뉴·진·스

Swift와 관련된 PS 상식

김준성 2023-07-11(화)

목차

1.시간 제한

2.메모리 제한

3.자주 쓰이는 함수

4.브루트 포스 알고리즘

5.그리디 알고리즘

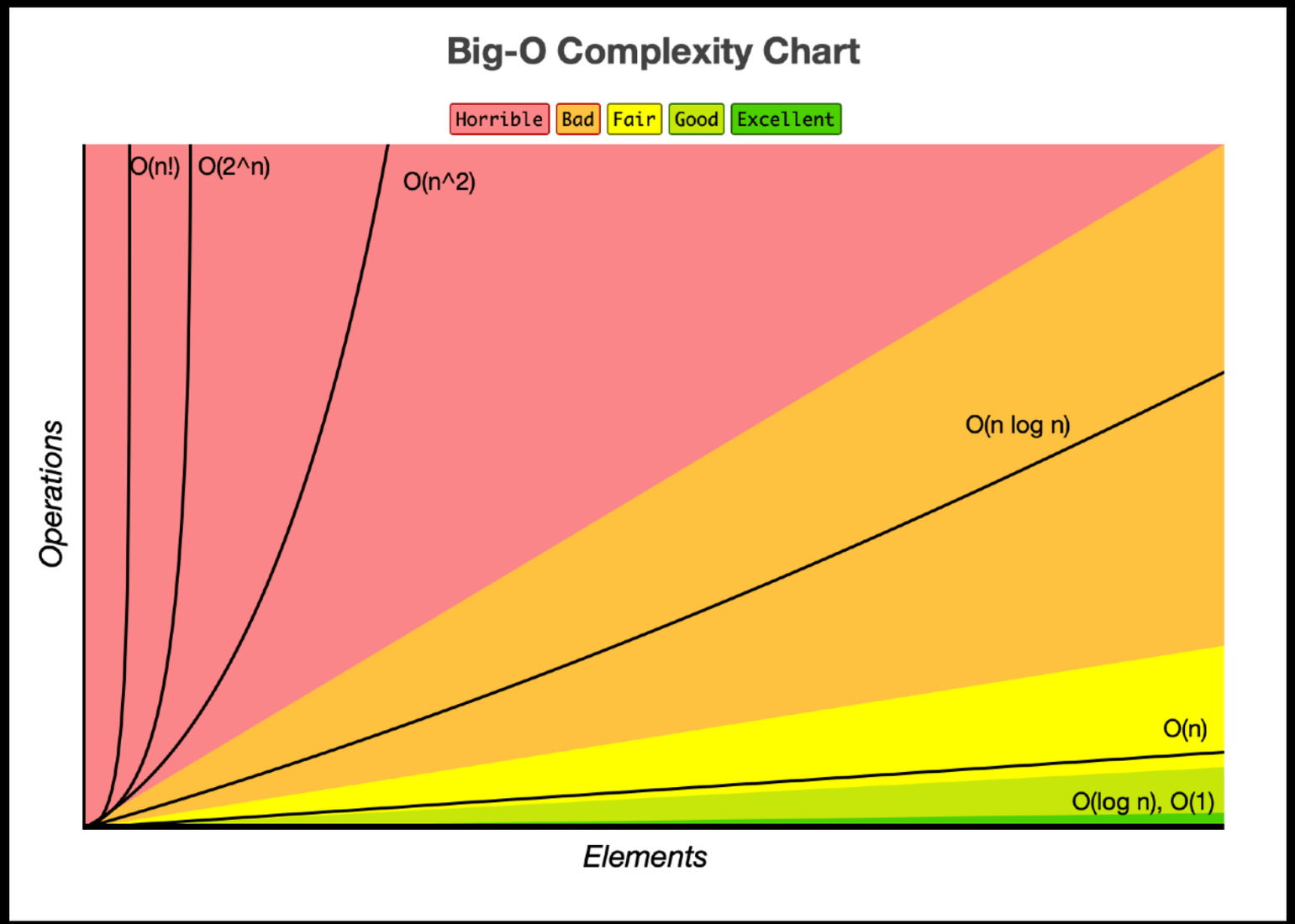
6.Live Coding

시간 제한

시간복잡도

- 컴퓨터 프로그램의 입력값과 연산 수행 시간의 상관관계를 나타내는 척도. 일반적으로는 점근 표기법을 이용해서 나타냄. (출처: 나무위키)

