Extended Introduction to Computer Science CS1001.py

Lecture 5: Integer Exponentiation; Search: Sequential vs. Binary Search

Instructors: Amir Rubinstein, Amiram Yehudai

Teaching Assistants: Yael Baran, Michal Kleinbort,

School of Computer Science
Tel-Aviv University

Winter Semester, 2015-16

http://tau-cs1001-py.wikidot.com

Lecture 4: Highlights

- More on functions
- Tuples and lists.
- Multiple values returned by functions.
- Side effects of function execution.
- Natural numbers:
  - Unary vs. binary representation.
  - Representation in different base binary, decimal, octal, hexadecimal, 31, etc.).

This concludes the first part of the course:
Python basics and number representation.

The next part is

Basic Algorithms and their efficiency

We will present algorithms on numbers (Integer Exponentiation), and on general data (searching)

Lecture 5: Plan

- Integer exponentiation: Naive algorithm (inefficient).
- Integer exponentiation: Iterated squaring algorithm (efficient).
- Modular exponentiation.

- Searching in unordered lists and in ordered lists.
- Sequential search vs. binary search.

Integer Exponentiation

How do we compute a^b, where $a, b \geq 2$ are both integers?

```
>>> for i in range (2 ,20):
    print (17** i)
289
4913
83521
1419857
24137569
410338673
6975757441
118587876497
2015993900449
```

(continue)

```
34271896307633
582622237229761
9904578032905937
168377826559400929
2862423051509815793
48661191875666868481
827240261886336764177
14063084452067724991009
239072435685151324847153
```

Integer Exponentiation: Naive Method

How do we compute a^b, where $a,b \geq 2$ are both integers?

The naive method: Compute successive powers $a, a^2, a^3, \ldots, a^b$.

Starting with a, this takes b - 1 multiplications, which is exponential in the length of b, which is $\lfloor \log_2 b \rfloor + 1$

For example, if b is 20 bits long, say $b = 2^{20} - 17$, such procedure takes $b - 1 = 2^{20} - 18 = 1048558$ multiplications.

If b is 1000 bits long, say $b = 2^{1000} - 17$, such procedure takes $b = 2^{1000} - 18$ multiplications. In decimal, $2^{1000} - 18$ is

1071508607186267320948425049060001810561404811705533607443750
3883703510511249361224931983788156958581275946729175531468251
8714528569231404359845775746985748039345677748242309854210746
0506237114187795418215304647498358194126739876755916554394607
.7062914571196477686542167660429831652624386837205668069358

A 1000 bits long input is not very large.

Yet such computation is completely infeasible.

Integer Exponentiation: Naive Method, Python Code

```
def naive_power (a,b):
    """  computes a**b using all successive powers
         assumes b is a nonnegative integer   """
    result =1
    for i in range (0,b):          # b iterations
        result = result *a
    return result
```

Let us now run this on a few cases:

Integer Exponentiation: Naive Method, running the Python Code

```
>>> naive_power (3 ,0)
1
>>> naive_power (3 ,2)
9
>>> naive_power (3 ,10)
59049
>>> naive_power (3 ,100)
515377520732011331036461129765621272702107522001
>>> naive_power (3 , -10)
1
```

Take a look at the code and see if you understand it, and specifically why raising 3 to -10 returned 1.

More Efficient Integer Exponentiation: A Concrete Example

Suppose we want to compute a^{67}. We represent 67 as a sum of powers of 2 (this representation is unique, and corresponds to the binary representation of $67 = 1 + 2 + 64$, 1000011)

We first compute a^2; a^4; a^8; a^{16}; a^{32}; a^{64}. Each additional squaring takes just one multiplication (eg. $a^{64} = a^{32} \cdot a^{32}$). So overall, computing all these six exponents takes just 6 multiplications.

Next, we note that $a^{67} = a^{64+2+1} = a^{64} \cdot a^2 \cdot a^1$. So to compute a^{67}, once we have all the powers a^{2^i} takes 2 additional multiplications.

All in all, we need just 6+2=8 multiplications. Way better than the 67-1=66 multiplications of the naive method.

Efficient Integer Exponentiation: general observations (1)

Let $2^{l-1} \leq b < 2^{l}$ for some l, so the binary representation of b has exactly $l = \lfloor \log_2 b \rfloor + 1$ bits.

