algorithms and imperative programming (i)

Available at JTANG.DEV/RESOURCES

complexity measures

$\mathbf{O}(f)$ denotes a set of functions:
$\{g : \mathbb{N} \to \mathbb{N} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+, \forall n > n_0, g(n) \leq c \cdot f(n)\}$
$\boldsymbol{\Omega(f)}$ denotes a set of functions:
$\{g : \mathbb{N} \to \mathbb{N} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+, \forall n > n_0, g(n) \geq c \cdot f(n)\}$
$\boldsymbol{\Theta(f)}$ denotes $\boldsymbol{O(f) \cap \Omega(f)}$

euclid's algorithm

```
Algorithm EuclidGCD(a, b):
  Input: Non-negative integers a and b
  Output: gcd(a, b)

  if b = 0 then
    return a
  return EuclidGCD(b, a mod b)
end
```

correctness
Let $d = gcd(a, b)$ and $c = gcd(b, a - rb)$.
We need to show that $gcd(a, b) = gcd(b, a - rb)$, so $d = c$.
By definition of d, we have the number $\frac{(a-rb)}{d} = \frac{a}{d} - r\frac{b}{d}$ is an integer as $d|a$ and $d|b$ and we have also shown $d|a - rb$ hence $d \leq c$.
Now by definition of c, $\frac{a-rb}{c} = \frac{a}{c} - r\frac{b}{c}$ shows that $c|a$ as we know $r\frac{b}{c}$ is an integer and $\frac{a-rb}{c}$ is an integer, so we have $c \leq d$.

complexity
After the first call, the first argument is always larger than the second one. Denote a_i as the first argument of the ith recursive call of EuclidGCD. It is clear that the second argument of a recursive call is equal to a_{i+1} and we also have

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

which implies the sequence a_i is strictly decreasing. We claim that

$$a_{i+2} < \frac{1}{2} a_i$$

case 1: $a_{i+1} \leq \frac{1}{2} a_i$, since the sequence of a_i's is strictly decreasing, we have

$$a_{i+2} < a_{i+1} \leq \frac{1}{2} a_i$$

case 2: $a_{i+1} > \frac{1}{2} a_i$, in this case $a_{i+2} = a_i \bmod a_{i+1}$, so we have

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

Thus the size of the first argument to the EuclidGCD method decreases by half with every other recursive call. Hence we have $\boldsymbol{O(log \max(a, b))}$.

modular arithmetic

```
Algorithm pow1(a, b, k):
  Input: Integers a, b, k
  Output: a^b mod k

  s = 1
  for i from 1 to b
    s = s * a mod k
  return s
end
```

The number of operations performed here is clearly $O(b)$, therefore the time complexity is $O(2^n)$ as the size of b is $\log_2 b$.

```
Algorithm pow2(a, b, k):
  Input: Integers a, b, k
  Output: a^b mod k

  d = a, e = b, s = 1
  until e = 0
    if e is odd
      s = s * d mod k
    d = d * d mod k
    e = floor(e / 2)
  return s
end
```

The number of operations performed here is proportional to the number of times e $(= b)$ can be halved before reaching 0, i.e. at most $\lceil log_2 b \rceil$. It follows that this algorithm has running time in $O(n)$.

primitive roots
We say that g is a **primitive root** with respect to p means that $\mathbb{Z}_p = \{1, 2, \cdots, p-1\} = \langle g \rangle = \{g^i \bmod p \,|\, i \in \mathbb{Z}\}$

```
Algorithm dl(y, g, p):
  Input: Integers y, g, p
  Output: x such that y = g^x mod p

  a = y mod p
  for x from 1 to p - 1
    b = pow2(g, x, p)
    if a = b
      return x
  end
end
```

The number of loop iterations is $O(p)$ and in each iteration, the pow2 call is $O(x)$. So the total number of operations is bounded by $O(px)$ but $x < p$ so this is also bounded by $O(p^2)$ which is $O(4^n)$ as the size of p is $log_2 p$.

El Gamal with private key x
public key (p, g, y) with $y = g^x \mod p$
cipher (a, b) with $a = g^k \mod p$ and $b = My^k \mod p$
message $M = b/(a^x) \mod p = b(a^x)^{-1} \mod p$

bubble sort

```
Algorithm bubbleSort(A):
  Input: An (unsorted) array A
  Output: An sorted array A

  n = length(A)
  swapped = true
  while swapped
    swapped = false
    for i from 0 to n
      if a[i] > a[i + 1]
        swap(a[i], a[i + 1])
        swapped = true
end
```

For each element in the array, bubbleSort does $n-1$ comparisons which is $O(n)$ and there are n elements in the array so bubbleSort has a total running time of $\boldsymbol{O(n^2)}$.

merge sort

```
Algorithm merge(L, R):
  Input: Two sorted arrays L and R
  Output: An sorted array of L and R

  if L = []
    return R
  if R = []
    return L
  a = L[1], b = R[1]
  L' = L without a, R' = R without b
  if a <= b
    return [a] + merge(L', R)
  return [b] + merge(L, R')
end
```

When merge(L, R) is called, at most one recursive call is made, in which $|L| + |R|$ decreases by 1. Therefore, at most $O(n)$ recursive calls are made, where $n = |L| + |R|$ is the length of the input and since a constant number of operations are executed for each recursive call, it takes at most $O(n)$ time to run.

```
Algorithm mergeSort(X):
  Input: An (unsorted) array X
  Output: An sorted array X

  if |X| <= 1
    return X
  split X into two halves, X = L + R
  return merge(mergeSort(L), mergeSort(R))
end
```