- 빠른 정도
  O(1) > O(log n) > O(n) > O(n log n) > O(n^2) > O(2^n) > O(n)



시간복잡도

```swift
func fooo(n: Int) {
    for i in 0..<n {
        print(i)
    }
}

func fooo2(n: Int) {
    for i in 0..<n {
        let num = i * i
        print(String(num + i))
        print(i)
        print(num)
    }
}
```

과연 각 함수들의 시간복잡도는
얼마나 될까요?

모두 다 O(N)!

시간복잡도

```
func fooo3(n: Int) {
    for _ in 0..<n {
        continue
    }

    for _ in 0..<n {
        for _ in 0..<n {
            continue
        }
    }
}
```

그렇다면 이 함수의 시간복잡도는
얼마나 될까요?

정답은 O(N^2)!

시간복잡도

```swift
func fooo3(n: Int) {
    var wantedNum = Int.random(in: 0...n)
    var leading = 0
    var trailing = n + 1

    while leading <= trailing {
        let mid = (trailing + leading) / 2
        if mid >= wantedNum {
            leading = mid + 1
        } else {
            trailing = mid - 1
        }
    }
    print(trailing)
}
```

마지막으로 이 함수의 시간복잡도는
얼마나 될까요?

정답은 O(log N)!

메모리 제한

기본적인 메모리 계산법

- 바이트 단위들의 관계
  1,000B = 1KB
  1,000,000B = 1,000KB = 1MB
  1,000,000,000B = 1,000,000KB = 1,000MB = 1GB

- Swift 정수형의 바이트
  Int8 = 1B, Int16 = 2B, Int32 = 4B, Int64 = 8B
  그렇다면 Int의 바이트는 몇일까?
  컴퓨터가 몇 비트 아키텍쳐인지 따라 다름!

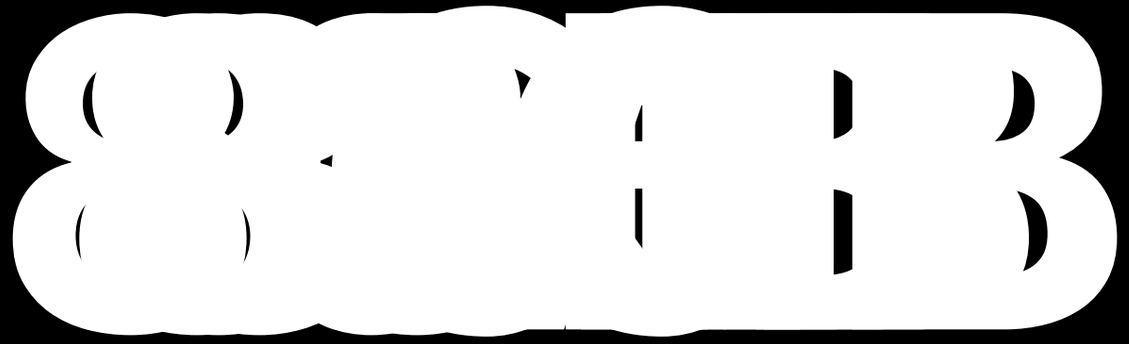
- Swift 실수형의 바이트
  Float = 4B, Double = 8B

기본적인 메모리 계산법

[Double](repeating: 0, count: 10_000_000_000)의 크기는?

Double은 8바이트!

10 _000 _000 _000



Ground Rules

강제 언래핑을 지향하자!