So Instead of computing all successive powers of a, namely $a, a^2, a^3, \ldots, a^b$

we can compute just successive powers of two powers of a, namely $a, a^2, a^4, a^8, \ldots, a^{2^{l-1}}$

To accomplish this, observe that $a^{2^{i+1}} = \left(a^{2^i}\right)^2$

So we start with $a^1 = a$, and iterate squaring of the last outcome to compute all the needed powers . Observe that squaring is just one multiplication.

How do we compute the desired power a^b?

Efficient Integer Exponentiation: general observations (2)

Having computed $\{a^1, a^2, a^4, a^8, \cdots, a^{2^{l-1}}\}$

We now want to combine them to the desired power, a^b , employing the relations $a^{c+d} = a^c \cdot a^d$

and $a^{c \cdot d} = (a^c)^d$

Let $b = \sum_{i=0}^{l-1} b_i \cdot 2^i$. The b_i ‘s are simply the bits in the binary representation of b. Then

$$a^b = a^{\sum_{i=0}^{l-1} b_i \cdot 2^i} = \prod_{i=0}^{l} (a^{2^i})^{b_i}$$

Thus we should accumulate only those powers that correspond to bits with value 1 in b

Integer Exponentiation: Iterated Squaring

We will not implement the algorithm as is, but we will develop an algorithm that is based on the same observations, but does not explicitly use the binary representation.

Integer Exponentiation: Iterated Squaring, Python Code

```
def power1(a,b):
    """ computes a**b using iterated squaring
        assumes b is a nonnegative integer   """
    result=1
    while b>0:
        if b % 2 == 1:      #  b is odd
            result = result*a
            b = b-1
        else:               #  b is even
            a=a*a
            b = b//2       #
    return result
```

since $a^b = \left(a^2\right)^{b/2}$

Integer Exponentiation:
Iterated Squaring, running the Python Code

Let us now run this on a few cases:

```
>>> power1(3,4)
81
>>> power1(5,5)
3125
>>> power1(2,10)
1024
>>> power1(2,30)
1073741824
>>> power1(2,100)
1267650600228229401496703205376
>>> power1(2,-100)
1
```

Integer Exponentiation: Iterated Squaring, correctness of the Python Code

We can prove the correctness of the function, by showing a loop invariant – a condition that holds each time we are about to check the loop condition.

Denote the arguments to the function by A, B (to distinguish from the changing values a, b).

We claim that each time we are about to check the loop condition, the following condition holds:

$$result \cdot a^b = A^B$$

First, lets check it by adding printing to the code:

```
…
while b>0:
        print( "result = “,result," a = “, a," b = “ ,b,
                " result*(a**b)= “, result*a**b)
        if b % 2 == 1:
        .....
```

Integer Exponentiation: Iterated Squaring, correctness of the Python Code (cont.)

When we run

```
>>> power1(3,11)
result =  1  a =  3  b =  11  result*(a**b)=  177147
result =  3  a =  3  b =  10  result*(a**b)=  177147
result =  3  a =  9  b =  5  result*(a**b)=  177147
result =  27  a =  9  b =  4  result*(a**b)=  177147
result =  27  a =  81  b =  2  result*(a**b)=  177147
result =  27  a =  6561  b =  1  result*(a**b)=  177147
177147
```

So at least in this example the condition holds every time!

Integer Exponentiation: Iterated Squaring, correctness of the Python Code (cont.)

Now we want to prove that this is indeed an invariant condition.

We need to show that it holds the first time we enter the loop.

Then show that if it holds before we check the loop condition, and we then execute the loop body once, the condition will hold again the next time.

And then we will have to show that if the condition holds the last time, (just before we exit the loop), then the function will return the correct value.

The first time we enter, result=1, a=A, and b=B, so the condition is true.

$$result \cdot a^b = A^B$$

Integer Exponentiation: Iterated Squaring, correctness of the Python Code (cont.)

Now execute the loop body once. The values of the variables change (‘ denote the value after). There are two possibilities:

If b is odd, then

The code:

```
if b % 2 == 1:
    result = result*a
    b = b-1
```

$$result' = result \cdot a$$

$$b' = b - 1$$

$$a' = a$$ unchanged

so $$result' \cdot a^{b'} = result \cdot a \cdot a^{b-1} = result \cdot a^{b} = A^{B}$$

Substitute the values

Was known before

So the condition remains true after executing the loop body.

Integer Exponentiation: Iterated Squaring, correctness of the Python Code (cont.)

