V.03.

Общий обзор занятия.

Влезли мы к слайсам (а также, конечно, к массивам, структам и строкам, но слайсы дают самые интересные эффекты) в печёнки. И в этот раз как раз момент немного отдышаться, подогнать хвосты, оставшиеся с прошлых занятий (а они точно остались, там уж очень плотно замешано). И первым делом для закрепления понимания устройства слайса посмотрим как ведут себя слайсы, переданные как параметры по значению и по адресу. Тут всё должно очень выпукло проявляться.

А потом рассмотрим некоторые стандартные операции со слайсами - как их реализовать. В принципе, мы об этом уже говорили в прошлый раз, но тут собраны операции либо не столь очевидные, либо такие, где лучший вариант не так легко выбрать, либо с намёком на близкое будущее. А ждёт нас там разговор о простых структурах данных. Которые мы будем в осноном реализовывать на связных структурах, но которые можно реализовывать и на слайсах - стек, дек, очередь, кольца и т.д. И это главная часть занятия.

И в заключение даём, показывем детям возможности, которые даёт нам для тестирования производительности (benchmarking) package testing. И чуть-чуть про команды go, точнее минимально про одну команду - go test. Будет у детей на практике возможность пореализовывать просмотренные стандартные движения со слайсами и потестировать производительность того или иного варианта.

Новый материал - основные вопросы.

- Слайсы как параметры - передача по значению и передача по адресу
- Трюки (стандартные действия) со слайсами
    - Движения типа слайс и элемент
        - Push / Добавить элемент x в конец слайса
        - Pop / Выдернуть последний элемент слайса
        - Push Front/Unshift / Вставить элемент x в начало
        - Pop Front/Shift / Выдернуть первый элемент слайса
        - Insert / Вставить x на i-ю позицию
        - Insert / Вставить x на i-ю позицию.
        - Get / Выдернуть i-й элемент слайса
        - Delete / Удалить i-й элемент, сохраняя порядок следования оставшихся
        - Safe delete/ Удалить i-й элемент, сохраняя порядок следования

оставшихся - безопасная (корректная) версия.
        - Safe delete last/ Безопасно удалить последний элемент.
        - Safe delete front/ Безопасно удалить начальный элемент.
        - Delete without preserving order / Удалить i-й элемент, не сохраняя порядок следования оставшихся
        - Safe delete without preserving order / Корректно удалить i-й элемент, не сохраняя порядок следования оставшихся
    - Движения типа слайс и слайс
        - Append slice / Прицепить b в конец a
        - Copy / Копировать a в b
        - Cut / Удалить отрезок a[i:j]
        - Safe cut / Безопасно удалить отрезок a[i:j]
        - Extend / Добавить в конец слайса пустой отрезок длины j
        - Expand / Вставить пустой отрезок длины j внутрь слайса, начиная с позиции i
          InsertVector / Вставить слайс b внутрь слайса a, начиная с позиции i
    - Другие операции
        - Reversing / Переворачивание слайса задом наперёд “на месте” - без перераспределения памяти.
        - Filtering without allocating / Фильтрация “на месте”
        - Batching with minimal allocation / Расщепление слайса на пакеты данных фиксированной длины
        - Move to front, or append if not present, in place / Поиск первого вхождения заданного элемента и перемещение его вперёд (без перераспределения памяти). Если такого элемента не было, то добавляем его в конец слайса.

- Benchmarking, или Тест производительности
    - Benchmark с помощью команды `go test`
    - Benchmark с помощью функции `testing.Benchmark`, аргументом которой является функция типа `func(b *testing.B)`

Лекция

Передача параметров-слайсов по значению и по адресу. Что, где, когда.

