Computing $a^b \bmod k$

```
1  pow1(a,b,k)
2      s=1
3      for i=1 to b
4          s=(s·a) mod k
5      return s
```

Number of operations is $O(b)$
Size of input is $\lfloor \log b \rfloor + 1$
$2^{\log b} = b$
Time complexity is $O(2^n)$

Computing $a^b \bmod k$

```
1  pow2(a,b,k)
2      d=a
3      e=b
4      s=1
5      while e>0
6          if e is odd
7              s=(s·d) mod k
8          d=d^2 mod k
9          e=⌊e/2⌋
10     return s
```

Number of operations is proportional to the number of times e=b can be halved before e=0 – ie at most $\lceil \log_2 b \rceil$
Number of operations is $O(\log b)$
Size of input is $\lfloor \log b \rfloor + 1$
Time complexity is $O(n)$

Notes:
Optimization – if d becomes 1, it will remain 1;
$1^2 \bmod k = 1$
hence s will not change;
$s < k$ and $s \cdot 1 = s$

Understanding – consider b in binary representation

$$b = \sum_{h=0}^{n-1} b_h \cdot 2^h$$

$$a^b = \prod_{h=0}^{n-1} a^{(b_h \cdot 2^h)} \bmod k$$

e holds a^{2^h} on entry to the h^{th} iteration of the loop (from h=0 to h=n-1)

Reminder $(a \cdot b) \bmod k = (a \bmod k \cdot b \bmod k) \bmod k$
$a^b \bmod k = (a \bmod k)^b \bmod k$

Determine whether array A of length n contains t

```
1  LinearSearch(A,n,t)
2      for i=0 to n-1
3          if A[i]=t
4              return 'Found'
5      return 'Not Found'
```

Worst case: t is not in A, $3n+1$ operations
size of input is proportional to n
Time complexity is $O(n)$

Determine whether sorted array A of length n contains t

```
1  BinarySearch(A,n,t)
2      beginp=0
3      endp=n
4      ptr=⌊n/2⌋
5      while (endp>beginp)
6          if A[ptr]=t
7              return 'Found'
8          if A[ptr]<t
9              beginp=ptr+1
10         if A[ptr]>t
11             endp=ptr
12         ptr=⌊(beginp+endp)/2⌋
13     return 'Not Found'
```

Worst case: t is not in A, $6 \log n + 3$ operations
Number of iterations of the loop is the logarithm (base 2) of the length of the list
- the length of the list considered is roughly halved on each iteration
- number of times a list can be halved $\log_2$ length of the list

Time complexity is $O(\log n)$

Given an array W of L words and an array D of n words, report which elements of W are not in D

```
1  SpellCheck(W,L,D,n)
2      for i=0 to L-1
3          if BinarySearch(D,n,W[i])!='Found'
4              print W[i] + 'Misspelled?'
```

For an input of L words and a dictionary of size n, SpellCheck runs in worst case $O(L \log n)$ time

Task — Find the highest common factor of a and b

Algorithm

```
1 //Assume a≥b
2 hcf(a,b)
3     if b=0
4         return a
5     r= a mod b
6     return hcf(b,r)
```

Correctness

Let $r = a \bmod b$

$hcf(a,b) = hcf(b,r)$, because

1) all factors of a and b are also factors of b and r
2) all factors of b and r are also factors of a and b

As a and b and b and r have the same factors, they must have the same highest common factor

As $r = a \bmod b$, $\exists q$ such that $a = bq + r$, so $r = a - bq$

Proof of 1: suppose x is a factor of a and b, then $\exists y$ and z such that $a = xy$, $b = xz$

hence $r = xy - xzq$

$r = x(y - zq)$

-x is a factor of r (and also b and r)

Proof of 2: suppose L is a factor of b and r, then $\exists m$ and n such that $b = Lm$ and $r = Ln$

$a = Lmq + Ln$

$a = L(mq + n)$

-L is a factor of a (and also a and b)

Termination

Let $b_1, b_2, \dots, b_n$ be the second arguments of successive calls to hcf

From the i^{th} call to hcf with arguments a_i and b_i, the $(i+1)^{th}$ call to hcf is made with arguments