The second possibility when we execute the loop body once:

If b is even, then

unchanged

$$result' = result$$

$$b' = b/2$$

$$a' = a^2$$

The code:

```
else
    a=a*a
    b = b//2
```

so $$result' \cdot a'^{b'} = result \cdot a^{2\cdot(b/2)} = result \cdot a^{b} = A^{B}$$

Substitute the values

since b is even $2\cdot(b/2) = b$

Was known before

So the condition remains true after executing the loop body.

Integer Exponentiation: Iterated Squaring, correctness of the Python Code (cont.)

So in both cases, the invariant condition remains true after each execution of the loop body.

Then, when the loop terminates, b=0 (why?)

so $$A^B = result \cdot a^b = result \cdot a^0 = result$$

The invariant holds

Substitute the value

So we showed that the function returns the desired value A^B

We can also see that the loop terminates, because b is reduced in each execution of the loop body

QED

Correctness of Code

In general, it is not easy to design correct code. It is even harder to prove that a given piece of code is correct (namely it meets its specifications).

In the course, we will see a couple more examples of program correctness, using the same technique of loop invariants. (You will not be expected to prove correctness in this way.)

However, in most cases you will have to rely on your understanding, intuition, test cases, and informative prints to convince yourselves that the code you wrote is ~~indeed~~ hopefully correct.

Finally, we remark that software and hardware verification are major issues in the corresponding industries. Elective courses on these topics are being offered at TAU (and elsewhere).

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
  while b  > 0 :
      if b % 2 == 1:
        result = result*a
        b = b-1
      else:
        a=a*a
        b = b//2
  return result
```

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
    while b  > 0 :
        if b % 2 == 1:
            result = result*a
            b = b-1

        a=a*a
        b = b//2
    return result
```

If b is odd, it becomes even. so no need to test again if b is odd or even.

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
    while b  > 0 :
        if b % 2 == 1:
            result = result*a


        a=a*a
        b = b//2
    return result
```

No need to decrement b, b //2 will give correct result for odd b.

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
    while b  > 0 :
        if b % 2 == 1:
            result = result*a



        a=a*a
        b = b//2
    return result
```

A numeric expression is viewed as True if not equal 0.

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
    while b        :
        if b % 2 == 1:
            result = result*a



        a=a*a
        b = b//2
    return result
```

A numeric expression is viewed as True if not equal 0.

Good style?(I don't like it)

Iterated Squaring, improving the Code

```
def power1 (a,b):
result=1
    while b  > 0  :
        if b % 2 == 1:
            result = result*a



        a=a*a
        b = b//2
    return result
```

A numeric expression is viewed as True if not equal 0.
Good style?(I don't like it)

Iterated Squaring, improving the Code

```
def power   (a,b):
result=1
    while b  > 0  :
        if b % 2 == 1:
            result = result*a



        a=a*a
        b = b//2
    return result
```

A numeric expression is viewed as True if not equal 0.

Good style?(I don't like it)

Iterated Squaring, improving the Code: explanation

Note that when b is odd, the next time it will be even. So we don’t have to test again if b is odd or even. This makes the code more compact (and saves some of the tests, but the number of multiplications is unchanged).

We also don’t have to decrement b, because b //2 will give the correct result also for the odd b. (eg. 7//2 == 3)

Another change is to use the fact that python allows us to put a numeric expression where a boolean expression is expected (for example as while loop expression). Any numeric result not equal to 0 is treated as True. Is it good style? certainly common among python (and other languages) programmers.

Integer Exponentiation: Iterated Squaring, improved Python Code

```
def power(a,b):
   """ computes a**b using iterated squaring
      assumes b is a nonnegative integer   """
   result=1
   while b > 0 :             # b is nonzero
      if b % 2 == 1:   #  b is odd
         result = result*a
      a=a*a
      b = b//2
   return result
```

Note that the computation of a and b the last time the loop body is executed are not needed (and do not affect the result)

The Two Types of Time Complexity Analysis

1) Mathematical analysis:

- Analyzing the number of operations exactly.
- Analyzing the number of operations approximately (up to
- constants) and asymptotically (we will do this a lot in the future).
- Caveat: When faced with a concrete task on a specific problem size, you may be far away from “the asymptotic”.

2) Direct measurements of the actual running time:

- For direct measurements, we will use either the time package and the time.clock() function.
- Or the timeit package and the timeit.timeit() function.
- Both have some deficiencies, yet are highly useful for our needs.

Running Time Analysis:
Naive vs. Iterated Squaring

We saw that the naïve algorithm makes an exponential number of multiplications.

The analysis of the number of multiplications performed by the iterated squaring algorithms is left for the homework.

We will now measure the actual running time directly.

Direct Time Measurement, Using time.clock()

The function elapsed measures the CPU time taken to execute the given expression (given as a string). Returns a result in seconds. Note that the code first imports the time module.