Главная сложность, главная причина всевозможных путаниц со слайсами состоит в том, что с одной стороны есть переменная-слайс, у неё естественно есть адрес, там она живёт и хорошеет, а с другой стороны сама эта переменная содержит не непостредственно данные (которые, если не влезать глубже, и хочется считать слайсом), а адрес, указатель на эти данные. И данные могут запросто меняться, а указатель - нет. И вот тут вот как раз требуется очень чёткое понимание, очень ясное представление, что из себя представляет слайс. Что, например, при передаче слайса по значению изменения длины формального параметра не отразится в фактическом параметре, а вот изменения данных вполне себе отразяться, если только не произошло перераспределение памяти. А изменения ёмкости слайса-формального параметра тоже не отображаются в слайсе-фактическом параметре вовсе не потому, что изменение ёмкости слайса всегда связано с перераспределением памяти и т.д. Всё это довольно просто, но необходимо кристалльно ясное понимание процессов. Впрочем, это я уже повторяюсь. Так что просто рассматриваем какие-то примеры, может дети что-нибудь ещё нафантазируют. И смотрим, что получается, и объясняем полученные результаты.

Чтобы освежить всё это - серия примеров. В каждом примере три функции. Делают все три одно и тоже, но первая получает слайс по значению, вторая - по адресу, а третья - получает слайс-значение, изменят его и возвращает результат.
Пример `01a.go`.

```
package main

import     "fmt"

// Изменяем только данные слайса-параметра
func f1(c []int)  {
    for i, _ := range c  {
         c[i] +=100
    }
    fmt.Println(c, len(c), cap(c))
}

func f2(pc *[]int)  {
    for i, _ := range *pc  {
        (*pc)[i] +=100
    }
```

```
    fmt.Println(*pc, len(*pc), cap(*pc))
}

func f3(c []int) []int {
    for i, _ := range c  {
        c[i] +=100
    }
    fmt.Println(c, len(c), cap(c))
    return c
}

func main() {
    a:= [...]int {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a)                      // [0 1 2 3 4 5 6 7 8 9]

    arr:= a
    b:= arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f1(b)                               // [102 103 104 105 106] 5 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106] 5 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f2(&b)                              // [102 103 104 105 106] 5 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106] 5 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    b = f3(b)                           // [102 103 104 105 106] 5 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106] 5 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 7 8 9]
}
```

Пример 01b.go.

```go
package main

import     "fmt"

// Изменяем только длину слайса-параметра
func f1(c []int)  {
    c = c[:cap(c)]
    fmt.Println(c, len(c), cap(c))
}

func f2(pc *[]int)  {
    *pc = (*pc)[:cap(*pc)]
    fmt.Println(*pc, len(*pc), cap(*pc))
}

func f3(c []int) []int {
    c = c[:cap(c)]
    fmt.Println(c, len(c), cap(c))
    return c
}

func main() {
    a:= [...]int {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a)                      // [0 1 2 3 4 5 6 7 8 9]

    arr:= a
    b:= arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f1(b)                               // [2 3 4 5 6 7 8 9] 8 8
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f2(&b)                              // [2 3 4 5 6 7 8 9] 8 8
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6 7 8 9] 8 8
```

```
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    b = f3(b)                           // [2 3 4 5 6 7 8 9] 8 8
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6 7 8 9] 8 8
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]
}
```

Пример 01c.go.

```
package main

import     "fmt"

// Изменяем длину слайса-параметра, а затем данные
func f1(c []int)  {
    c = c[:cap(c)]
    for i, _ := range c  {
        c[i] +=100
    }
    fmt.Println(c, len(c), cap(c))
}

func f2(pc *[]int)  {
    *pc = (*pc)[:cap(*pc)]
    for i, _ := range *pc  {
        (*pc)[i] +=100
    }
    fmt.Println(*pc, len(*pc), cap(*pc))
}

func f3(c []int) []int {
    c = c[:cap(c)]
    for i, _ := range c  {
        c[i] +=100
    }
```

```
    fmt.Println(c, len(c), cap(c))
    return c
}

func main() {
    a:= [...]int {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a)                      // [0 1 2 3 4 5 6 7 8 9]

    arr:= a
    b:= arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f1(b)                               // [102 103 104 105 106 107 108 109] 8 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106] 5 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 107 108 109]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f2(&b)                              // [102 103 104 105 106 107 108 109] 8 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106 107 108 109] 8 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 107 108 109]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    b = f3(b)                           // [102 103 104 105 106 107 108 109] 8 8
    fmt.Println(b, len(b), cap(b))      // [102 103 104 105 106 107 108 109] 8 8
    fmt.Println(arr)                    // [0 1 102 103 104 105 106 107 108 109]
}
```

