Python Programlama

Ders 9

Ali Mertcan KOSE Msc.
amertcankose@ticaret.edu.tr

İstanbul Ticaret Üniversitesi

Program Verimliliğinin Ölçülmesi

Bilgisayarlar hızlı ve daha da hızlanıyor, peki program verimliliği neden önemlidir?

- Veri kümeleri çok büyük olabilir, Google tarafından aranan ve alınan verileri düşünün, bu nedenle basit çözümler bu kadar büyük veri kümeleri için ölçeklenmeyecektir.

Bir problem için birçok algoritma olabilir, hangisinin en verimli olduğuna nasıl karar veririz? Bir programın zaman ve mekan verimliliğini ayırın.

Bir programın aldığı süre ile bir program için gereken bellek arasında bir değiş tokuş vardır:

- Bazen daha hızlı aramak için sonuçları önceden hesaplayabilir ve depolayabiliriz

Zaman verimliliğine odaklanacağız.

Program Verimliliğinin Ölçülmesi

Örneğin, aşağıdaki işlevin çalışması ne kadar sürer sorusuna cevap vermek istiyoruz.

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

Biraz girdi ile çalışabilir ve zamanlayabiliriz, ancak sonuçlar tutarsız olacaktır, çünkü bunlar bilgisayarın hızına, Python uygulamasının verimliliğine ve girdinin değerine bağlı olacaktır.

Zamanlama programları tutarsız. . .

Hedef: Farklı algoritmaların verimliliğini değerlendirme

- Çalışma süresi algoritmalar arasında değişir.
- Çalışma süresi uygulamalar arasında farklılık gösterir.
- Çalışma süresi bilgisayarlar arasında değişir.
- Çalışma süresi, küçük girdilere dayalı olarak tahmin edilemez.

Zaman, farklı girdiler için değişir, ancak girdiler ve zaman arasında bir ilişki ifade edemeyiz.

Program Verimliliğinin Ölçülmesi

Program tarafından yürütülen temel adımların sayısı açısından zamanı ölçmenin daha iyi bir yolu.

Adım, sabit bir süre alan bir işlemdir, örneğin

- Atama (bir değişkeni bir nesneye bağlama).
- Karşılaştırma yapmak.
- Aritmetik işlem yürütme.
- Bellekteki bir nesneye erişme.

Sayım İşlemleri

- Bu adımların sürekli zaman aldığını varsayalım
  - Matematiksel İşlemler
  - Karşılaştırmalar
  - Değerlendirmeler
  - Bellekte nesnelere erişmek
- Ardından, girdi boyutunun işlevi olarak yürütülen işlem sayısını sayın.

Sayım İşlemleri

```
def c_to_f(c):
return c*9.0/5 +32
def mysum(x):
total=0
for i in range(x):
total+=i
return total
```

Program Verimliliğinin Ölçülmesi

Farklı girdiler bir programın verimliliğini değiştirir. Bir programın karmaşıklığını değerlendirirken, dikkate alınması gereken 3 geniş durum vardır.

- En iyi durum: Girişler olumlu olduğunda algoritmanın çalışma süresi.
- En kötü durum: Çalışma süresi, belirli bir boyuttaki tüm olası girişler üzerindeki maksimum çalışma süresidir.
- Ortalama durum: Çalışma süresi, belirli bir boyuttaki tüm olası girişler üzerindeki ortalama çalışma süresidir.

Programcılar genellikle en kötü durumu göz önünde bulundururlar çünkü bu, çalışma süresinin üst sınırını sağlar.

Program Karmaşıklığı

Faktöriyel programın en kötü durum çalışma süresine bakalım

```
def fact(n):
""""Assumes n is an int and n >= 0""" "
answer = 1
while n >= 1:
answer *= i
n-= 1
return answer
```

Program Karmaşıklığı

Adım sayısı $2 + 5n$

- 2:1 İlk atama için ve geri dönüş için 1 e döner.
- 5n:
  - Bu süre zarfında test için 1,
  - Döngüdeki ilk ifade için 2 (atama ve çarpma)
  - Döngüdeki ikinci ifade için 2 (atama ve çıkarma)

Program Karmaşıklığı

Faktöriyel örneğinde, n 1000'e eşitse kaç adım vardır?

Cevap 5002 adımdır, gördüğümüz gibi 5000 ile 5002 adım arasında bir fark yoktur, n büyüdükçe 5n ile 5n2 arasındaki fark önemsizdir.