```
import time        # imports the Python time module

def elapsed (expression, number =1):
''' computes elapsed time for executing code number times
(default is 1 time). expression should be a string representing a
Python expression.'''
t1= time.clock ()
for i in range (number):
    eval (expression)        # eval invokes the interpreter
t2= time . clock ()
return t2 -t1
```

Direct Time Measurement, Using time.clock()

From the edit window with the file containing elapsed, we hit the F5 button or choose run => run module

Examples:

```
>>> elapsed (" sum ( range (10**7)) ")
0.33300399999999897
>>> elapsed (" sum ( range (10**8)) ")
3.362785999999998
>>> elapsed (" sum ( range (10**9)) ")
34.029920000000004
```

Reality Show: Naive Squaring vs. Iterated Squaring

Actual Running Time Analysis:

We'll measure the time needed (in seconds) for computing 3 raised to the powers $2 \cdot 10^5$, 10^6 , $2 \cdot 10^6$ using the two algorithms.

```
>>> from power import *                    # Note!!
>>> elapsed (" naive_power (3 ,200000) ")
2.244201
>>> elapsed (" power (3 ,200000) ")
0.03179299999999996
>>> elapsed (" naive_power (3 ,1000000) ")
57.69631299999996
>>> elapsed (" power (3 ,1000000) ")
0.3366879999999952
>>> elapsed (" naive_power (3 ,2000000) ")
205.5677550000003
>>> elapsed (" power (3 ,2000000) ")
1.0069569999999999
```

Iterated squaring wins (big time)!

Comment about time.clock()

The python documentation for time.clock() states that it is

> Deprecated since version 3.3: The behaviour of this function depends on the platform: use perf_counter() or process_time() instead, depending on your requirements, to have a well defined behaviour.

Deprecated means: advice not to use it in new code written, but it is not removed from the language, so that old code does not stop functioning.

The reason – it measures different “time” on different systems: processor time vs. wall-clock time.

Wait a Minute

Using iterated squaring, we can compute a^b for any a and, say, $b = 2^{100} - 17$ (= 1267650600228229401496703205359). This will take less than 200 multiplications, a piece of cake even for an old, faltering machine.

A piece of cake? Really? 200 multiplications of what size numbers?

For any integer a other then 0 or 1, the result of the exponentiation above is over 2^{99} bits long. No machine could generate, manipulate, or store such huge numbers.

Can anything be done? Not really!

Unless you are ready to consider a closely related problem:

Modular exponentiation: Compute $a^b \bmod c$, where $a, b, c \geq 2$ are all integers. This is the remainder of a^b when divided by c. In Python, this can be expressed as `(a**b) % c`.

Modular Exponentiation

We should still be a bit careful. Computing a^b first, and then taking the remainder mod c, is not going to help at all.

Instead, we compute all the successive squares mod c, namely a^1 mod c, a^2 mod c, a^4 mod c (and any other power that is needed).

In fact, following every multiplication, we compute the remainder. We rely on the fact that for all a, b, c :

$((a \bmod c) \cdot (b \bmod c)) \bmod c = (a \cdot b) \bmod c.$

This way, intermediate results never exceed c^2, eliminating the problem of huge numbers.

Modular Exponentiation in Python

We can easily modify our function, power, to handle modular exponentiation.

```
def modpower(a,b,c):
    """ computes a**b modulo c, using iterated squaring
        assumes b is a nonnegative integer   """
    result=1
    while b>0:                # while b is nonzero
        if b % 2 == 1:     # b is odd
            result = (result * a) % c
        a= (a*a) % c
        b = b//2
    return result
```

Modular Exponentiation in Python

A few test cases:

```
>>> modpower(2,10,100) # sanity check: 2^10 = 1024
24
>>> modpower(17,2**100+3**50,5**100+2)
3568728177468732193582328510109849308957750682733818418319936978305748
>>> 5**100+2 # the modulus, in case you are curious
7888609052210118054117285652827862296732064351090230047702789306640627
>>> modpower(17,2**1000+3**500,5**100+2)
1119887451125159802119138842145903567973956282356934957211106448264630
```

Built In Modular Exponentiation: pow(a,b,c)

Guido van Rossum has not waited for our code, and Python has a built in function, pow(a,b,c), for efficiently computing a^b mod c.

