بسم الله الرحمن الرحيم

چکیده

در این پروژه در مورد الگوریتم FFT بحث شده است ، الگوریتمی که دو چندجمله با درجه n را در زمان $O(n \log n)$ در یکدیگر ضرب کرده و حاصل را به دست می آورد. با استفاده از این الگوریتم مسائل مهمی که در فصل های انتهایی معرفی شده اند در زمانی بهینه قابل حل هستند.

برای شناخت نحوه عملکرد این الگوریتم ابتدا مباحث ریاضی مورد نیاز معرفی شده است که شامل خواصی از میدان اعداد مختلط، ماتریس واندرموند و خواص آن می باشد. همچنین در مورد الگوریتم محاسبه x^n در زمان $O(\log n)$ به طوری که x عضوی از یک حلقه جبری باشد بحث شده است. سپس به معرفی الگوریتم FFT پرداخته شده است که با استفاده از روش تقسیم و غلبه دو چند جمله ای را در زمان ذکر شده در یکدیگر ضرب می نماید. سپس به معرفی چند مسئله پرداخته شده است که بعضی از آن ها نیز نحوه حلشان با استفاده از FFT مورد بحث قرار گفته است.

همچنین مسئله «دزد حریص» معرفی شده است که با استفاد از الگوریتم FFT و الگوریتم محاسبه x^n راه حلی برای این مسئله ارائه شده است.

فهرست مطالب

فصل اول: مقدمه

دنیای علوم کامپیوتر به عنوان علمی جدانشدنی از ریاضیات همواره با مسائل مختلف و متنوع در طول تاریخ درگیر بوده است و همواره در پی یافتن راه حل های بهینه تر و کارآمدتر می باشد که اکثر این مسائل در دنیای واقعی بر خلاف ظاهر تئوری شان بسیار کاربردی می باشند.

در این پروژه قرار است به یکی از الگوریتم های مهم و در عین حال پیچیده و زیبا در علوم کامپیوتر پرداخته شود.

الگوریتم FFT یکی از مهم ترین کشفیات علم علوم کامپیوتر بوده است، مبنای اصلی این الگوریتم تبدیلات فوریه می باشد (Fast Fourier Transform) که از این تبدیلات در پردازش سیگنال های دیجیتال همواره استفاده می شده است تا این که در مقاله ای توسط J.W.Tukey و J.W.Cooley مورد استفاده قرار گرفتند و منجر به کشف یک الگوریتم کارآمد برای ضرب دو چند جمله ای شدند.

به علت مباحث ریاضی محض گسترده مورد نیاز برای فهم این الگوریتم در فصل های ابتدایی به شرح مباحث پایه ریاضیات از جمله میدان اعداد مختلط و خواص آن ها و همچنین جبرخطی و ماتریس مشهور واندروموند پرداخته شده است که برای تشریح الگوریتم FFT و نحوه کارکرد آن مورد نیاز می باشد. توصیه می شود که فصل های اولیه با دقت مطالعه شود.

سپس به تشریح الگوریتم FFT می پردازیم و چند مسئله کاربردی را تشریح می کنیم و چند مسئله دیگر را نیز معرفی می نماییم.

فصل دوم: اعداد مختلط

برای طرح مطالب اصلی و فهم الگوریتم پیش رو در فصل های بعد نیاز است نگاهی به برخی از خواص میدان اعداد مختلط داشته باشیم.

میدان اعداد مختلط ($\mathbb{C}$) به واقع تعمیمی است از میدان اعداد حقیقی ($\mathbb{R}$) به عبارت دیگر میدان $\mathbb{R}$ زیر میدانی از $\mathbb{C}$ می باشد.

ساختار هندسی اعداد مختلط:

هر عدد مختلط به صورت $z=a+bi$ را می توان به صورت زوج مرتب (a,b) در نظر گرفت هر عدد در واقع برداری ست در صفحه مختلط با طول a (محور حقیقی) و عرض b (محور موهومی).

برای مثال عدد مختلط i برابر است با $(0,1)$

توابع Re و Im

دو تابع Re و Im را نیز این گونه تعریف می نمایند که برای عدد مختلط $z=a+bi$ ، $Re(z)=a$ و $Im(z)=b$.

