算法导论复习7——经典分治策略

Chapter 30 快速傅里叶变换

单位复数根

- $\omega_n = e^{2\pi i/n}$
- $\omega_n^{n/2} = -1$
- 消去定理：（上下消去公因子）

  引理 30.3(消去引理) 对任何整数 $n\geqslant 0$，$k\geqslant 0$，以及 $d>0$，
  $$\omega_{dn}^{dk} = \omega_n^k$$

- 折半定理：

  引理 30.5(折半引理) 如果 $n>0$ 为偶数，那么 n 个 n 次单位复数根的平方的集合就是 $n/2$ 个 $n/2$ 次单位复数根的集合。

  证明 根据消去引理，对任意非负整数 k，我们有 $(\omega_n^k)^2=\omega_{n/2}^k$。注意，如果对所有 n 次单位复数根进行平方，那么获得每个 $n/2$ 次单位根正好 2 次，因为
  $$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$$
  因此，ω_n^k 与 $\omega_n^{k+n/2}$ 平方相同。我们也可以由推论 30.4 来证明该性质，因为 $\omega_n^{n/2}=-1$ 意味着 $\omega_n^{k+n/2}=-\omega_n^k$，所以 $(\omega_n^{k+n/2})^2=(\omega_n^k)^2$。 ■

  折半定理保证了每次分解子问题后的规模只有一半

- 求和定理：

  引理 30.6(求和引理) 对任意整数 $n\geqslant 1$ 和不能被 n 整除的非负整数 k，有
  $$\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$$

快速傅里叶变换

- 将多项式系数分解为偶数项和奇数项，这样就把一个多项式求值转换为了两个多项式求值

  FFT 利用了分治策略，采用 $A(x)$ 中偶数下标的系数与奇数下标的系数，分别定义两个新的次数界为 $n/2$ 的多项式 $A^{[0]}(x)$ 和 $A^{[1]}(x)$：
  $$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$
  $$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$
  注意到，$A^{[0]}(x)$ 包含 A 中所有偶数下标的系数(下标的相应二进制表达的最后一位为 0)，以及 $A^{[1]}(x)$ 包含 A 中所有奇数下标的系数(下标的相应二进制表达的最后一位为 1)。于是有
  $$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{30.9}$$

- 根据折半引理和消去定理，x^2 在下一层递归中恰好是周期为一半的单位根

```
RECURSIVE-FFT(a)
n = a.length              // n is a power of 2
if n == 1
    return a
ω_n = e^{2πi/n}
ω = 1
a^[0] = (a_0, a_2, ..., a_{n-2})
a^[1] = (a_1, a_3, ..., a_{n-1})
y^[0] = RECURSIVE-FFT(a^[0])
y^[1] = RECURSIVE-FFT(a^[1])
for k = 0 to n/2 - 1
    y_k = y_k^[0] + ω y_k^[1]
    y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
    ω = ω ω_n
return y                  // y is assumed to be a col
```

- 逆FFT：

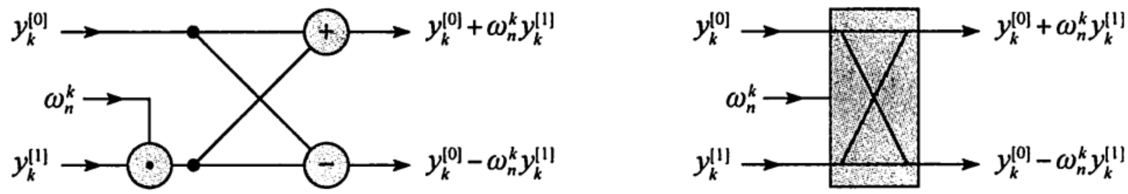
$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