Пример 01d.go.

```
package main

import     "fmt"

// Добавляем в слайс-параметр данные по одному,
// пока не произойдёт перераспределения памяти
```

```go
func f1(c []int)  {
    ccap:= cap(c)
    for cap(c) == ccap  {
        c = append(c, len(c)*10)
    }
    fmt.Println(c, len(c), cap(c))
}

func f2(pc *[]int)  {
    ccap:= cap(*pc)
    for cap(*pc) == ccap  {
        *pc = append(*pc, len(*pc)*10)
    }
    fmt.Println(*pc, len(*pc), cap(*pc))
}

func f3(c []int) []int {
    ccap:= cap(c)
    for cap(c) == ccap  {
        c = append(c, len(c)*10)
    }
    fmt.Println(c, len(c), cap(c))
    return c
}

func main() {
    a:= [...]int {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a)                      // [0 1 2 3 4 5 6 7 8 9]

    arr:= a
    b:= arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f1(b)                               // [2 3 4 5 6 50 60 70 80] 9 16
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 50 60 70]

    arr = a
    b = arr[2:7]
```

```
    fmt.Println(b, len(b), cap(b))     // [2 3 4 5 6] 5 8
    f2(&b)                             // [2 3 4 5 6 50 60 70 80] 9 16
    fmt.Println(b, len(b), cap(b))     // [2 3 4 5 6 50 60 70 80] 9 16
    fmt.Println(arr)                   // [0 1 2 3 4 5 6 50 60 70]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))     // [2 3 4 5 6] 5 8
    b = f3(b)                          // [2 3 4 5 6 50 60 70 80] 9 16
    fmt.Println(b, len(b), cap(b))     // [2 3 4 5 6 50 60 70 80] 9 16
    fmt.Println(arr)                   // [0 1 2 3 4 5 6 50 60 70]
}
```

Пример 01e.go.

```
package main

import  "fmt"

// Добавляем в слайс-параметр данные группой, размер
// которой требует перераспределения памяти
func f1(c []int)  {
    cc:= make([]int, cap(c) - len(c) + 1)
    c = append(c, cc...)
    fmt.Println(c, len(c), cap(c))
}

func f2(pc *[]int)  {
    cc:= make([]int, cap(*pc) - len(*pc) + 1)
    *pc = append(*pc, cc...)
    fmt.Println(*pc, len(*pc), cap(*pc))
}

func f3(c []int) []int {
    cc:= make([]int, cap(c) - len(c) + 1)
    c = append(c, cc...)
    fmt.Println(c, len(c), cap(c))
    return c
```

```
}

func main() {
    a:= [...]int {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a)                      // [0 1 2 3 4 5 6 7 8 9]

    arr:= a
    b:= arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f1(b)                               // [2 3 4 5 6 0 0 0 0] 9 16
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    f2(&b)                              // [2 3 4 5 6 0 0 0 0] 9 16
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6 0 0 0 0] 9 16
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]

    arr = a
    b = arr[2:7]
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6] 5 8
    b = f3(b)                           // [2 3 4 5 6 0 0 0 0] 9 16
    fmt.Println(b, len(b), cap(b))      // [2 3 4 5 6 0 0 0 0] 9 16
    fmt.Println(arr)                    // [0 1 2 3 4 5 6 7 8 9]
}
```

Трюки со слайсами.

Движения типа слайс и элемент

Push / Добавить элемент x в конец слайса

```
a = append(a, x)
```

Pop / Выдернуть последний элемент слайса

```
x, a = a[len(a)-1], a[:len(a)-1]
```

Push Front/Unshift / Вставить элемент x в начало

```
a = append([]T{x}, a...)
```

Pop Front/Shift / Выдернуть первый элемент слайса

```
x, a = a[0], a[1:]
```

Insert / Вставить x на i-ю позицию

Менее эффективно:

```
a = append(a[:i], append([]T{x}, a[i:]...)...)
```

Так, конечно, получается короткая и эффектная запись, но эффективнее сделать так:

Insert / Вставить x на i-ю позицию.