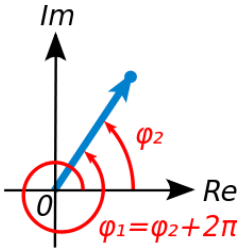
قدر مطلق z

قدر مطلق z را طول بردار z در صفحه مختلط تعریف می نمایند لذا $|z|=\sqrt{a^2+b^2}$ زمانی که $z=a+bi$

آرگومان z

فرض کنید $z=(r,\theta)$ مختصات قطبی عدد مختلط z باشد به این معنی که $r=|z|$ و θ همان زاویه ای است که بردار z با جهت مثبت محور حقیقی صفحه مختلط می سازد به طوری که $-\pi<\theta\leq\pi$ که در واقع θ را آرگومان اصلی z تعریف نموده و با $Arg(z)$ نمایش می دهند و همچنین آرگومان z با $arg(z)$ نمایش می دهند و به صورت زیر تعریف می شود.

$$arg(z)=Arg(z)+2k\pi \qquad (k=0,\pm1,\pm2,\dots)$$



شکل 1

نمایش قطبی اعداد مختلط

فرض کنید $z=a+bi$ و $|z|=r$ و $\theta=arg\ z$ به وضوح $cos\ \theta=\frac{a}{r}$ و همچنین $sin\ \theta=\frac{b}{r}$ و بنابراین $\theta=tan^{-1}\frac{b}{a}$

داریم:

$z=r\cos\theta + (r\sin\theta)i=r(\cos\theta+i\sin\theta).$ (1)

همچنین فرمول اویلر به صورت زیر تعریف می شود:

$e^{i\theta}=\cos\theta+i\sin\theta.$ (2)

حال می توان طبق (1)و(2) عدد مختلط z را به صورت زیر در نظر گرفت:

$z=re^{i\theta}.$

ضرب اعداد مختلط

حال طبق مرجع 1 ضرب دو عدد مختلط $z_1 = r_1 e^{i\theta_1}$ و $z_2 = r_2 e^{i\theta 2}$ به صورت زیر می باشد:

$$z_1.z_2 = r_1 r_2 e^{i(\theta_1+\theta_2)}$$

به عبارتی دیگر اندازه بردار حاصل ضرب دو عدد مختلط می شود حاصل ضرب اندازه دو بردار و آرگومان بردار حاصل ضرب نیز برابر است با جمع آرگومان دو بردار.

ریشه های n ام واحد

طبق منبع 1 ریشه های n ام $z = re^{i\theta}$ به صورت زیر می باشند:

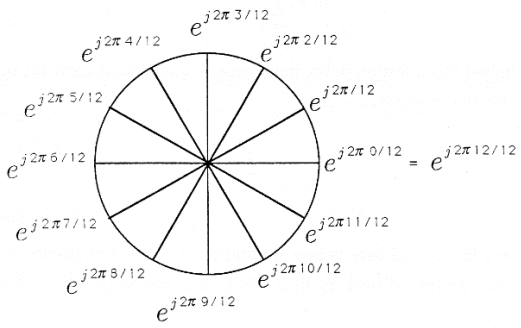
$$\sqrt[n]{r}e^{i(\frac{2k\pi}{n}+\frac{\theta}{n})}, k \in \mathbb{Z}$$

که طبق آنچه در قسمت قبل گفته شد به راحتی می توان بررسی کرد که هر کدام از این ریشه ها به توان n برسند به z تبدیل می شوند.

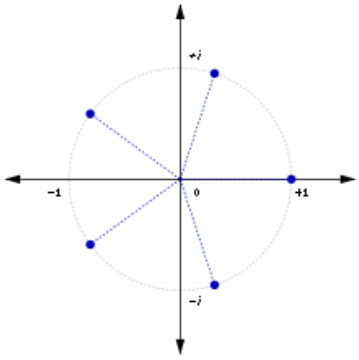
پس ریشه های n ام واحد به صورت زیر در می آیند:

$$\sqrt[n]{1} = e^{i\frac{2k\pi}{n}}$$

که در واقع همگی آن ها روی دایره واحد می باشند و همچنین اگر دایره واحد را به n قطاع مساوی تقسیم کنیم به طوری که یکی از خطوط جداکننده قطاع ها روی قسمت مثبت محور حقیقی باشد ریشه های n ام واحد مشخص می شوند.