```
>>> modpower (17 ,2**1000+3**500 ,5**100+2)\ # line continuation
        - pow (17 ,2**1000+3**500 ,5**100+2)
0
# Comforting : modpower code and Python pow agree . Phew ...
>>> elapsed (" modpower (17 ,2**1000+3**500 ,5**100+2) ")
0.002635999999999542
>>> elapsed (" modpower (17 ,2**1000+3**500 ,5**100+2) ",number
=1000)
2.280894000000046
>>> elapsed (" pow (17 ,2**1000+3**500 ,5**100+2) ",number
=1000)
0.7453199999999924
```

So our code is just three times slower than pow.

Does Modular Exponentiation Have Any Uses?

Applications using modular exponentiation directly (partial list):

- Randomized primality testing.
- Diffie Hellman Key Exchange
- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC)

We will discuss the first two topics later in this course, and leave RSA PKC to an (elective) crypto course.

Search

(taken from http://bizlinksinternational.com/web/web%20seo.php)

Search

Search has always been a central computational task. In early days, search supposedly took one quarter of all computing time. The emergence and the popularization of the world wide web has literally created a universe of data, and with it the need to pinpoint information in this universe.

Various search engines have emerged, to cope with this challenge. They constantly collect data on the web, organize it, index it, and store it in sophisticated data structures that support efficient (fast) access, resilience to failures, frequent updates, including deletions, etc., etc.

In this class we will deal with two much simpler data structures that support search:

- unordered list
- ordered list

Representing Items in a List

We are about to study search. Assume our data is arranged in a list. Recall that in Python, a list with n elements is simply a mapping from the set of indices, {0, … , n-1}, to a set of items.

In our context, we assume that items are records having a fixed number of information fields. For example, our Student items will include two fields each: name, identity number.

We will arrange each item as a list with two entries (corresponding to the example above).

We note that this is cumbersome and will not scale up easily. How would one remember that entry 19 corresponds to weight, and entry 17 corresponds to height?

Indeed, we will later introduce classes and object oriented programming for a slicker representation of such records.

Representing Students' Records in a List

The following list was generated manually from the 109 strong list of students in a previous year class. To protect their privacy, only first names are given (hopefully spelled correctly), and their id numbers were generated at random.

(Bear this in mind when you apply to get your new biometric ID card.)

Representing Students' Record in a List (cont)

```
import random
names =["Or","Yana","Amir","Roee","Noa","Gal","Barak",
           "Rina","Tal","Lielle","Shady","Yuval"]
students_list =[[ name , random.randint (2*10**7 ,6*10**7)] \
        for name in names ]    # leading digits are 2 ,3 ,4 ,5
>>> print (students_list )
[[ 'Or', 28534293] , ['Yana', 45929500] , ['Amir', 37076235] ,
['Roee', 55421212] , ['Noa', 46931670] , ['Gal', 55522009] ,
['Barak', 22162470] , ['Rina', 25310060] , ['Tal', 23374569] ,
['Lielle', 26549109] , ['Shady', 34859880] , ['Yuval', 28714343]]
>>> len ( students_list )
12
```

Searching the List

We are now interested in searching the list. For example, we want to know if a student called Yuval is in the list, and if so, what is his/her ID number. In this example, the student's name we look for is viewed as the key, and the associated ID is the value we are interested in.

With such an unordered list, we have no choice but to search for items sequentially, one by one, in some order. For example, by going over the list from the first entry, students_list[0], to the last entry, students_list[11].

What is the best case running time of sequential search? Worst case running time?

Food for thought: Would it be better to sample items at random? (think of best, worst, and average cases).

Sequential Searching: Code

```
def sequential_search (key , lst ):
    """ sequential search from lst [0] till last lst element
    lst need not be sorted for sequential search to work """
    for elem in lst :
        if elem [0]== key :
            return elem
    # we get here when the key is not in the list
    print (key , "not found" )      # For debugging purposes!
    return None
```

Searching backwards : Code

```
def sequential_search_back (key , lst ):
    """ sequential search from last lst element to first element
    lst need not be sorted for sequential search to work """
    for elem in lst [:: -1]:   # goes over elements in reversed lst
        if elem [0]== key :
            return elem
    # we get here when the key is not in the list
    print (key , "not found" )
    return None
```

Is this list reversing a good idea? Think what will happen to the worst and best case inputs. If not, how would you fix this?

Searching the Short List