Более эффективно:

```
a = append(a, 0 /*нулевое значение типа элементов слайса*/)
copy(a[i+1:], a[i:])
a[i] = x
```

Рост скорости достигается за счёт того, что в самом первом случае по ходу пьесы создаются вспомогательные слайсы a[i:] и append([]T{x}, a[i:]...), а на выделение под них памяти и дальнейшее освобождение памяти от них требуется ресурс.

Insert / Вставить x на i-ю позицию.

Промежуточный вариант:

```
a = append(a[:i+1],a[i:]...)
a[i] = x
```

Используется только один временный слайс a[i:].

Get / Выдернуть i-й элемент слайса

```
x, a = a[i], append(a[:i], a[i+1:]...)
```

либо

```
x, a = a[i], a[:i+copy(a[i:], a[i+1:])]
```

Второй вариант уже немного припахивает трюкачеством - сначала сдвигаем хвост от (i+1)-го элемента на одну позицию влево, а потом откидываем последний элемент. copy как раз даёт нам нужную длину.

И совершенно то же самое мы делаем, если нам надо просто выбросить из слайса i-й элемент

Delete / Удалить i-й элемент, сохраняя порядок следования оставшихся

```
a = append(a[:i], a[i+1:]...)
```

либо

```
a = a[:i+copy(a[i:], a[i+1:])]
```

И вот здесь мы натыкаемся на разницу между взять/выдернуть элемент и вырезать/удалить/выкинуть элемент. А фишка здесь в том, что весьма возможен случай, когда выбрасываемый элемент слайса есть ссылка на что-нибудь, адрес чего-то. Ну, или является какой-то структурой (структура здесь - это не обязательно struct, хотя вполне может быть и struct'ом), которая содержит ссылки, хоят бы одну. И тогда сборщик мумора видит, что этот элемент слайса можно выбросить и выбрасывает его, а то, на что он ссылается, сборщик мусора, естественно, не анализирует, поскольку он тупо сбрасывает освободившееся. В итоге остается шмат навоза в памяти. Это явление имеет более интеллигентное и установившееся название - memory leak (утечка памяти), и это не есть хорошо. Так что более корректно писать так

Safe delete/ Удалить i-й элемент, сохраняя порядок следования

оставшихся - безопасная (корректная) версия.

```
copy(a[i:], a[i+1:])
a[len(a)-1] = nil // нулевое значение типа элементов слайса
a = a[:len(a)-1]
```

И естественные частные случаи

Safe delete last/ Безопасно удалить последний элемент.

a[len(a)-1] = nil // нулевое значение типа элементов слайса
a = a[:len(a)-1]

Safe delete front/ Безопасно удалить начальный элемент.

a[0] = nil // нулевое значение типа элементов слайса
a = a[1:]

Delete without preserving order / Удалить i-й элемент, не сохраняя порядок следования оставшихся

```
a[i] = a[len(a)-1]
a = a[:len(a)-1]
```

Safe delete without preserving order / Корректно удалить i-й элемент, не сохраняя порядок следования оставшихся

```
a[i] = a[len(a)-1]
a[len(a)-1] = nil //или нулевое значение типа элементов слайса
a = a[:len(a)-1]
```

Движения типа слайс и слайс

Append slice / Прицепить b в конец a

```
a = append(a, b...)
```

Copy / Копировать a в b

```
b = make([]T, len(a))
copy(b, a)
```

Cut / Удалить отрезок a[i:j]

```
if i <= j  { a = append(a[:i], a[j:]...) }
```

И опять, как всегда при удалении, бывает необходимо удалять более корректно

Safe cut / Безопасно удалить отрезок a[i:j]. Условие i<=j не проверяется.

```
copy(a[i:], a[j:])
for k, n := len(a)-j+i, len(a); k < n; k++ {
    a[k] = nil // или нулевое значение типа элементов слайса
}
a = a[:len(a)-j+i]
```

Extend / Добавить в конец слайса пустой отрезок длины j

```
a = append(a, make([]T, j)...)
```

Expand / Вставить пустой отрезок длины j внутрь слайса, начиная с позиции i

```
a = append(a[:i], append(make([]T, j), a[i:]...)...)
```

InsertVector / Вставить слайс b внутрь слайса a, начиная с позиции i

a = append(a[:i], append(b, a[i:]...)...)