شکل 2 : ریشه های 12 ام واحد



شکل 3: ریشه های 5 ام واحد

همچنین می توان همه ریشه های n ام واحد را بر حسب توان هایی از یکی از ریشه ها نشان داد.

یکی از ریشه ای n ام واحد ریشه $e^{i\frac{2\pi}{n}}$ می باشد که در واقع اولین ریشه در حرکت در جهت مثبت مثلثاتی از 1 روی دایره واحد می باشد. این ریشه را ω_n می نامند که در واقع ریشه های nام واحد به صورت زیر می باشند:

$$\{\omega_n, \omega_n^2, \omega_n^3, ..., \omega_n^n = 1\}$$

در واقع چون ریشه های n ام واحد همگی قدرمطلق یک دارند و آرگومان آن ها نیز ضریبی از $\frac{2\pi}{n}$ می باشد. تشکیل یک مجموعه بسته ضربی می دهند که به وسیله ω_n قابل تولید است.

مطالب گفته شده همگی بخش هایی از خصوصیات اعداد مختلط بودند که در فصل های بعد کاربرد خواهند داشت.

فصل سوم: ماتریس واندرموند

در این فصل قرار است کمی با خصوصیات ماتریس واندرموند آشنا شویم ، که در نوع خود یک ماتریس جالب و کاربردی می باشد.

ماتریس واندرموند به فرم زیر می باشد:

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{bmatrix}$$

اگر تمام x_i ها متمایز باشند طبق منبع 2 دترمینان همواره ناصفر است بنابراین ماتریس فوق وارون پذیر می باشد.

لازم به ذکر است که x_i ها هر مقدار دلخواهی از میدانی که ماتریس واندرموند روی آن ایجاد شده باشد می توانند بگیرند.

همچنین طبق منبع 2 دترمینان این ماتریس به صورت زیر محاسبه می شود:

$$\det(V) = \prod_{1 \le i < j \le n} (x_j - x_i)$$

همچنین وارون این ماتریس طبق منبع 3 به صورت زیر می باشد:

$$b_{ij} = \begin{cases} (-1)^{j-1} \left(\dfrac{\displaystyle\sum_{\substack{1 \le m_1 < \ldots < m_{n-j} \le n \\ m_1, \ldots, m_{n-j} \ne i}} x_{m_1} \cdots x_{m_{n-j}}}{x_i \displaystyle\prod_{\substack{1 \le m \le n \\ m \ne i}} (x_m - x_i)} \right) & : 1 \le j < n \\ \dfrac{1}{x_i \displaystyle\prod_{\substack{1 \le m \le n \\ m \ne i}} (x_i - x_m)} & : j = n \end{cases}$$

فصل چهارم: محاسبه x^n در یک حلقه جبری با زمان کارآمد

هدف از این فصل همان طور که از عنوان مشخص است محاسبه x^n با زمان $O(\log n)$ می باشد.

برای سادگی در ابتدا فرض کنید که n توانی از 2 می باشد. از آنجایی که x عضو یک حلقه جبری است لذا عمل ضرب دارای خاصیت جبری شرکت پذیری می باشد پس ترتیب ضرب کردن ها در محاسبه $\underbrace{xxxx...x}_{n}$ اهمیتی ندارد.

حال فرض کنید تابع $f(x,n)$ تابعی است که قرار است مقدار x^n را محاسبه نماید طبق آنچه گفته شد و خاصیت شرکت پذیری می توان ضرب را این گونه در نظر گرفت $(\underbrace{xxx...x}_{n/2}).(\underbrace{xxx...x}_{n/2})$ لذا اگر مقدار $p=f(x,n/2)$ را داشته باشیم آنگاه می توان با محاسبه $p^2=f(x,n)$ جواب را یافت اگر این عمل را به صورت بازگشتی نیز انجام دهیم یعنی $f(x,n/2)=(f(x,n/4))^2$ آنگاه برای محاسبه پیچیدگی زمانی می توان گفت:

$$T(n)=T(n/2)+1$$

اگر $T(n)$ زمان اجرای تابع $f(x,n)$ باشد. پس

$$T(n)=\theta(\log n)$$

حال حالتی را در نظر بگیرید که n عددی دلخواه باشد. در هر مرحله دو حالت پیش خواهد آمد:

حالت اول: n زوج باشد

در این حالت به مانند قبل برای محاسبه $f(x,n)$ تنها کافیست یک بار $f(x,n/2)$ را محاسبه نموده و در خودش ضرب نماییم.