```
>>> sequential_search ("Or", students_list )
['Or ', 28534293]
>>> sequential_search ("Benny", students_list )
Benny not found
>>> sequential_search_back ("Shady", students_list )
['Shady ', 34859880]
>>> sequential_search ("Shady", students_list )
['Shady ', 34859880]
>>> sequential_search_back (8, students_list )
8 not found
```

Question: What keys cause worst case running time for both forward and backward sequential searches?

Sequential Search: Time Analysis

Any sequential search in an unordered list goes over it, item by item. If the list is of length n, sequential search will take n steps in the worst case (when the item is not found because it is missing).

For our exclusive (thus short) list of students, n steps is not a problem. But if n is very large, such a search will take very long.

Search in Unordered vs. Ordered Lists

Hands on experience: Searching for a word in a book vs. searching for it in a dictionary.

(We mean a real world, hard copy, dictionary, not Python's dict, which we soon will get familiar with!)

Sequential vs. Binary Search

For unordered lists of length n, in the worst case, a search operation compares the key to all list items, namely n comparisons.

On the other hand, if the n element list is sorted, search can be performed much faster. We first compare input key to the key of the list's middle element, an element whose index is $\lfloor (n-1)/2 \rfloor$

- If the input key equals the middle element's key, we return the middle element and terminate.
- If the input key is greater than the middle element's key, we can restrict our search to the top half of the list (indices from $\lfloor (n-1)/2 \rfloor + 1$ to *n-1*
- If the input key is smaller than the middle element's key, we can restrict our search to the bottom half of the list (indices from 0 to $\lfloor (n-1)/2 \rfloor - 1$

Binary Search, cont.

Starting with an ordered list with n elements, we will initialize three indices:

left=0, right=n-1, middle=(n-1)//2.

We compare the values at the middle index to the key.

If equal – we found an item equal to the key, and return its index.

If key is smaller than middle element – we assign the value of right to the variable middle. Update middle, and iterate.

If key is larger than middle element – we assign the value of left to the variable middle. Update middle, and iterate.

We announce that the key was not found if right becomes smaller than left.

Binary Search: Python Code

