Logaritma

Bagaimana memikirkannya, terutama dalam wawancara pemrograman dan desain algoritma

Apa *arti* logaritma

Inilah yang ditanyakan logaritma:

"Kekuatan apa yang harus kita tingkatkan pangkalan ini, untuk mendapatkan jawaban ini?"

Jadi jika kita mengatakan:

.log10100 \log_{10}{100}

10 disebut *basis* (masuk akal — ada di bawah). Pikirkan 100 sebagai "jawaban." Itulah yang kami ambil *lognya*. Jadi ungkapan ini akan diucapkan "basis log 10 dari 100."

Dan semua itu berarti adalah, "Kekuatan apa yang kita butuhkan untuk meningkatkan basis ini (1010) untuk, untuk mendapatkan jawaban ini (100100)?"

10x=100 10^x = 100

Apa xx memberi kita hasil dari 100100? Jawabannya adalah 22:

102=100 10^2 = 100

Jadi kita bisa mengatakan:

.log10100=2 \log_{10}{100} = 2

Bagian "jawaban" bisa dikelilingi oleh tanda kurung, atau tidak. Jadi kita bisa mengatakan .log10(100)\log_{10}{(100)} atau .log10100\log_{10}{100}. Salah satunya baik-baik saja.

Untuk apa logaritma *digunakan*

Hal utama yang kami gunakan untuk logaritma adalah **Memecahkan untuk xx Kapan xx berada dalam eksponen**.

Jadi jika kita ingin menyelesaikan ini:

10x=100 10^x = 100

Kita perlu membawa xx turun dari eksponen entah bagaimana. Dan logaritma memberi kita trik untuk melakukan itu.

Kami mengambil .log10\log_{10} dari kedua sisi (kita bisa melakukan ini — kedua sisi persamaan masih sama):

.log1010x=.log10100 \log_{10}{10^x} = \log_{10}{100}

Sekarang sisi kiri bertanya, "kekuatan apa yang harus kita tingkatkan 1010 untuk mendapatkan 10x10^x?" Jawabannya, tentu saja, adalah xx. Jadi kita bisa menyederhanakan seluruh sisi kiri itu menjadi hanya "xx":

x=.log10100 x = \log_{10}{100}

Kami telah menarik xx turun dari eksponen!

Sekarang kita hanya perlu mengevaluasi sisi kanan. Kekuatan apa yang harus kita tingkatkan 1010 untuk mendapatkan 100100? Jawabannya masih 22.

x=2 x = 2

Begitulah cara kita menggunakan logaritma untuk menarik variabel ke bawah dari eksponen.

Aturan logaritma

Ini sangat membantu jika Anda mencoba melakukan beberapa hal aljabar dengan log.

Penyederhanaan: .logb(bx)=x\log_{b}{(b^x)} = x . . . *Berguna untuk menurunkan variabel dari eksponen.*

Perkalian: .logb(x∗y)=.logb(x)+.logb(y)\log_{b}{(x*y)} = \log_{b}{(x)} + \log_{b}{(y)}

Divisi: .logb(x/y)=.logb(x)−.logb(y)\log_{b}{(x/y)} = \log_{b}{(x)} - \log_{b}{(y)}

Kekuatan: .logb(xy)=y∗.logb(x)\log_{b}{(x^y)} = y * \log_{b}{(x)}

Perubahan basis: .logb(x)=.logc(x).logc(b)\log_{b}{(x)} = \frac{\log_{c}{(x)} }{\log_{c}{(b)} } . . . *Berguna untuk _mengubah basis* logaritma dari bb ke cc._

Di mana log muncul dalam algoritma dan wawancara

"Berapa kali kita harus menggandakan 1 sebelum kita sampai nn" adalah pertanyaan yang sering kita tanyakan pada diri sendiri dalam ilmu komputer. Atau, setara, **"Berapa kali kita harus membagi nn setengah untuk kembali ke 1?"**

Dapatkah Anda melihat bagaimana itu adalah pertanyaan yang sama? Kami hanya pergi ke arah yang berbeda! Dari nn ke 1 dengan membagi dengan 2, atau dari 1 ke nn dengan mengalikan dengan 2. Either way, itu *adalah jumlah yang sama kali* bahwa kita harus melakukannya.

Jawaban untuk kedua pertanyaan ini adalah .log2n\log_{2}{n}.