자주 쓰이는 함수

readLine

- 콘솔로부터 입력을 받아 변수에 저장합니다.
- 반환 타입은 String?이기 때문에 언래핑을 해줘야 합니다.
- "" 또한 입력을 받은 것이기 때문에 nil이 반환되는 경우는 백준 환경에서 없다고 생각하면 됩니다.
- 따라서 readLine을 통해 강제 언래핑을 해도 무방합니다.

map

- 배열의 요소들을 사용자가 원하는 타입으로 형변환하여 원하는 타입의 배열로 반환합니다.
- 배열과 마찬가지로 String도 Sequence 프로토콜을 채택하기 때문에 map 함수를 사용할 수 있습니다.
- 시간 복잡도는 O(N)입니다.

```
let str = "123456"
let array = str.map { Int(String($0))! }
```

```
▾ array : 6 elements collection
  Array<Int>
    0 : 1
      Int
    1 : 2
      Int
    2 : 3
      Int
    3 : 4
      Int
    4 : 5
      Int
    5 : 6
      Int
```

split

- 문자열을 특정 분리자 조건에 맞춰 쪼개서 Substring 배열로 반환합니다.
- Regex는 정규표현식을 의미하지만, 지금은 separator에 띄어쓰기 할 원하는 String을 집어넣으면 된다고 생각하시면 됩니다.
- 문자열을 split을 통해서 반환했을 경우, 배열의 각 요소는 Substring이기 때문에 map을 통해 원하는 타입으로 형변환해야 합니다.
- 시간 복잡도는 O(N)입니다.

```
let newJeans = "뉴준성의 진지한 스터디"
newJeans.split(separator: " ")
```

```
▾3 elements collection
  Array<Substring>
    0 : "뉴준성의"
    Substring
    1 : "진지한"
    Substring
    2 : "스터디"
    Substring
```

sort & sorted

- 클로저에 입력한 조건에 따라 정렬해줍니다.
- swift에 탑재된 기본 정렬 알고리즘은 introsort입니다.
- introsort의 경우 stable하기 때문에, 이전의 정렬을 보장해줍니다.
- 시간복잡도는 O(N log N)입니다.

```
var array = [10, 2, 3, 27, 7645, -1]
let array2 = array.sorted(by: <)
```

array2 : 6 elements collection
Array<Int>

0 : −1
Int

1 : 2
Int

2 : 3
Int

3 : 10
Int

4 : 27
Int

5 : 7,645
Int

```
array.sort(by: >)
```

```
var array = [10, 2, 3, 27, 7645, -1]
let array2 = array.sorted(by: <)
array.sort(by: >)
```

array : 6 elements collection
Array<Int>

0 : 7,645
Int

1 : 27
Int

2 : 10
Int

3 : 3
Int

4 : 2
Int

5 : −1
Int

sort & sorted

- stable 하다는 것은 무슨 소리고, 이전 정렬을 보장한다는 것은 또 무슨 소리일까요?


```swift
struct Person: CustomStringConvertible {
    let name: String
    let age: Int

    var description: String { "이름: \(name), 나이: \(age)" }
}

let people = [
    Person(name: "Jack", age: 83),
    Person(name: "Jack", age: 22),
    Person(name: "Jack", age: 10),
    Person(name: "Brian", age: 36),
    Person(name: "Brian", age: 22),
    Person(name: "Miles", age: 83),
    Person(name: "Miles", age: 36),
    Person(name: "Miles", age: 22),
    Person(name: "Miles", age: 10)
]
```

sort & sorted

- 우선 unstable한 heapsort를 활용하여, 오름차순 기준으로 사람들을 나이정렬한 뒤에 다시 이름정렬해보겠습니다.

```swift
var heap1 = Heap<Person> { lhs, rhs in
    return lhs.age < rhs.age
}
var heap2 = Heap<Person> { lhs, rhs in
    return lhs.name < rhs.name
}
people.forEach{ heap1.enheap($0) }

let heapsortedPeople1 = heapsort(heap1)
heapsortedPeople1.forEach { heap2.enheap($0) }
let heapsortedPeople2 = heapsort(heap2)
heapsortedPeople2.forEach { print($0) }
```

```
이름: Brian, 나이: 22
이름: Brian, 나이: 36
이름: Jack, 나이: 10
이름: Jack, 나이: 83
이름: Jack, 나이: 22
이름: Miles, 나이: 36
이름: Miles, 나이: 83
이름: Miles, 나이: 10
이름: Miles, 나이: 22
Program ended with exit code: 0
```

