Divide-and-conquer

Serge Kruk

September 22, 2021

Divide-and-conquer

- A very common design technique.
- Appears in many forms, from simple to absurdly complex.
- General idea : split an object into pieces (most often two) and process each piece separately. Sometimes, this allows us to ignore a piece.

Fill-in this table

Assuming the first column is the runtime, in microseconds (10^{-6} seconds), what is the largest n processed in the stated time?

Runtime/Duration	1 second	1 minute	1 hour	1 day	1 month	1 ye
$\log_2 n$						
$\sqrt{n}$						
n						
$n \log_2 n$						
n^2						
n^3						
2^n						
$n!$						

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.
- $f(n) = \sqrt{n}$ and 1 second, or 16^6 micro seconds

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.
- $f(n) = \sqrt{n}$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $\sqrt{n} \leq 10^6$ or $n = 10^{12}$.

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.
- $f(n) = \sqrt{n}$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $\sqrt{n} \leq 10^6$ or $n = 10^{12}$.
- $f(n) = n^3$ and 1 second, or 16^6 micro seconds

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.
- $f(n) = \sqrt{n}$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $\sqrt{n} \leq 10^6$ or $n = 10^{12}$.
- $f(n) = n^3$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n^3 \leq 10^6$ or $n = 10^2$.

Could compute all by hand

- $f(n) = n$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n \leq 10^6$ or $n = 10^6$.
- $f(n) = \sqrt{n}$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $\sqrt{n} \leq 10^6$ or $n = 10^{12}$.
- $f(n) = n^3$ and 1 second, or 16^6 micro seconds
  - Need largest n such that $n^3 \leq 10^6$ or $n = 10^2$.
- We could automate these computations...

Writing code to do the heavy lifting

```
from math import sqrt
times = [1e6, 60*1e6, 60*60*1e6, 24*60*60*1e6, 30*24*60*60*1e6]
functions = [lambda n: sqrt(n), lambda n : n, lambda n : n*log(n,2)]
print('...')
for f in functions:
  large_n = []
  for t in times:
    n = 1
    while f(n) < t:
      n = n+1
    large_n.append(n-1)
  for n in large_n:
    print("{0:10.1e}".format(n), end='')
  print('')
```

- Problem: much too slow. We search the space as an ant crawling.
- Solution: divide the search space in half and focus on the "right" half.

Two-stages approach

- First bracket the solution. (Find a, b such that $n \in [a, b]$).
- Then split the interval in half and check which half contains n.
- Adjust the interval by changing a or b. (Move a up or b down.)
- Repeat until the interval is small enough for your needs.

First stage, bracketing

How about doubling the size of the interval? Jumbled version. Put in order.

```
a,b = 1,2
a,b = b,2*b
def bracket(f, t):
return a,b
while f(b) < t:
```

First stage, bracketing

- How about doubling the size of the interval?
-
```
def bracket(f, t):
  a,b = 1,2
  while f(b) < t:
    a,b = b,2*b
  return a,b
```

Testing bracketing

Simple functions function.

```
bracket(lambda n : n, 20)
```

16 32

Second stage, zooming in

Once we have a bracket, we look in the correct half. Rinse and repeat. Jumbled version.

```
def find_largest(f, l, h, t):
elif f(m) > t:
else:
h = m-1
if f(m) < t:
l = m+1
m = l+(h-l)//2
return h
return m
while l < h:
```

Second stage, zooming in

- Once we have a bracket, we look in the correct half. Rinse and repeat.
-
```python
def find_largest(f, l, h, t):
  while l < h:
    m = l+(h-l)//2
    if f(m) < t:
      l = m+1
    elif f(m) > t:
      h = m-1
    else:
      return m
  return h
```

Testing zooming in

Same linear function

```
find_largest(lambda n: n, 64, 128, 100)
```

```
100
```

Putting it all together

```
from math import sqrt,log, factorial
times = [1e6, 60*1e6, 60*60*1e6, 24*60*60*1e6,
         30*24*60*60*1e6, 365*30*24*60*60*1e6]
functions =[lambda n: sqrt(n), lambda n : n, lambda n : n*log(n
            lambda n : n*n, lambda n : n*n*n, lambda n : 2**n,
            lambda n : factorial(n)]
for f in functions:
  large_n = []
  for t in times:
    a,b = bracket(f,t)
    n = find_largest(f,a,b,t)
    large_n.append(n)
  for n in large_n:
    print("{0:10.2e}".format(n), end='')
  print('')
```

```
1.00e+12  3.60e+15  1.30e+19  7.46e+21  6.72e+24  8.95e+29
1.00e+06  6.00e+07  3.60e+09  8.64e+10  2.59e+12  9.46e+14
6.27e+04  2.80e+06  1.33e+08  2.76e+09  7.19e+10  2.14e+13
```

