Chapter 5
Recursive Algorithms

Exponentiation
- \( A^{2 \times M} = (A^M)^2 \)
- \( A^M \times N = A^M \times A^N \)

```java
quickPower(A, N) {
  if (N == 1) return A;
  if (N is even)
    B = quickPower(A, N/2);
    return B*B;
  return A*quickPower(A, N-1);
}
```

Q: How many times quickPower uses *?
A: If binary form of \( N \) is \((b_k, \ldots, b_1, b_0)_2\) the answer is \( k - 1 + b_k + \ldots + b_1 + b_0 \)

```java
slowPower(A, N)
X = A
for j = 2 to N
  X = X*A
return X
```

Evaluation of Polynomial
- \( P(x) = 3x^5+2x^4+7x^3+8x^2+2x+4 \)
  - \( t_1 = (3^x+x^x+x^x) \)
  - \( t_2 = t_1 + (2^x+x^x+x^x) \)
  - \( t_3 = t_2 + (7^x+x^x+x^x) \)
  - \( t_4 = t_3 + (8^x+x^x) \)
  - \( P = t_4 + (2^x) + 4 \)
- 15 Multiplications, 5 Additions

A Little Smarter
- \( P(x) = 3x^5+2x^4+7x^3+8x^2+2x+4 \)
  - \( t_1 = 4; \quad x^p = x; \)
  - \( t_2 = (2^x*x^p) + t_1; \quad x^p = x^p * x; \)
  - \( t_3 = (8^x*x^p) + t_2; \quad x^p = x^p * x; \)
  - \( t_4 = (7^x*x^p) + t_3; \quad x^p = x^p * x; \)
  - \( t_5 = (2^x*x^p) + t_4; \quad x^p = x^p * x; \)
  - \( P = (3^x*x^p) + t_5; \)
- 9 Multiplications, 5 Additions

Horner’s Rule
- \( P(x) = 3x^5+2x^4+7x^3+8x^2+2x+4 \)
  - \( t_1 = (3^x) + 2 \)
  - \( t_2 = (t_1^x) + 7 \)
  - \( t_3 = (t_2^x) + 8 \)
  - \( t_4 = (t_3^x) + 2 \)
  - \( P = (t_4^x) + 4 \)
- 5 Multiplications, 5 Additions

Horner’s Rule for Polynomial
Any polynomial in the form \( a_0 + x^1a_1 + x^2a_2 + \ldots + x^na_n \) can be written as \( a_n + x(a_n + x(a_n + x(... + x(a_n + x(a_n))...))) \)
In this form we will reduce the number of computations that need to be done.

```java
HornerRule(a[0..n], x)  {
  // Pre: a[0..n] is coefficients \( a_0, a_1, \ldots, a_n \)
  // Post: return \( a_n + x(a_n + x(a_n + x(... + x(a_n + x(a_n))...))) \)
  p = a[n];
  for (i = n – 1; i > 0; i--) p = a[i] * x + p;
  return p;
}
```
Greatest Common Divisors

```java
static int GCD(int A, int B) {
    // Pre: A>B>=0.
    // Post: return gcd(A, B)
    // gcd(A, 0) = A
    if (B == 0) return A;

    int remainder = A % B;  // %: mod
    return GCD(B, remainder);
}
```

**Runtime:**
- O(log(A+B)) if "%" takes O(log2(A+B))
- return GCD(B, remainder);

**Claim:** # of nested calls of GCD(A, B) is O(log(A)).

- **Claim:** Let \( A_0, B_0, A_1, B_1, \ldots, A_n, B_n \) be the inputs to the consecutive recursive calls of GCD(A,B), where
  - \( A_i > B_i \) for all \( i = 0, \ldots, m \).
  - \( A_i = B_i \) \( \mod B_i \) for \( i = 1, \ldots, m \).
  - Then \( A_{i+1} < \frac{1}{2} A_i \) for \( i=2, \ldots, m \).

- **Proof:** Because \( A_i = B_i \mod B_i \):
  - Case 1: \( B_i > \frac{1}{2} A_i \), then \( A_i = A_i \mod B_i \leq \frac{1}{2} A_i \);
  - Case 2: \( B_i < \frac{1}{2} A_i \), then \( A_i = A_i \mod B_i \leq \frac{1}{2} A_i \).

- If \( 2^{n-1} < A_0 \leq 2^n \), then \( 2^n < 2A_0 \). or \( n < \log_2 A_0 + 1 \).
- If \( 2^{n-1} < A_0 \leq 2^n \), then \( A_0 < 2^n, A_0 = 2^n, \ldots, A_n < 2^{m/2} \).
- Since \( n-m/2 \geq 0, m \leq 2n < 2(\log_2 A_0 + 1) \).