$a_{i+1} = b_i$

$b_{i+1} = a_i \bmod b_i$

clearly $b_{i+1} < b_i$ and as $a_i \geq b_i$, $b_{i+1} < a_i$

The $(i+2)^{th}$ call to hcf has arguments

$a_{i+2} = b_{i+1} = a_i \bmod b_i < b_i \leq a_i$

$b_{i+2} = a_{i+1} \bmod b_{i+1} < b_{i+1} < b_i$

The second argument decreases with each call and cannot go below 0 (as it's the result of a modulo operation. It must therefore for some call equal 0 at which point no successive calls are made

Run time

Claim $- a_{i+2} \leq a_i/2$

Proof:

case 1 - $a_{i+1} \leq a_i/2$

$a_{i+2} \leq a_{i+1}$ so $a_{i+2} \leq a_i/2$

case 2 - $a_{i+1} > a_i/2$

$a_{i+2} = a_i \bmod a_{i+1}$

as $a_{i+1} > a_i/2$, $a_i \bmod a_{i+1} < a_i/2$

Conclusion, a is halved at least every other call to hcf.

a can only be halved $\lfloor \log_2 a \rfloor + 1$ times.

Number of calls is at most $\max(2, 2\log_2 a)$

Each call requires a constant number of operations, $O(1)$.

Running time is $O(\log a)$

ie linear in the size of input $(\log a + \log b)$

Task	Algorithm	Analysis
Sorting a list, L	1 QuickSort(L) 2 if $\|L\| \leq 1$ 3 return L 4 remove first element, pivot, from L 5 $L_{\leq}$ = elements of L $\leq$ pivot 6 $L_{>}$ = elements of L $>$ pivot 7 L_l = QuickSort($L_{\leq}$) 8 L_r = QuickSort($L_{>}$) 9 return L_l + [pivot] + L_r	Worst case: For each recursive call one of $L_{\leq}$ or $L_{>}$ is an empty list. $\|L\|$ recursive calls are made, the argument is one element shorter each call. (calculating $L_{\leq}$ and $L_{>}$ requires $O(\|L\|)$ steps Let $n = \|L\|$, total work is order $n+n-1+\cdots+1 = \frac{n(n+1)}{2}$ $O(\frac{1}{2}n(n+1)) = O(n^2)$ Worst case time complexity: $O(n^2)$
Merging 2 sorted lists, L_1 and L_2	1 Merge(L_1, L_2) 2 if L_1 = [] return L_2 3 if L_2 = [] return L_1 4 $x_1 = L_1[0]$ 5 $x_2 = L_2[0]$ 6 $L_1' = L_1[1:\|L_1\|-1]$ 7 $L_2' = L_2[1:\|L_2\|-1]$ 8 if $x_1 \leq x_2$ 9 return $[x_1]$+Merge(L_1', L_2) 10 return $[x_2]$+Merge(L_1, L_2')	Worst case (same as best and average): When Merge(L_1, L_2) is called, 1 recursive call is made in which $\|L_1\|+\|L_2\|$ decreases by 1 A constant number of operations is executed for each recursive call. Let $n = \|L_1\|+\|L_2\|$ Worst/average/best case time complexity: $O(n)$
Sorting a list, L	1 MergeSort(L) 2 If $\|L\| \leq 1$ 3 return L 4 Split L into roughly equal halves L_l and L_r 5 return Merge(MergeSort(L_l), MergeSort(L_r))	Worst case (same as best and average): Each call gives rise to 2 other calls to Mergesort at one greater depth of recursion - each depth of recursion there are twice as many recursive calls **Maximum depth of recursion** is the maximum number of times the list can be halved = $\lceil \log_2 n \rceil$ The number of calls is $2^{\lceil \log_2 n \rceil} \leq 2n$ (a better bound is $n-1$) Time taken to merge is $O(n)$ suggesting complexity bound $O(n^2)$ but this is not the case. - Total lengths of lists processed at each level of recursion is $\|L\| = n$ so total amount of work done at each level of recursion is linear in the size of the ~~input~~ Number of times L can be halved: $O(\log n)$ So maximum depth of recursion: $O(\log n)$ Worst/average/best case time complexity: $O(n \log n)$