И опять, для повышения эффективности предпочтительнее избегать использования append.

InsertVector / Вставить слайс b внутрь слайса a, начиная с позиции i

Более эффективно

```
func Insert(s []T, k int, vs []T) []T {
    if n := len(s) + len(vs); n <= cap(s) {
```

```
        s2 := s[:n]
        copy(s2[k+len(vs):], s[k:])
        copy(s2[k:], vs)
        return s2
    }
    s2 := make([]int, len(s) + len(vs))
    copy(s2, s[:k])
    copy(s2[k:], vs)
    copy(s2[k+len(vs):], s[k:])
    return s2
}
a = Insert(a, i, b)
```

Хотя, конечно, гораздо длиннее...

Другие операции

Reversing / Переворачивание слайса задом наперёд

Операция выполняется “на месте” - без перераспределения памяти.

```
// Variant 1.
last:= len(a) - 1
for i := len(a)/2-1; i >= 0; i-- {
    a[i], a[last-1] = a[last-i], a[i]
}
```

То же самое, но с двумя индексами. Выглядит естественнее.

```
// Variant 2.
for left, right := 0, len(a)-1; left < right; left, right = left+1, right-1 {
    a[left], a[right] = a[right], a[left]
}
```

Filtering without allocating / Фильтрация “на месте”

Фильтр - булевская функция keep:

`func keep(x SliceElementType) bool`

```
// Variant 1.
n := 0
for _, x := range a {
    if keep(x) {
        a[n] = x
        n++
    }
}
// if there exist elements which must be garbage
// collected, the following cycle can be included
for i:= n; i<len(a); i++  {
    a[i] = nil // or the zero value of T
}
a = a[:n]
```

```
// Variant 2.
b := a[:0]
for _, x := range a {
    if f(x) {
        b = append(b, x)
    }
}
// if there exist elements which must be garbage
// collected, the following cycle can be included
for i := len(b); i < len(a); i++ {
    a[i] = nil // or the zero value of T
}
a = a[:len(b)]
```

Здесь слайс b располагается над тем же массивом данных и имеет ту же ёмкость, что и слайс a. Так что перераспределения памяти не происходит, и слайс a изменяется именно так, как нам надо.

Batching with minimal allocation / Расщепление слайса на пакеты данных фиксированной длины

```
actions := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
batchSize := 3
batches := make([][]int, 0, (len(actions) + batchSize - 1) / batchSize)

for batchSize < len(actions) {
    actions, batches = actions[batchSize:], append(batches, actions[0:batchSize:batchS
ize])
}
batches = append(batches, actions)
// batches: [[0 1 2] [3 4 5] [6 7 8] [9]]
```

Move to front, or prepend if not present, in place if possible/Поиск первого вхождения заданного элемента и перемещение его вперёд (без перераспределения памяти). Если такого элемента не было, то добавляем его в начало слайса.

```
// moveToFront moves needle to the front of haystack, in place if possible.
func moveToFront(needle string, haystack []string) []string {
    if len(haystack) != 0 && haystack[0] == needle {
        return haystack
    }
    prev := needle
    for i, elem := range haystack {
        switch {
        case i == 0:
            haystack[0] = needle
            prev = elem
        case elem == needle:
            haystack[i] = prev
            return haystack
        default:
            haystack[i] = prev
            prev = elem
        }
    }
    return append(haystack, prev)
}
```

```
haystack := []string{"a", "b", "c", "d", "e"} // [a b c d e]
haystack = moveToFront("c", haystack)         // [c a b d e]
haystack = moveToFront("f", haystack)         // [f c a b d e]
```