Bu nedenle, sonucu etkilemedikleri için genellikle toplamsal sabitleri göz ardı edebiliriz.

Büyük ‘O’ Gösterimi (Asimptotik Gösterim)

Büyük O gösterimi, bir algoritmanın çalışma süresi ile girdilerinin boyutu arasındaki ilişkiyi resmi olarak tanımlamanın bir yoludur. Bir fonksiyonun büyümesinin üst sınırını vermek için kullanılır. Girdilerinin boyutu sonsuza yaklaştıkça bir algoritmanın karmaşıklığını açıklar.

Asymptotic Notation-Örnek

def f(x):

“ “ “Assume x is an int $>$ 0”“ ”

ans $=$ 0

Asymptotic Notation-Örnek

```
#Loop that takes constant time
```

for i in range(1000):

ans += 1

print('Number of additions so far', ans)

Asymptotic Notation-Örnek

```
#Loop that takes time x
```

for i in range(x):

ans += 1

print('Number of additions so far', ans)

Asymptotic Notation-Örnek

```
#Nested loops take time x**2
```

for i in range(x):

for j in range(x):

ans += 1

ans += 1

print('Number of additions so far', ans)

return ans

Asymptotic Notation-Örnek

Running time: $1000 + x + 2x^2$

f(10)

Number of additions so far 1000

Number of additions so far 1010

Number of additions so far 1210

f(1000)

Number of additions so far 1000

Number of additions so far 2000

Number of additions so far 2002000

Yukarıdaki uygulamalar, x'in küçük değerleri için sabit terimin baskın olduğunu, x büyük olduğunda, baskın olan son terimin (iç içe döngü) baskın olduğunu göstermektedir.

Asimptotik karmaşıklığı hesaplama kuralları

Birden fazla terimin toplamında en büyük büyüme oranına sahip terimi tutun ve diğerlerini bırakın. Kalan terim bir çarpımsa, sabitleri bırakın. Bu kurala göre, f()'nin çalışma süresi $O(x^2)$'dir, çünkü örnek çalışmalarda gösterildiği gibi, girdiler arttıkça en büyük büyüme oranına sahiptir.

Büyüme Emirleri

Hedefler:

- Giriş çok büyük olduğunda programın verimliliğini değerlendirmek istiyoruz.
- Giriş boyutu büyüdükçe programın çalışma süresinin büyümesini sağlamak istiyoruz.
- Büyümeye mümkün olduğunca sıkı bir üst sınır koymak istiyoruz.
- kesin olmayan büyümenin kesin sırası olması gerekmez.
- Çalışma süresindeki en büyük faktörlere bakacağız (programın hangi bölümünün çalıştırılması en uzun sürecek?)
- Bu nedenle, genellikle en kötü durumda, girdi boyutunun fonksiyonu olarak büyümeye sıkı bir üst sınır istiyoruz.

Basitleştirme Örnekleri

- Düşme sabitleri ve çarpma faktörleri
- Baskın terimlere odaklanın

$O(n^2)$: $n^2 + 2n + 2$

$O(n^2)$: $n^2 + 100000n + 3^{1000}$

$O(n)$: $\log(n) + n + 4$

$O(n \log n)$: $0.0001*n*\log(n) + 300n$

$O(3^n)$: $2n^{30} + 3^n$

Figure 1: Birleştirme Örnekleri

En Yaygın Büyük O Ölçümleri

N'nin girdi sayısı olduğu varsayılarak, aşağıdaki ölçüler kullanılır:

O(1): sürekli çalışan

O(log n): logaritmik çalışma süresi, her seferinde sorunu yarı yarıya azaltır.

O(n): Doğrusal çalışma süresi, basit yinelemeli veya özyinelemeli programlar.

O(n log n):Doğrusal çalışma süresini günlüğe kaydet.

O(n^k):Polinom çalışma süresi (k bir sabittir).

O(c^n):üstel çalışma süresi.

Sabit Karmaşıklık O(1)

Asimptotik karmaşıklık, girdilerin boyutundan bağımsızdır. Bir python listesinin uzunluğunu bulma, iki kayan noktalı sayıyı çarpma, bir mesaj görüntüleme, bir dizideki bir öğeye erişme örneği. Sabit karmaşıklık, döngü olmadığı, özyinelemeli çağrı olmadığı anlamına gelmez, ancak yineleme/özyinelemeli çağrı sayısının girdinin boyutundan bağımsız olduğu anlamına gelir.

Logaritmik Karmaşıklık O(log(n))

Bir örnek, ayrıntılı olarak bahsedeceğimiz ikili arama algoritmasıdır.

