацией. Это будет работать быстрее.
- Вместо массива было бы удобнее `List<KeyValuePair<int, string>>`. И используя вместо массивов листы можно было не реализовывать умное расширение размера.
- Если уже сортируем по ключу, было бы здорово еще и сортировать по значению.

## Оптимизация

Исхожу из того, что по какой-то причине мы не можем создать отдельный класс для нашей коллекции и мы обязаны сделать функции для работы с именно массивом `KeyValuePair<int, string>`. Значит провести оптимизацию по выделению памяти я не могу. Допускаю условность, что мы точно знаем, что так будем взаимодействовать только с `KeyValuePair<int, string>` и нам нет смысла делать функции generic'овыми.

Исходник тык