Final table

1.00e+12	3.60e+15	1.30e+19	7.46e+21	6.72e+24	8.95e+29
1.00e+06	6.00e+07	3.60e+09	8.64e+10	2.59e+12	9.46e+14
6.27e+04	2.80e+06	1.33e+08	2.76e+09	7.19e+10	2.14e+13
1.00e+03	7.74e+03	6.00e+04	2.94e+05	1.61e+06	3.08e+07
1.00e+02	3.91e+02	1.53e+03	4.42e+03	1.37e+04	9.82e+04
2.00e+01	2.50e+01	3.20e+01	3.60e+01	4.20e+01	4.90e+01
9.00e+00	1.10e+01	1.30e+01	1.30e+01	1.60e+01	1.80e+01

Analysis

What do we want to count?

Analysis

- First loop

Analysis

- First loop
  - The variable `b` takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$

Analysis

- First loop
  - The variable `b` takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times

Analysis

- First loop
  - The variable b takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times
  - For $b \approx f^{-1}(t)$

Analysis

- First loop
  - The variable b takes on values 1,2,4,8,16,32,64,128,256, . . . until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times
  - For $b \approx f^{-1}(t)$
- Second loop

Analysis

- First loop
  - The variable `b` takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times
  - For $b \approx f^{-1}(t)$
- Second loop
  - We divide the interval $[b/2, b]$ in half until it is of length 1.

Analysis

- First loop
  - The variable `b` takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times
  - For $b \approx f^{-1}(t)$
- Second loop
  - We divide the interval $[b/2, b]$ in half until it is of length 1.
  - We do this $\log_2(b - b/2)$ times

Analysis

- First loop
  - The variable b takes on values 1,2,4,8,16,32,64,128,256, ... until $t \in [f(b/2), f(b)]$
  - We do this doubling $\log_2 b$ times
  - For $b \approx f^{-1}(t)$
- Second loop
  - We divide the interval $[b/2, b]$ in half until it is of length 1.
  - We do this $\log_2(b - b/2)$ times
- Therefore runtime is proportional to $f^{-1}(t)$

Discuss

Problem: Given an increasing function $f(x)$, defined for non-negative x, and a number $T > f(0)$, find a number $z \in [0, \infty[$, (assume one exists), such that $f(z) \approx T$. (You can call f, though it is expensive, but you cannot look at it.)

```
def b(f,T):

    return z
```

Divide-and-conquer!

```python
def solve (f,T,tolerance=1):
    a,b = 0,1
    fa, fb = f(a)-T, f(b)-T
    while fa*fb > 0:
        a,b = b,2*b
        fa,fb = fb,f(b)-T
    c = (a+b)/2.0
    fc = f(c)-T
    while abs(fc)> tolerance:
        if fc > 0:
            b = c
        else:
            a = c
        c = (a+b)/2
        fc = f(c)-T
    return c,fc
```

A bit of algebra

Worst-case analysis

$$
\begin{aligned}
T(b) &= \log_2 b + \log_2(b - \frac{b}{2}) \\
&\in O(\log_2(b)) + O(\log_2(\frac{b}{2})) \\
&\in O(\log_2(b))
\end{aligned}
$$

And $b = f^{-1}(T)$

- Therefore, overall runtime is $O(\log_2 f^{-1}(T))$.
- Note carefully that this multiplies the cost of calling `f`.
- Note also the dependence on the input parameters `f` and `T`.

Interlude

What is the relation between the following:

- An integer n.
- The value $\log_2 n$.
- The number of bits of the binary representation of n.

Interlude

n	Bin	bits	log n
1	1	1	0
2	10	2	1
3	11	2	1
4	100	3	2
5	101	3	2
6	110	3	2
7	111	3	2
8	1000	4	3
9	1001	4	3

- The number of bits required to express n is $\lfloor \log_2 n \rfloor + 1$.
- Integer division by 2 is a right shift.

Discuss

Given an array of integers a and an integer e, find the position of e in a, if present.