Tidak apa-apa jika belum jelas mengapa itu benar. Kami akan memperolehnya dengan beberapa contoh.

Logaritma dalam pencarian biner (mis. 1)

Ini muncul dalam biaya waktu **pencarian biner**, yang merupakan algoritma untuk menemukan nomor target dalam daftar *yang diurutkan*. Prosesnya berjalan seperti ini:

1. **Mulailah dengan angka tengah: apakah lebih besar atau lebih kecil dari angka target kita?** Karena daftar diurutkan, ini memberi tahu kita apakah target akan berada di bagian *kiri* atau bagian *kanan* daftar kita.
2. **Kami telah secara efektif membagi masalah menjadi dua**. Kita dapat "mengesampingkan" seluruh setengah dari daftar yang kita tahu tidak mengandung jumlah target.
3. **Ulangi pendekatan yang sama (mulai dari tengah) pada masalah setengah ukuran baru**. Kemudian lakukan lagi dan lagi, sampai kita menemukan nomor atau "mengesampingkan" seluruh rangkaian.

Dalam kode:

```
  def binary_search(target, nums):
    """See if target appears in nums"""
    # We think of floor_index and ceiling_index as "walls" around
    # the possible positions of our target so by -1 below we mean
    # to start our wall "to the left" of the 0th index
    # (we *don't* mean "the last index")
    floor_index = -1
    ceiling_index = len(nums)

    # If there isn't at least 1 index between floor and ceiling,
    # we've run out of guesses and the number must not be present
    while floor_index + 1 < ceiling_index:
        # Find the index ~halfway between the floor and ceiling
        # We use integer division, so we'll never get a "half index"
        distance = ceiling_index - floor_index
        half_distance = distance // 2
        guess_index = floor_index + half_distance
```

```
        guess_value = nums[guess_index]
        if guess_value == target:
            return True

        if guess_value > target:
            # Target is to the left, so move ceiling to the left
            ceiling_index = guess_index
        else:
            # Target is to the right, so move floor to the right
            floor_index = guess_index

    return False
```

Jadi berapa biaya waktu pencarian biner? Satu-satunya bagian non-konstan dari biaya waktu kami adalah berapa kali loop while kami berjalan. Setiap langkah dari while loop kami memotong rentang (ditentukan oleh floor_index dan ceiling_index) menjadi dua, sampai jangkauan kami hanya memiliki satu elemen tersisa.

Jadi pertanyaannya adalah, "berapa kali kita harus membagi ukuran daftar asli kita (nn) menjadi dua sampai kita turun ke 1?"

$$n * 12 * 12 * 12 * 12 * \ldots = 1 n * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \ldots = 1$$

Berapa banyak 12\frac{1}{2}Apakah ada? Kami belum tahu, tapi kami bisa menghubungi nomor itu xx:

n*(12)x=1 n * (\frac{1}{2})^x = 1

Sekarang kita memecahkan untuk xx:

$$n * 1x2x\=1n*frac1^x2^x = 1n * 12x\=1n*frac12^x = 1n2x\=1fracn2^x = 1n\=2xn = 2^x$$

Sekarang untuk mendapatkan xx dari eksponen itu! Kami akan menggunakan trik yang sama seperti terakhir kali.

Ambil .log2\log_{2} dari kedua belah pihak ...

.log2n=.log22x \log_{2}{n} = \log_{2}{2^x}

Sisi kanan bertanya, "kekuatan apa yang harus kita tingkatkan 22 untuk, untuk mendapatkan 2x2^x?" Yah, itu saja xx.

.log2n=x \log_{2}{n} = x

Jadi begitulah. Total biaya waktu pencarian biner adalah O(.log2n)O(\log_{2}{n}).

Logaritma dalam penyortiran (mis. 2)

Biaya penyortiran O(n.log2n)O(n\log_{2}{n}) waktu *secara umum*. Lebih khusus lagi, O(n.log2n)O(n\log_{2}{n}) adalah runtime *kasus terburuk* terbaik yang bisa kita dapatkan untuk penyortiran.

Itulah runtime terbaik kami untuk penyortiran *berbasis perbandingan*. Jika kita dapat mengikat erat kisaran angka yang mungkin dalam daftar kita, kita dapat menggunakan peta hash melakukannya di O(n)O(n) waktu dengan menghitung urutan.