How to show GCD is correct?

- **Definition 1:** \( d \) is a divisor of \( b \), written \( d \mid b \), if there exists an integer \( c \) such that \( b = d \cdot c \).
- **Definition 2:** \( \gcd(a, b) = \max \{ \gcd(d, b) \mid d \in \{ d : d \mid a, d \mid b \} \} \), the set of common divisors.

- **Claim 1:** \( d \mid 0 \) for any \( d \).
- **Claim 2:** \( \gcd(d, 0) = d \).
- **Claim 3:** If \( d \mid (x+y) \) and \( d \mid x \), then \( d \mid y \).
- **Claim 4:** \( \gcd(a, b) = \gcd(b, a \mod b) \).

How to compute \( x/2^n \), \( x \mod 2^n \)?

- Using bit-wise and bit-shift operators.
- In java/c++/c#, the following operators are provided:
  - The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0".
  - The left shift operator "<<" shifts a bit pattern to the left.
  - The right shift operator ">>" shifts a bit pattern to the right.
  - The bitwise & operator performs a bitwise AND operation.
  - The bitwise ^ operator performs a bitwise exclusive OR operation.
  - The bitwise | operator performs a bitwise inclusive OR operation.

- \( y = x/2^n \) is done by \( y = x \gg n \); \( y = x \mod 2^n \) is done by \( y = x \& (\{1\ll n\} - 1) \).

Generating Permutations

- Generating all permutations of the numbers 1, 2, \ldots, \( n \).

- Based on the assumption that if we can generate all the permutations of \( n-1 \) numbers, then we can get algorithms for generating all the permutations of \( n \) numbers.

Generating Permutations

- Let \( N \) be the set \( \{1, 2, 3, \ldots, n\} \).
- There are \( n! \) different permutations on \( N \).
- How to enumerate all of them? E.g., if \( n = 5 \), how can we enumerate them from \( [1, 2, 3, 4, 5] \) to \( [5, 4, 3, 2, 1] \)?
- How about the permutation of \( m \) elements from \( n \) elements?
- There are \( n!/((n-m)! \cdot m!) \) m-permutations out of \( n \) elements.
- If \( n = 5 \) and \( m = 3 \), how to enumerate them from \( [1, 2, 3] \) to \( [5, 4, 3] \)?
A Straightforward Solution

- Let N be the set \{1, 2, ..., n\}.
- If m is fixed, we can use \(\text{m}\times\text{n}\) nested loops:

```java
public static void ThreePerm (int n) {
    // Pre: m = 3
    for (int x1 = 1; x1 <= n; x1++)
        for (int x2 = 1; x2 <= n; x2++)
            if (x1 != x2)
                System.out.print(x1+x2+x3);
}
```

Complexity: \(O(n^3)\)

Generating m-Permutations

- If we can generate all the permutations of \(\text{n}\)-1 numbers, then we can get algorithms for generating all the permutations of \(\text{n}\) numbers by inserting \(\text{n}\) at positions 1 to \(\text{n}\) in all the permutations of \(\text{n}\)-1 numbers.

  Recursive Algorithm 1:
  - Suppose we generate all the \((\text{m}\times\text{n})\)-permutations of the numbers 1, 2, ..., \(\text{n}\). For each number \(\text{n}\) not in that permutation, we put it at position \(\text{n}\) to obtain a \(\text{m}\)-permutation.

Generating Permutations

- If we can generate all the permutations of first \(\text{n}\)-1 numbers, then we can get algorithms for generating all the permutations of \(\text{n}\) numbers by inserting \(\text{n}\) at positions 1 to \(\text{n}\) in all the permutations of \(\text{n}\)-1 numbers.

  Algorithm 2:
  - Repeat this procedure until finally the permutations of 1, 2, ..., \(\text{n}\)-1 are generated in array[1..n-1] and the number \(\text{n}\) is added array[n].
  - Generate all the permutations of the numbers 1, 2, 3, ..., \(\text{n}\)-1 in array[1..n-1] and array[n], and add the number \(\text{n}\) at array[n].
  - ...
  - Generate all the permutations of the numbers 1, 2, 3, ..., \(\text{n}\)-1 in array[1..n-1] and array[n], and add the number \(\text{n}\) at array[1].
  - Generate all the permutations of the numbers 1, 2, 3, ..., \(\text{n}\)-1 in array[2..n], and add the number \(\text{n}\) at array[1].