使用折半引理依然分治计算，最后除以n，这是因为

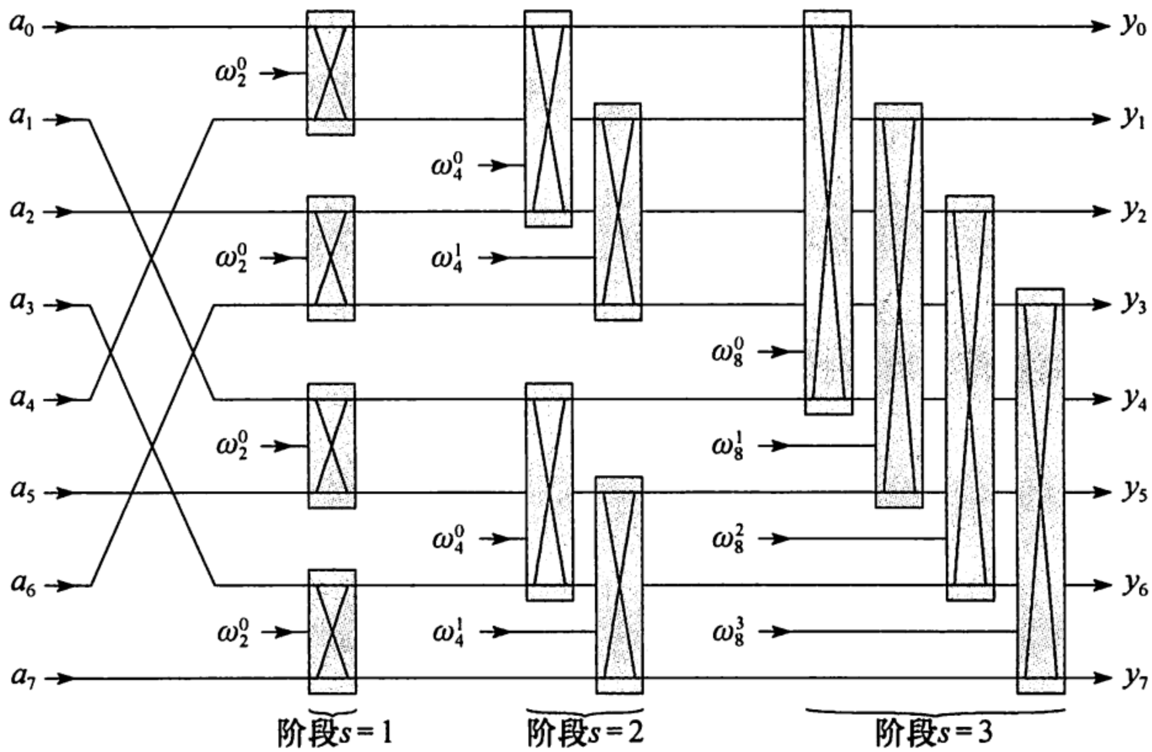
定理 30.7 对 j，$k=0$，1，…，$n-1$，V_n^{-1} 的(j,k)处元素为 ω_n^{-kj}/n。

高效FFT实现

- 蝴蝶操作：



（a）　　　　　　　　　　　　（b）



- FFT的迭代实现

```
ITERATIVE-FFT(a)
BIT-REVERSE-COPY(a, A)
n = a.length            // n is a power of 2
for s = 1 to lg n
    m = 2^s
    ω_m = e^(2πi/m)
    for k = 0 to n-1 by m
        ω = 1
        for j = 0 to m/2-1
            t = ωA[k+j+m/2]
            u = A[k+j]
            A[k+j] = u+t
            A[k+j+m/2] = u-t
            ω = ωω_m
return A
```

Chapter 32 字符串匹配

朴素算法

```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n-m
        if P[1..m] == T[s+1..s+m]
                print "Pattern occurs with shift" s
```

- 时间复杂度：

  模式串长度为m，查找串长度为n，则时间复杂度为$O(mn)$

Rabin-Karp算法

- 在不考虑整数范围的情况下：
  - 将模式串看做一个d进制的数字**（小索引在高位）**，其中d是字母表总长度，求出这个数字时间复杂度为$O(m)$：

    $$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$$

  - 将查找串的每m位看成一个d进制的数字，共有n-m+1个数字。第0个数字需要$O(m)$的时间求出，其余串每个需要$O(1)$时间：

    $$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

  - 将模式串数字与所有查找串数字比较，时间复杂度为$O(n-m+1)$
  - 总时间复杂度为$O(n+m)$
- 在考虑整数范围的情况下：
  - 每一次更新p和t时使用一个大素数（其10倍恰好为字长）来**取模**，可以保证正确性：

    $$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

  - 但取模只能筛掉一定无效的偏移，找到可能有效的位置后，需要进一步比较

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                          // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n-m                        // matching
    if p == t_s
        if P[1..m] == T[s+1..s+m]
            print "Pattern occurs with shift" s
    if s < n-m
        t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

- 当匹配点比较少（常数个）时，时间复杂度为$O(n-m+1+cm)=O(n+m)$

KMP算法

- 核心思想：没有重复前缀是最好的情况
- 模式串的前驱：
  - 前提：使用q+1的位置进行匹配
  - 意义：q+1点失配后，应该让哪个点挪到q的位置以继续匹配q+1点
  - 性质：前驱一定在当前点之前，即q的前驱一定小于q
  - 直观意义：能让字符串挪动的最远距离，如果有很多相同匹配点，应该选择编号最大的点。**如果q为0，那么这个点没有前驱，应该跳过挪动模式串的操作**
  - 求法：使用匹配的方法和自己的第二个位置进行匹配求解即可，每次q结束循环后就可以确定q的前驱

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
        while k > 0 and P[k+1] ≠ P[q]
                k = π[k]
        if P[k+1] == P[q]
                k = k + 1
    π[q] = k
return π
```

- KMP匹配
  - 整个算法过程中，q只能最多自增m-1次，而循环一共进行n次，根据聚合分析，时间复杂度为 $O(m+n)$
  - 查找串向前移动的时机：成功匹配了一位或不匹配时挪到不能再挪

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0
for i = 1 to n
```

```
        while q>0 and P[q+1]≠T[i]
            q=π[q]
        if P[q+1] == T[i]
            q=q+1
    if q == m
            print "Pattern occurs with shift" i−m
    q=π[q]
```