```
def binary_search (key , lst ):
    """ iterative binary search. lst must be sorted """
    n= len ( lst )
    left =0
    right =n -1
    outcome = None              # default value
    while left <= right :
        middle =( right + left )//2
        if key == lst [ middle ][0]:            # item found
            outcome =lst [ middle ]
            break                # gets out of the loop if key was found
        elif key < lst [ middle ][0]:           # item cannot be in top half
            right =middle -1
        else :                          # item cannot be in bottom half
            left = middle +1
    if not outcome :                # holds when the key is not in the list
        print (key , "not found")
    return outcome
```

Animated Example

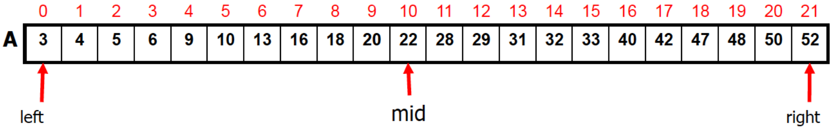
For simplicity, the entries in our list will be plain integers (not lists).

Animated Example: Searching for the Existing Item, 18[§]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

[§] artwork by an AR (anonymous researcher)

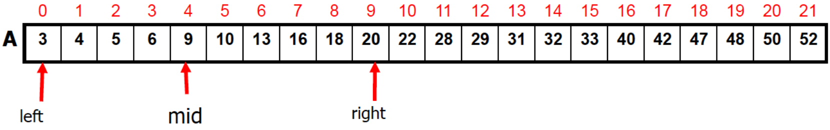
Animated Example: Searching for the Existing Item, 18[¶]



`A[mid] = 22 > 18`

[¶] artwork by an AR (anonymous researcher)

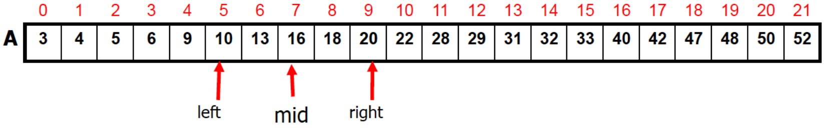
Animated Example: Searching for the Existing Item, 18‖



`A[mid] = 9 < 18`

‖artwork by an AR (anonymous researcher)

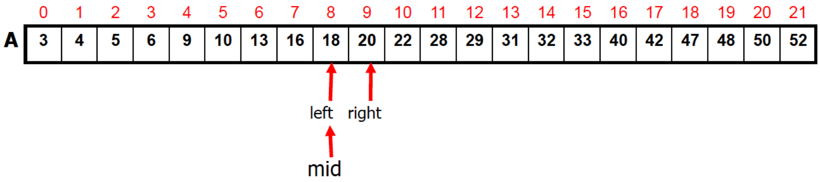
Animated Example: Searching for the Existing Item, 18**



`A[mid] = 16 < 18`

**artwork by an AR (anonymous researcher)

Animated Example: Searching for the Existing Item, 18[††]



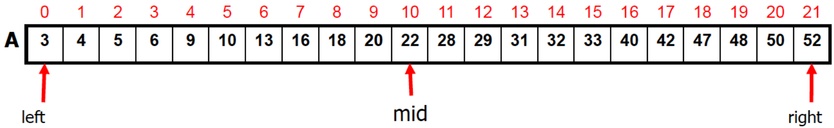
`A[mid] = 18 = 18`. Item found.

[††] artwork by an AR (anonymous researcher)

Animated Example: Searching for a Non Existing Item, 17

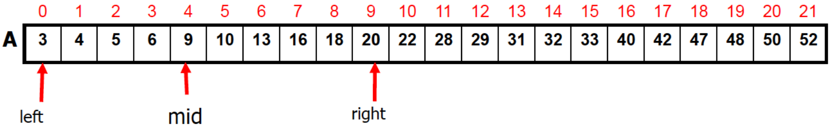
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

Animated Example: Searching for a Non Existing Item, 17



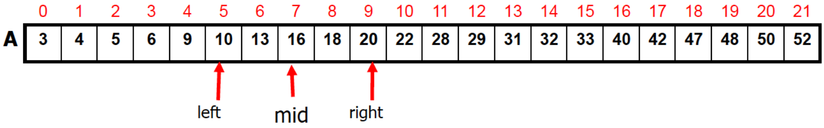
`A[mid] = 22 > 17`

Animated Example: Searching for a Non Existing Item, 17



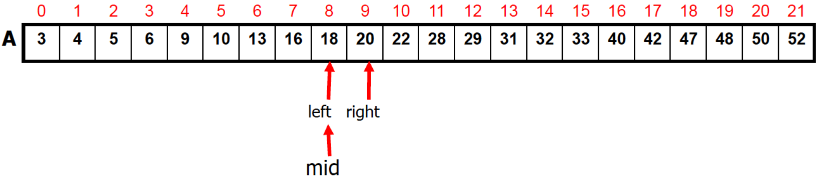
`A[mid] = 9 < 17`

Animated Example: Searching for a Non Existing Item, 17



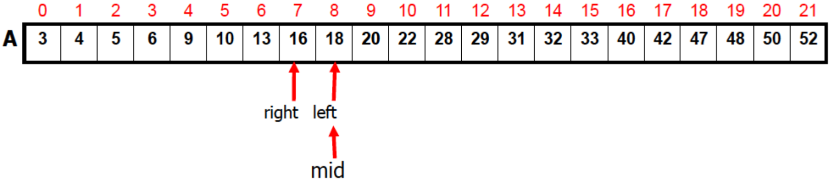
`A[mid] = 16 < 17`

Animated Example: Searching for a Non Existing Item, 17



`A[mid] = 18 > 17.`

Animated Example: Searching for a Non Existing Item, 17



`A[mid] = 18 > 17`. Item not found.

Time Analysis of Binary Search

At each stage, we either terminate or cut the size of the remaining list by half. Hence the name binary search.

- We start with an ordered list of length n.
- If n = 1, we compare the only item in the list to the key, and terminate.
- If n > 1, we either terminate in one step (if the key is found), or continue to look for the key in a list of length $\lfloor n/2 \rfloor$
- In each iteration we perform one comparison, and cut the length by one half, till the length reaches n = 1.
- The number of times we can halve n till we reach 1 is $\lceil \log_2(n) \rceil \leq \log_2(n) + 1$
- So the running time of binary search is proportional to $\log_2 n$. For large n, it is much faster than the n steps of sequential search.
- Binary search requires preprocessing: The list must be sorted.

Binary Search: Preprocessing

As a sanity check, we first run the code on the small students list of length 12, used above to test the sequential search code.

>>> print (students_list)

[['Or', 28534293] , ['Yana', 45929500] , ['Amir', 37076235] , ['Roee', 55421212] , ['Noa', 46931670] , ['Gal', 55522009] , ['Barak', 22162470] , ['Rina', 25310060] , ['Tal', 23374569] , ['Lielle', 26549109] , ['Shady', 34859880] , ['Yuval', 28714343]]

To apply binary search, we better sort the list first. We want to sort the items by their names. We employ a built-in sorting function, sorted, and tell it to use the name as the key, employing a lambda expression: key = lambda elem : elem[0].

Binary Search: Preprocessing, cont.