Generate all m-permutations of n elements

- Input: positive integers \(\text{m}\) and \(\text{n}\), \(1 \leq \text{m} \leq \text{n}\);
- Output: All m-permutations of the numbers 1, 2, ..., \(\text{n}\);

```java
public static void ThreePerm2 (int n) {
    // Pre: \text{m} = 3
    int P[3];
    for (int x1 = 1; x1 <= n; x1++)
        for (int x2 = 1; x2 <= n; x2++)
            if (x1 != x2)
                System.out.print(x1+x2+x3);
    }
```

```
Generating m-Permutations

- What’s the performance of the algorithm Generating Permutations?
  - Time Complexity?
  - \(\Theta(n!/(n-m)!)\)
  - Space Complexity?
  - \(\Theta(n)\)

Generating m-Permutations

- Input: A positive integer \(\text{n}\);
- Output: All permutations of the numbers 1, 2, ..., \(\text{n}\);

```java
1. for \(\text{x1}=\text{n}\) to \(\text{n}\) \(\text{P}[\text{x1}]=\text{P}[\text{x1}]0;\) // all positions are empty
2. perm2(n); // for each \(\text{x1}=\text{n}\) put \(\text{n}\) into a position of \(\text{P}[1..\text{n}]\)
perm2(k) { // Pre: \(\text{0} \leq \text{k} \leq \text{n}\); there are \(\text{n}\) positions to be filled; the numbers
    // between \(\text{k+1}\) and \(\text{n}\) already took their positions in \(\text{P}[1..\text{n}]\).
    // Post: the numbers between \(\text{k}\) and \(\text{n}\) took their positions in \(\text{P}[1..\text{n}]\).
    1. if \(\text{k} = \text{0}\) then output \(\text{P}[1..\text{n}]\);
    2. else
        3. for \(\text{x1} = \text{n}\) downto \(\text{k}+1\) if \(\text{P}[\text{x1}] = \text{0}\) {
            \(\text{P}[\text{x1}] = \text{k};\) // Place number \(\text{k}\) at position \(\text{x1}\)
            perm2(k-1); // Generate permutations for all numbers < \(\text{k}\)
            \(\text{P}[\text{x1}] = \text{0};\) // release position \(\text{x1}\)
        } // end for
    7. }
8. }
```
Generating Permutations

- What’s the performance of the algorithm Generating Permutations?
  - Time Complexity?
  - $\Theta(n!)$
  - Space Complexity?
  - $\Theta(n)$

Finding the Majority Element

- Let $A[1...n]$ be a sequence of integers. An integer $a$ in $A$ is called the majority if it appears more than $\lceil n/2 \rceil$ times in $A$.

  - For example:
    - Sequence 1, 3, 2, 3, 4, 3: 3 is the majority element since 3 appears 4 times which is more than $\lceil n/2 \rceil$
    - Sequence 1, 3, 2, 3, 4: 3 is not the majority element since 3 appears three times which is equal to $\lceil n/2 \rceil$, but not more than $\lceil n/2 \rceil$.

How to Generate m-Combinations

- Let $N$ be the set \{1, 2, 3, …, $n$\}.
- How to generate combinations of $m$ elements from $n$ elements?
  - There are $n!/(m!(n-m)!)$ $m$-combinations out of $n$ elements.
  - If $n = 5$ and $m = 3$, how to enumerate them from [1, 2, 3] to [3, 4, 5]?

Finding the Majority Element

- If two different elements in the original sequence are removed, then the majority in the original sequence remains the majority in the new sequence.
  - The above observation suggests the following procedure for finding an element that is a candidate for being the majority.
Finding the Majority Element

- Let $x = A[1]$ and set a counter to 1.
- Starting from $A[2]$, scan the elements one by one increasing the counter by one if the current element is equal to $x$ and decreasing the counter by one if the current element is not equal to $x$.
- If all the elements have been scanned and the counter is greater than zero, then return $x$ as the candidate.
- If the counter becomes 0 when comparing $x$ with $A[j]$, $1 < j < n$, then call procedure candidate recursively on the elements $A[j+1...n]$.

Finding the Majority Element

- Why just return $x$ as a candidate?

```
Input: An array $A[1...n]$ of $n$ elements;
Output: The majority element if it exists; otherwise none;
1. $x \leftarrow \text{candidate}(1);
2. count \leftarrow 0;
3. for $j \leftarrow 1$ to $n$
   4. if $A[j] = x$ then $count \leftarrow count + 1$;
   5. end for;
   6. if $count \geq \lfloor n/2 \rfloor$ then return $x$;
   7. else return none;
\text{candidate}(m)
1. $j \leftarrow m; x \leftarrow A[m]; count \leftarrow 1;
2. while $j < n$ and count $> 0$
   3. $j \leftarrow j + 1;
   4. if $A[j] = x$ then count $\leftarrow count + 1$;
   5. else count $\leftarrow count - 1$;
   6. end while;
   7. if $j = n$ then return $x$;
   8. else return candidate[$j+1$];
```