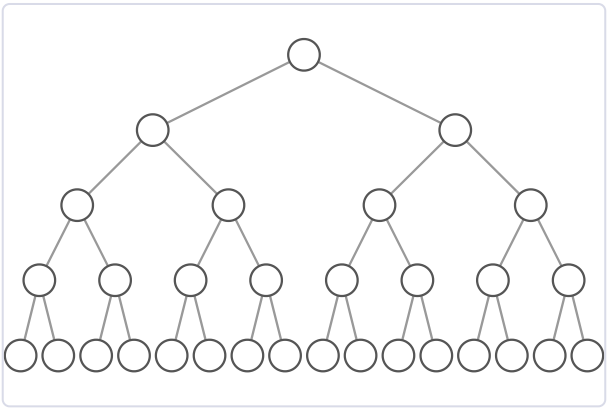
Cara termudah untuk melihat mengapa adalah dengan melihat merge sort. Dalam urutan gabungan, idenya adalah membagi daftar menjadi dua, mengurutkan dua bagian, dan kemudian menggabungkan dua bagian yang diurutkan menjadi satu keseluruhan yang diurutkan. Tapi bagaimana kita mengurutkan dua bagian? Nah, kami membaginya menjadi dua, mengurutkannya, dan menggabungkan bagian yang diurutkan ... dan sebagainya.

```python
  def merge_sort(list_to_sort):
    # Base case: lists with fewer than 2 elements are sorted
    if len(list_to_sort) < 2:
        return list_to_sort

    # Step 1: divide the list in half
    # We use integer division, so we'll never get a "half index"
    mid_index = len(list_to_sort) // 2
    left  = list_to_sort[:mid_index]
    right = list_to_sort[mid_index:]

    # Step 2: sort each half
    sorted_left  = merge_sort(left)
    sorted_right = merge_sort(right)

    # Step 3: merge the sorted halves
    sorted_list = []
    current_index_left = 0
    current_index_right = 0

    # sortedLeft's first element comes next
    # if it's less than sortedRight's first
    # element or if sortedRight is exhausted
    while len(sorted_list) < len(left) + len(right):
        if ((current_index_left < len(left)) and
                (current_index_right == len(right) or
                 sorted_left[current_index_left] <
sorted_right[current_index_right])):
            sorted_list.append(sorted_left[current_index_left])
            current_index_left += 1
        else:
            sorted_list.append(sorted_right[current_index_right])
            current_index_right += 1
    return sorted_list
```

Jadi berapa total biaya waktu kami? O(n.log2n)O(n\log_{2}{n}). Si .log2n\log_{2}{n} berasal dari berapa kali kita harus memotong nn setengah untuk turun ke subdaftars hanya 1 elemen (kasus dasar kami). Tambahan nn berasal dari biaya waktu penggabungan semua nn item bersama-sama setiap kali kita menggabungkan dua subdaftaryang diurutkan.

Logaritma dalam pohon biner (kel. 3)

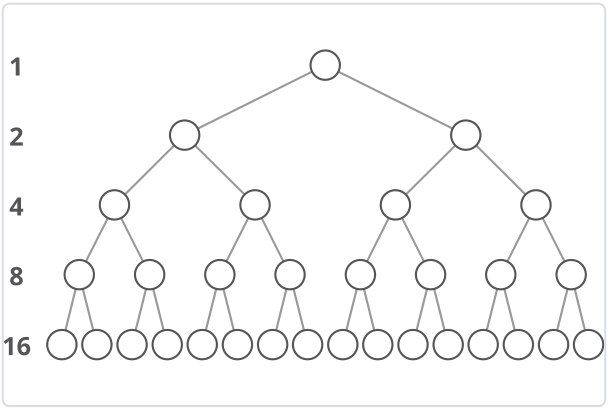
Dalam pohon biner, setiap node memiliki dua atau lebih sedikit anak.



Pohon di atas istimewa karena setiap "tingkat" atau "tingkat" pohon sudah penuh. Tidak ada celah. Kami menyebut pohon seperti itu "**sempurna**."

Satu pertanyaan yang mungkin kita tanyakan adalah, jika ada nn node secara total , berapa *tinggi* pohon (hh)? Dengan kata lain, berapa *level* yang dimiliki pohon itu?