Linear search

Jumbled version. Put in order.

```
def s(a,e):
for i in range(len(a)):
if a[i] == e:
return -1
return i
```

Linear search

```python
def s(a,e):
    for i in range(len(a)):
        if a[i] == e:
            return i
    return None
```

Correctness

LI: When we enter the loop with `i` at value `k` we know that the element `e` is not in `a[0..k-1]`

In the loop we either find `e` at position `k` and return or else we know that `e` is not in `a[0..k]` and go back to the top of the loop. We end at the last element of the array, therefore we terminate.

Runtime

- $O(n)$
- Can we do better? NO! We may have to read every element of the array.
- We have the optimal algorithm.

Discuss

Given a, an array of integers, sorted in ascending order and an integer e, find the position of e in a, if present.

Binary search (Class project)

Return position of element e in array a or return -1

```
def bsearch(a, e)
```

Binary search

```python
def bsearch(a,e):
    l,h = 0,len(a)-1
    while l <= h:
        m = (l+h)//2
        if a[m] == e:
            return m
        elif a[m] < e:
            l = m+1
        else:
            h = m-1
    return None
```

Test

```
a=[-10, -3, 0, 1, 3, 10]
[bsearch(a,-10), bsearch(a, 0), bsearch(a, 2), bsearch(a,10)]
```

0 2 -1 5

Correctness

At the start the interval `[l,h]` encompasses the whole array and it is reduced at every step. Therefore, the algorithm terminates.
LI: if element `e` is in the array, it is in `a[l,..,h]`. The algorithm considers the mid-point and compares its element with the target `e`. If the element is too large, the target must be in the lower half and reduces `h`. If the element is too small, the element must be in the upper half and increases `l`. It stops when it finds `e` or when the array is exhausted.

Correctness

Warning: If you write the exact same algorithm in Java, then it is not correct. In fact, most implementations of binary search were in error until 2006 (Jon Bentley).
`https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`

Efficiency

The sub-array is reduced in size by half at every step. In the worst case (when the element is not in the array), it will start at length `n` and go down to `1`. Therefore, the algorithm is $O(\log_2 n)$

$$n \to \left\lfloor \frac{n}{2} \right\rfloor \to \left\lfloor \frac{n}{4} \right\rfloor \to \left\lfloor \frac{n}{8} \right\rfloor \to \left\lfloor \frac{n}{16} \right\rfloor \to \dots$$

We need

$$n \approx 2^k \Leftrightarrow k \approx \log n$$

A note on notation

The notation $\log n$ (without a base) means:

- For a computer scientist : $\log_2 n$ (or $\lg n$)
- For a mathematician : $\log_e n$ (or $\ln n$)
- For an engineer : $\log_{10} n$

Recursive variation

Can you write a recursive version of `bsearch`?

Recursive variation

```python
def rbsearch(a,e,l,h):
    if l > h:
        return None
    else:
        m = (l+h)//2
        if a[m] == e:
            return m
        if a[m] < e:
            return rbsearch(a,e,m+1,h)
        else:
            return rbsearch(a,e,l,m-1)
```

Efficiency

- Let $T(n)$ be the number of calls to `rbsearch` with an array of length n

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(n/2) + C, & \text{otherwise} \end{cases}$$

- Therefore

$$\begin{aligned} T(n) &= T(n/2) + C \\ &= T(n/4) + 2C \\ &= T(n/8) + 3C \\ &= T(n/16) + 4C \\ &= \dots \\ &= 1 + (\log_2 n)C \end{aligned}$$

- Therefore, the algorithm is (still) $O(\log_2 n)$.

Moral(s) of this section

- Divide-and-conquer is a useful design technique.
- The runtime will often involve a log, which is very fast.
- Recursive implementation of divide-and-conquer are often natural.
- Recursive algorithms are just as easy to analyze as imperative ones.
- They do not affect the runtime (pace Guido).

Homework/Test questions

- Modify binary search to return the index of the first instance of the element.
- Modify binary search to return the indices of the first and last instance of the element.
- Given an unsorted array of numbers, return the k largest numbers.
- At what point does it get more efficient to sort the array?
- Can you extend the binary search of an vector to the search of a 2-d array?

Class project

Problem

You are given an array of length n that is filled with two symbols (say 0 and 1); all m copies of one symbol appear first, at the beginning of the array, followed by all $n - m$ copies of the other symbol. You are to find the index of the first copy of the second symbol in time $O(logm)$.