sort & sorted

- 이번엔 stable한 introsort를 활용하여, 오름차순 기준으로 사람들을 나이정렬한 뒤에 다시 이름정렬해보겠습니다.

```swift
let introsortedPeople1 = people.sorted(by: { (lhs, rhs) -> Bool in
    return lhs.age < rhs.age
})
let introsortedPeople2 = introsortedPeople1.sorted(by: { (lhs, rhs) -> Bool in
    return lhs.name < rhs.name

})
introsortedPeople2.forEach {
    print($0)
}
```

```
이름: Brian, 나이: 22
이름: Brian, 나이: 36
이름: Jack, 나이: 10
이름: Jack, 나이: 22
이름: Jack, 나이: 83
이름: Miles, 나이: 10
이름: Miles, 나이: 22
이름: Miles, 나이: 36
이름: Miles, 나이: 83
Program ended with exit code: 0
```

sort & sorted

- introsort가 stable하다는 것은 알겠지만, 두 번에 걸쳐서 하는 방법은 비효율적입니다. 더 나은 방법은 없을까요?

```swift
let introsortedPeople = people.sorted(by: { (lhs, rhs) -> Bool in
    if lhs.name == rhs.name {
        return lhs.age < rhs.age
    } else {
        return lhs.name < rhs.name
    }
})
introsortedPeople.forEach {
    print($0)
}
```

```
이름: Brian, 나이: 22
이름: Brian, 나이: 36
이름: Jack, 나이: 10
이름: Jack, 나이: 22
이름: Jack, 나이: 83
이름: Miles, 나이: 10
이름: Miles, 나이: 22
이름: Miles, 나이: 36
이름: Miles, 나이: 83
Program ended with exit code: 0
```

브루트 포스 알고리즘

브루트 포스 알고리즘

- 우리 말로 번역을 하면 완전 탐색이고, 직역을 하면 무지성 대입입니다.
- 즉 아무 생각 없이 모든 경우를 대입하고 연산하는 것입니다.
- 브루트 포스 알고리즘은 정형화 된 알고리즘이 아니기 때문에,
  여러 문제를 풀며 익숙해지는 수 밖에 없습니다.

[1, 2, 3, 4, 17, 13, 29, 19] 배열에서
서로 다른 숫자를 더했을 때, 20이 되는 경우의 수는?
이때 순서는 신경쓰지 않는다고 가정.

두 수의 합이 20이 되는 경우의 수는?

let array

1	2	3	4	17	13	29	19

두 수의 합이 20이 되는 경우의 수는?

let array

1	2	3	4	17	13	29	19

그리디 알고리즘

```swift
let array = [1, 2, 3, 4, 17, 13, 29, 19] // 두 수의 합이 20이 되는 경우의 수는?

var count = 0
for i in 0..<(array.count - 1) {
    for j in (i + 1)..<array.count {
        if array[i] + array[j] == 20 {
            count += 1
        }
    }
}
print(count)
```

그리디 알고리즘

그리디 알고리즘

- 그리디 알고리즘은 최대한 우선순위가 높은 값을 먼저 처리해주는 알고리즘입니다.

- 브루트 포스와 마찬가지로 정형화 된 알고리즘이 아니기 때문에, 여러 문제를 풀며 익숙해지는 수 밖에 없습니다.

```
// 5만원권, 1만원권, 5천원권, 1천원권으로 만들수 있는 최소한의 지폐 개수는?
var money = 67000
var count = 0
```

6만 7천원을 최소한의 지폐 개수로 만들어보자.

5만원

50,000 x 1 = 50,000

1만원

10,000 x 6 = 60,000

5천원

5,000 x 3 = 15,000

1천원

1,000 x 2 = 2,000

그리디 알고리즘

- 그리디 알고리즘은 최대한 우선순위가 높은 값을 먼저 처리해주는 알고리즘입니다.

- 브루트 포스와 마찬가지로 정형화 된 알고리즘이 아니기 때문에, 여러 문제를 풀며 익숙해지는 수 밖에 없습니다.

```
// 5만원권, 1만원권, 5천원권, 1천원권으로 만들수 있는 최소한의 지폐 개수는?
var money = 67000
var count = 0

count += (money / 50000)
money %= 50000
count += (money / 10000)
money %= 10000
count += (money / 5000)
money %= 5000
count += (money / 1000)
money %= 1000

print(count)
```

더 알아보기

실눈 상태

11047 동전 0

3711 학번

끝