Jika kita menghitung jumlah node *pada setiap level*, kita dapat melihat bahwa itu berturut-turut *berlipat ganda* saat kita pergi:



Itu membawa kembali menahan diri kita, "berapa kali kita harus menggandakan 1 untuk sampai ke nn." Tapi kali ini, kami tidak menggandakan 1 untuk sampai ke nn; nn adalah *jumlah total node di pohon*. Kami menggandakan 1 sampai kami mencapai . . . jumlah node pada *tingkat terakhir* pohon.

Berapa banyak node yang dimiliki level terakhir? Lihat kembali diagram di atas.

Tingkat terakhir memiliki sekitar *setengah* dari jumlah total node di pohon. Jika Anda menjumlahkan jumlah node pada semua level kecuali yang terakhir, Anda mendapatkan *sekitar* jumlah node pada level terakhir — 1 lebih sedikit.

$$1 + 2 + 4 + 8 = 151 + 2 + 4 + 8 = 15$$

Rumus yang tepat untuk jumlah node pada tingkat terakhir adalah:

$$n + 12fracn + 12$$

Dari mana +1 itu berasal?

Jumlah node di pohon biner sempurna kita selalu ganjil. Kita tahu ini karena level pertama selalu memiliki 1 node, dan level lainnya selalu memiliki jumlah node genap. Menambahkan banyak angka genap selalu memberi kita angka genap, dan menambahkan 1 ke hasil itu selalu memberi kita angka ganjil.

Mengambil setengah dari angka ganjil memberi kita pecahan. Jadi jika level terakhir memiliki *tepat* setengah dari kami nn node, itu harus memiliki "setengah node." Tapi itu bukan apa-apa.

Sebaliknya, ia memiliki versi "dibulatkan" dari setengah dari ganjil kami nn Node. Dengan kata lain, ia memiliki setengah yang *tepat* dari jumlah node yang lebih besar dan *genap* n+1n+1. Maka n+12\frac{n+1}{2}

Jadi *tinggi badan* kita (hh) kira-kira "berapa kali kita harus menggandakan 1 untuk sampai ke n+12\frac{n+1}{2}." Kita dapat mengutarakan ini sebagai logaritma:

$$h \approx .log2(n + 12)happroxlog\_2(fracn + 12)$$

Satu penyesuaian: Pertimbangkan pohon 2 tingkat yang sempurna. Ada 2 level secara keseluruhan, tetapi "berapa kali kita harus menggandakan 1 untuk mencapai *2*" hanya *1*. Tinggi badan kita sebenarnya *satu lebih banyak dari* jumlah penggandaan kita. Jadi kami menambahkan 1:

$$h\=.log2(n + 12) + 1h = log\_2(fracn + 12) + 1$$

Kita dapat menerapkan beberapa aturan logaritma kita untuk menyederhanakan ini:

$$h\=.log2(n + 12) + 1h = log\_2(fracn + 12) + 1h\=.log2(n + 1)-.log2(2) + 1h = log\_2(n + 1) - log\_2(2) + 1h\=.lo$$

Konvensi dengan basis

Terkadang orang tidak menyertakan basis. Dalam ilmu komputer, biasanya tersirat bahwa basisnya adalah 2. Jadi .logn\log{n} umumnya berarti .log2n\log_{2}{n}.

Beberapa orang mungkin ingat bahwa dalam kebanyakan matematika lain, basis yang tidak ditentukan tersirat menjadi 10. Atau terkadang konstanta khusus ee. (Jangan khawatir jika Anda tidak tahu apa ee adalah.)

Ada notasi khusus untuk basis log 2 yang terkadang digunakan: Lg\lg. Jadi bisa kita katakan Lgn\lg{n}atau nLgnn\lg{n} (yang banyak muncul dalam penyortiran). Kami banyak

menggunakan notasi ini pada Interview Cake, tetapi perlu dicatat bahwa tidak semua orang menggunakannya.

Beberapa orang mungkin tahu ada notasi khusus yang serupa untuk basis log ee: Ln\ln (diucapkan "log alami").

Dalam notasi O besar, basis dianggap konstan. Jadi orang biasanya tidak memasukkannya. Orang biasanya mengatakan O(.logn)O(\log{n})tidak O(.log2n)O(\log_{2}{n}),

Tetapi orang mungkin masih menggunakan notasi khusus Lgn\lg{n}, seperti pada O(Lgn)O(\lg{n}). Ini menyelamatkan kita dari keharusan menulis "o" :)