Benchmarking, или Тест производительности

Тестирование кода, определение его производительности, проверка эффективности кода по времени, по памяти, проверка работоспособности кода в критических или пограничных условиях - всё это есть важнейшая составляющая работы. И go предоставляет очень развитые возможности для всего этого. Командная строка go поддерживает кучу разных команд (справка по ним: в командной строке подать команду `go -?`, естественно). В частности есть команда test (справка: в командной строке подать команду `go test -?`, естественно). У ней есть куча параметров и аргументов, которые позволяют очень разнообразно настроить процесс тестирования, есть всякие флаги, например, сколько запускать процессов параллельно, на скольки ядрах и т.д. Я приведу здесь пример, как можно протестировать производительность кода имено с помощью команды go test, но разбираться сейчас со всеми командамим, или даже с некоторыми, или даже с одной командой test - боюсь, будет в перебор. Так что я приведу пример, и скажем “так надо”. И приведу пример тестирования производительности в более привычный детям манере - с функцией main, с exe-файлом, ну, как обычно. Да, так в реальной разработке не делают, но мне тупо сейчас страшно в это залезать - можно не усепть вылезти. Так что пусть будут оба варианта.

1.

Benchmark с помощью команды `go test`.

```
package main

// Запуск из командной строки:
//         go test -bench . benchmark_sample_test.go
// В данном случае benchmark_sample_test.go - это имя файла,
// в котором находится данная программа. Имя тестируемого
// файла обязательно должно заканиваться на _test

import "testing"
```

```
func lala(s []int) {
    for i, x := range s {
        s[i] = (x-1)*x*(x+1)
    }
}

// Оценивается функция BenchmarkLala.
// Её название должно начинаться на Benchmark,
// за которым идёт название, начинающееся с большой буквы

func BenchmarkLala(b *testing.B)  {
    var s [100000]int
    for i:= 0; i < b.N; i++  {
        lala(s[:])
    }
}
```

Кроме того, приведу два пакета, в которых проведено тщательное тестирование операций копирования и вставки. Тексты предоставил Ярослав Самчук. Всё очень тщательно, повторю, протестировано, очень хорошо откомментировано, текст написан и отформатирован очень ясно. Очень хочется, чтобы дети это посмотрели вникающе. Один текст я приведу и здесь, и в печатном варианте:

Benchmark операций вставки: файл `insert_test.go`.

```
// Запуск:
//       go test -bench . insert_test.go

// файл должен иметь суффикс _test
// оцениваемые функции должны начинаться с Benchmark,
// за которым идёт название, начинающееся с большой буквы

package insert

import (
    "container/list"
    "testing"
```

```
)

var (
    slice10   = slice(10)
    slice100  = slice(100)
    slice1k   = slice(1000)
    slice10k  = slice(10000)
    slice100k = slice(100000)
    slice1m   = slice(1000000)
)

func slice(size int) []byte { return make([]byte, size) }

func BenchmarkInsertHeadSlice10(b *testing.B)      {
    benchmarkInsertSlice(b, slice10) }
func BenchmarkInsertHeadSlice100(b *testing.B)     {
    benchmarkInsertSlice(b, slice100) }
func BenchmarkInsertHeadSlice1000(b *testing.B)    {
    benchmarkInsertSlice(b, slice1k) }
func BenchmarkInsertHeadSlice10000(b *testing.B)   {
    benchmarkInsertSlice(b, slice10k) }
func BenchmarkInsertHeadSlice100000(b *testing.B)  {
    benchmarkInsertSlice(b, slice100k) }
func BenchmarkInsertHeadSlice1000000(b *testing.B) {
    benchmarkInsertSlice(b, slice1m) }

func benchmarkInsertSlice(b *testing.B, a []byte) {
    c := cap(a)
    for n := 0; n < b.N; n++ {
        // insert at i trick:
        // a = append(a[:i], append([]T{x}, a[i:]...)...)
        // could be simplified as we insert to head
        a = append([]byte{0}, a...)
        // reset a
        a = a[:c:c]
    }
}
```

```
var (
    linkedList10   = linkedList(10)
    linkedList100  = linkedList(100)
    linkedList1k   = linkedList(1000)
    linkedList10k  = linkedList(10000)
    linkedList100k = linkedList(100000)
    linkedList1m   = linkedList(1000000)
)

func linkedList(size int) *list.List {
    l := list.New()
    for i := 0; i < size; i++ {
        l.PushBack(0)
    }
    return l
}

func BenchmarkInsertHeadList10(b *testing.B)      {
    benchmarkInsertList(b, linkedList10) }
func BenchmarkInsertHeadList100(b *testing.B)     {
    benchmarkInsertList(b, linkedList100) }
func BenchmarkInsertHeadList1000(b *testing.B)    {
    benchmarkInsertList(b, linkedList1k) }
func BenchmarkInsertHeadList10000(b *testing.B)   {
    benchmarkInsertList(b, linkedList10k) }
func BenchmarkInsertHeadList100000(b *testing.B)  {
    benchmarkInsertList(b, linkedList100k) }
func BenchmarkInsertHeadList1000000(b *testing.B) {
    benchmarkInsertList(b, linkedList1m) }

func benchmarkInsertList(b *testing.B, l *list.List) {
    for n := 0; n < b.N; n++ {
        l.PushFront(0)
        // reset l
        l.Remove(l.Front())
    }
}
```