The total lengths of lists processed at each level of recursion is constant at $|X| = n$ and the total amount of work done for each call is linear in the lengths of the arguments. The number of times X can be halved is $O(\log n)$ hence the time complexity of mergeSort is $\boldsymbol{O(n \log n)}$.

quick sort

In the algorithm, p will be our pivot.

```
Algorithm quickSort(L):
  Input: Array to be sorted L
  Output: An sorted array of L

  if length(L) <= 1
    return L
  remove first element, p, from L
  A = elements in L that are <= p
  B = elements in L that are > p
  L = quickSort(A)
  R = quickSort(B)
  return L + p + R
end
```

The worst case occurs when for each recursive call, one of A or B is empty. Let n be the size of our array L. Then n recursive calls are made, with the argument one element shorter each time. Before each recursive call, A and B must be calculated which requires $O(n)$ steps. So the total work done is $n + (n-1) + ... + 1 = \frac{1}{2}n(n+1)$. Hence quick sort is in $\boldsymbol{O(n^2)}$.

bucket sort

Suppose we wanted to sort n items whose keys are integers in the range $[0, N-1]$ for some integer $N \geq 2$. For example, we want to sort the two-digit numbers $[15, 45, 10, 30, 25, 28, 15, 50, 36]$ into ascending order of the first digit then bucket sort will return $[15, 10, 15, 25, 28, 30, 36, 45, 50]$. Some implementations will use another algorithm to sort each bucket itself.

```
Algorithm bucketSort(S):
  Input: S with keys in [0, N - 1]
  Output: S sorted in order of keys

  B array of N empty lists
  foreach x in S
    k = key of x
    remove x from S
    add x to B[k]
  for i = 1 to N
    sort(B[i])
  for i = 1 to N
    for each x in B[i]
      remove x from B[i]
      add x to end of S
end
```

The worse case for bucket sort is when all elements are allocated to the same bucket and we get $\boldsymbol{O(n^2)}$. Since individual buckets are sorted using another algorithm, if only a single bucket needs to be sorted, bucket sort will take on the complexity of the inner sorting algorithm.

determinants and permanents

permutations
A **permutation** is a 1-1 map of a set X onto itself. The number of permutations on an n-element set is $n!$. A simple inductive proof shows that, for all $n \geq 4$, $2^n \leq n! \leq 2^{n^2}$. That is: $n \mapsto n!$ is $\Omega(2^n)$ and $O(2^{n^2})$.

transpositions
A **transposition** is a permutation of two elements, i.e $\sigma = (\alpha\beta)$.
parity
The **parity** of a permutation σ, denoted $sgn(\sigma)$ is 1 if σ is the product of an even number of transpositions, -1 otherwise.

proof that $sgn(\sigma) = \pm 1$
Let $\sigma \in S_n$. Write $\sigma = (\alpha_1\alpha_2 \cdots \alpha_m)$. In the view of

$$(\beta_1\beta_2\beta_3 \cdots \beta_k) = (\beta_1\beta_2)(\beta_1\beta_3)\cdots(\beta_1\beta_k)$$

any permutation of length k can be written as a composite of $k-1$ transpositions. Now consider when k is even or odd, then the result follows.

Note that $sgn(\sigma \cdot \tau) = sgn(\sigma) \cdot sgn(\tau)$ and $sgn(t) = -1$, where t is a transposition.

matrices

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

$$det(A) = \sum_{\sigma \in \text{perm}\{1,\cdots,n\}} sgn(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$permanent(A) = \sum_{\sigma \in \text{perm}\{1,\cdots,n\}} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

calculating the determinant
We can convert the matrix into upper triangular form, then we know the determinant is the product of the elements across the diagonal. To zero the i column below the diagonal, we need to do a transposition of columns $O(n)$ and for each of $(n-i+1) \leq n$ rows below the ith row, subtract a multiple of the ith row from that row, which contains $n-i+1 \leq n$ non-zero elements, so $O(n)$ operations per row. So cost of zeroing the ith column below the diagonal is $O(n^2)$. There are $n-1 \leq n$ columns to zero below the diagonal, so cost of converting to UT form is $O(n^3)$ and cost of multiplying diagonal elements is $O(n)$. Hence total cost is $O(n^3 + n) = \boldsymbol{O(n^3)}$.

calculating the permanent
There are no efficient methods to calculate the permanent - the best-known algorithms run in exponential time.

lexicographic order

Consider a finite set A which is totally ordered. Given two different elements of the same length $\alpha_1\alpha_2 \cdots \alpha_k$ and $\beta_1\beta_2 \cdots \beta_k$, the first sequence is smaller than the second one for lexicographic order, if $a_i < b_i$ for the first i where a_i and b_i are different.

If one sequence is shorter than another, then pad it with "blank" characters - a character than is treated as smaller than every element of A.

$\boldsymbol{\Omega(n \log n)}$ for comparison-based algorithm

Suppose we want to sort n elements. There are $n!$ permutations of these n elements. If we draw a binary tree with each leaf represents a permutation of these n elements, the number of comparisons we need at most is the height of tree. A tree with height h has at most 2^h leaves, then we have $n! \leq 2^h$, it follows that $log(n!) \leq h$. In the view of

$$n! > (\frac{n}{2})^{\frac{n}{2}} \text{ for } n \geq 1$$

we know that $h \geq log(n!) \geq log(\frac{n}{2})^{\frac{n}{2}} = (\frac{n}{2})\, log\, \frac{n}{2}$ so it follows that $h \in \Omega(n \log n)$.

notes