```
>>> sorted_list = sorted (students_list,
                                   key = lambda elem : elem [0])
# sorting students_list by the names
>>> sorted_list
[[ 'Amir', 37076235] , ['Barak', 22162470] , ['Gal', 55522009] ,
['Lielle', 26549109] , ['Noa', 46931670] , ['Or', 28534293] ,
['Rina', 25310060] , ['Roee', 55421212] , ['Shady', 34859880] ,
['Tal', 23374569] , ['Yana', 45929500] , ['Yuval', 28714343]]
```

(Lambda expressions will be discussed in the course soon, and we will then explain how this works)

Binary Search: Running the Code

```
>>> binary_search ("Or", sorted_list)
['Or ', 28534293]
>>> binary_search ("Shady", sorted_list )
['Shady ', 34859880]
>>> binary_search ("Benny", sorted_list)
Benny not found
```

What happens if we run binary search on an unsorted list?

```
>>> binary_search ("Or", students_list )
Or not found
```

This should not come as a surprise.

The running time of sequential and binary search for short lists (e.g. of length 12, namely $1 + \log_2(12) = 4$ vs. 12) are not easy to tell apart.

The Difference is distinguishable when considering longer lists, e.g. of length $n = 10^6 = 1,000,000$.

Sequential and Binary Search: Timing the Code on Long Lists

We could have based our list on the leaked Israeli population registry. However, to avoid potential legal troubles, the names and identity numbers were generated completely at random (details later).

```
>>> large_stud_list = students (10**6)
>>> large_stud_list [5*10**5+3]
['Rlne Qmgedu', 39925262]
```

For the binary search, we sort the list

```
>>> large_sorted_list = sorted ( large_stud_list ,
                                   key = lambda elem : elem [0])
>>> large_sorted_list [5*10**5+3]
['Naq Tbsc', 58042807]
```

Sequential Search: Timing the Code on Long Lists

```
>>> sequential_search ('Rlne Qmgedu', large_stud_list )
['Rlne Qmgedu ', 39925262]
>>> elapsed ("sequential_search('Rlne Qmgedu',
                        large_stud_list )", number =1000)
44.626914
>>> sequential_search ('Rajiv Gandhi', large_stud_list )
>>> elapsed (" sequential_search ('Rajiv Gandhi',
            large_stud_list )", number =1000)
91.31152699999998
```

So, one thousand sequential searches in a 1, 000, 000 long list for an existing key (located around the middle of the list) took about 45 seconds, while a non-existing key took about 91 seconds.

Note: print(key, "not found") was disabled. (Why?).

Binary Search:
Timing the Code on Long Lists

>>> binary_search ("Naq Tbsc", large_sorted_list)
['Naq Tbsc', 58042807]
>>> binary_search ("Rajiv Gandhi", large_sorted_list)
Rajiv Gandhi not found
>>> elapsed (" binary_search ('Naq Tbsc', large_stud_list)", number =1000)
0.031651999999997981
>>> elapsed (" binary_search ('Rajiv Gandhi', large_stud_list)", number =1000)
0.030768999999992275

So, one thousand binary searches in a 1, 000, 000 long list for both an existing and a non-existing item took about 0.03 seconds. This is 1,410 times faster than sequential search.

Binary Search: A High Level View

Binary search is widely applicable (not only for searching a list). In general, when we look for an item in a huge space, and that space is structured so we could tell if the item is

1. right at the middle,
2. in the top half of the space,
3. or in the lower half of the space.

In case (1), we solve the search problem in the current step. In cases (2) and (3), we deal with a search problem in a space of half the size.

In general, this process will thus converge in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance of binary search to that of going sequentially over the original space, item by item.

Sometimes this algorithmic idea is call "A lion in the desert".