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return res
```

Figure 2: Logaritmik Karmaşıklık

Lineer Karmaşıklık O(n)

Her öğeye sıfırdan büyük sabit sayıda bir dizide erişin. Örneğin, bir öğe için bir listede arama yapmak veya ondalık basamaklardan oluştuğu varsayılan bir dizenin karakterlerini eklemek.

```
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

Figure 3: Lineer Karmaşıklık

Özyinelemeli Faktöriyel O(n)

Doğrusal karmaşıklık ile aynı. Döngü yoktur, ancak özyinelemeli çağrıların sayısı karmaşıklık düzeyini belirler. Bellek kullanımı daha fazla olmasına rağmen, burada karmaşıklık zamanlamayı yansıtır.

Log-Lineer Karmaşıklık O(nlog(n))

Her iki terim de iki terimin çarpımındaki girdilerin büyüklüğüne bağlıdır. Örnek, ayrıntılı olarak inceleyeceğimiz mergesort.

Polynomial Karmaşıklık O(n^k)

Karmaşıklık, girdilerinin boyutunun karesi olarak büyür. En yaygın polinom algoritmaları ikinci dereceden O(n^2) Aşağıdaki algoritma, bir listenin ikincisinin alt kümesi olup olmadığını belirler.

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Figure 4: Polynomial Karmaşıklık

Polynomial Karmaşıklık O(n^k)

İki listenin kesişimini bulun.

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

Figure 5: Polynomial Karmaşıklık

ÜstelKarmaşıklık $O(c^n)$

Birçok önemli sorun üsteldir. Karmaşıklık, sorunun boyutunun bir üssü olarak büyüdüğünden, üstel olarak karmaşık programlar yararlı olmayabilir. Bunun yerine daha az karmaşık algoritmalar bulunabilir.

Büyüme Eğri Tipleri

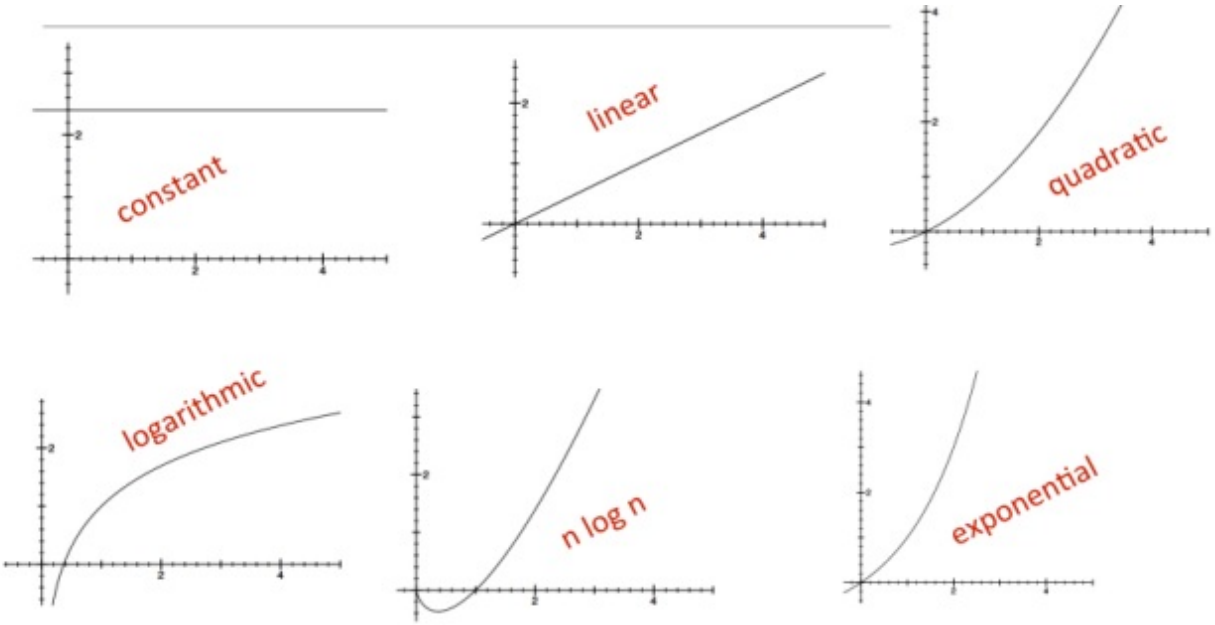

Figure 6: Büyüme Eğri Tipleri