Да, здесь кроме вставки в начало слайса исполняется вставка в начало списка - list из package “container/list”, которого у нас ещё не было. Но пусть будет, вл-первых скоро он у нас появится, и рукотворный, и библиотечный, но и так всё более или менее понятно, и, самое главное, результаты хорошо показывают, что вставка в начало списка исполняется не только гораздо быстрее, но и время исполнения не зависит от размера списка, в отличие от слайсов.

Второй текст, в котором бенчмаркается операция копирования, я здесь и печатном варианте приводить не буду - в силу объёма, там больше 200 строк, но детям дать очень желательно. И настоятельно рекомендовать разобраться тщательно, да там и несложно, важна структура самого процесса тестирования скорости.

Benchmark операций копирования: файл `copy_test.go`.
Файл лежит в каталоге samples.

2.

Benchmark с помощью функции `testing.Benchmark`, аргументом которой является функция типа `func(b *testing.B)`.

```
package main

import (
    "testing"
    "fmt"
)

func lala(s []int) {
    for i, x := range s {
        s[i] = (x-1)*x*(x+1)
    }
}

func main() {
    var s [100000]int
    res:= testing.Benchmark(
        func(b *testing.B) {
            for i:= 0; i < b.N; i++  {
                lala(s[:])
```

```
            }
        })
    fmt.Println(res)
}
```

Задачи и упражнения

Ну, а что тут говорить. Берём всякие действия со слайсами и реализуем их, старясь выжать эффективность. Действий полно - и в текущем плане, и в плане предыдущего занятия. Действия сегодняшние приведены в начале, в оглавлении, действия с прошлого занятия я перечисляю ниже (только заголовки/краткие формулировки), подробности в конце концов можно заглянуть в предыдущий план:

1. Вычеркнуть i-й элемент слайса, сохраняя порядок следования остающихся элементов
2. Вставить x на i-е место.
3. Написать свою функцию copy. Сравнить не только реализации между собой, но и с функцией copy
4. Вычеркнуть i-й элемент, не обязательно сохраняя порядок следования остающихся элементов.
5. Сдвинуть циклически на k позиций влево.
6. Сдвинуть циклически на k позиций влево, не заводя дополнительных массивов/слайсов.

Если захочется раздать детям описание сегодняшних действий со слайсами, то соответствующий pdf лежит в materials: Действия со слайсами. Описание. Сегодняшние..

Да, и всё **бенчмарчить**. Обязательно.

Ну, и маловероятно, что все три варианта циклического сдвига массива на месте дети сделали. Так что это великое задание есть с нами всегда. Подробное описаие - в материалах прошлого занятия. И все эти три способа очень интересно сравнить по эффективности (а также и варианты, которые используют дополнительные слайсы/массивы, хорошо бы сравнить).