حالت دوم: n فرد باشد

در این حالت نیز حاصل ضرب به فرم $x.\underbrace{xxx...x}_{\lfloor \frac{n}{2} \rfloor}.\underbrace{xxx...x}_{\lfloor \frac{n}{2} \rfloor}$ در می آید پس تنها کافیست که $f(x,\lfloor n/2 \rfloor)$ را محاسبه نموده در خودش ضرب نموده و حاصل را در x ضرب نماییم که پس از محاسبه $f(x,\lfloor n/2 \rfloor)$ تنها دو عمل ضرب دیگر نیاز است.

برای پیچیدگی زمانی در بدترین حالت می توان گفت:

$$T(n)=T(n/2)+2=2(\log n)$$

لذا در پیچیدگی زمانی به صورت $O(\log n)$ خواهد بود.

تکه کد زیر به زبان $C++$ پیاده سازی تابع $f(x,n)$ می باشد:

```
int f(int x,int n)
}
  if(n==0)
    return 1;
  int p=f(x,n/2);
  if(n%2)
    return p*p*x;
  return p*p;
}
```

الگوریتم 1 : محاسبه x^n به صورت بازگشتی

حال از آنجایی که تنها خاصیت شرکت پذیری ایجاب می کند که بتوانیم از این روش تقسیم و غلبه استفاده کنیم پس این الگوریتم برای هر x به عنوان عضوی از یک حلقه جبری قابل اجرا است و جواب صحیح را برای ما محاسبه می کند که حلقه مورد نظر می تواند حلقه اعداد حقیقی، حلقه اعداد مختلط، حلقه اعداد صحیح، حلقه ماتریس ها بر روی یک میدان دلخواه، حلقه F^n که F میدان دلخواهی باشد، حلقه $F[x]$ که F میدان دلخواهی باشد و....

اما برای خیلی از حلقه ها روش بازگشتی به علت کپی کردن x در پیاده سازی ممکن است زیاد کار آمد نباشد و هر بار کپی کردن و فرستان x به تابع هنگام فراخوانی برای عمل بازگشت خود زمان بر و ناکارآمد باشد لذا در این حالت روش غیر بازگشتی می تواند بسیار کارآمد تر باشد.

برای روش غیر بازگشتی کافیست که n را در مبنای 2 نوشته و از انتها شروع کنیم. حاصل فعلی را اگر یک بیت 1 بود در خودش و x ضرب نموده و اگر یک بیت صفر بود تنها در خودش ضرب نماییم که در واقع این روش همان روش بازگشتی است حاصل را برعکس، می سازیم الگوریتم زیر که کدی به زبان $C++$ است روش ما را بهتر تشریح می کند:

```
int f(int x,int n)
{
  vector<int>bt;
  while(n)
  {
      bt.push_back(n%2);
      n/=2;
   }
  int ret=1;
  for(int i=bt.size()-1;i>=0;i--)
    if(bt[i]==0)
      ret*=ret;
    else
      ret*=ret*x;
  return ret;
}
```

الگوریتم 2: محاسبه x^n با روشی غیر بازگشتی

فصل پنجم: الگوریتم ضرب چندجمله ای ها (FFT)

در این فصل در مورد الگوریتم FFT بحث خواهیم کرد که بر مبنای تبدیلات فوریه می باشد که این تبدیلات تاثیرات زیادی بر زندگی ما داشته اند و کابرد اصلی آن ها در زمینه پردازش سیگنال های دیجیتال می باشد.

به عقیده خیلی از دانشمندان الگوریتم پیش رو یکی از ده الگوریتم بزرگ قرن 20 ام می باشد که در زیبایی و کاربرد در نوع خود کم نظیر است.

حال به تشریح این الگوریتم و نحوه کارآیی آن می پردازیم

فرض کنید دو چند جمله ای زیر داده شده اند:

$$p(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$$
$$q(x) = b_0 + b_1 x + \ldots + b_{n-1} x^{n-1}$$

حاصل ضرب آن ها به صورت زیر می باشد:

$$p(x).q(x) = c_0 + c_1 x + \ldots + c_{2n-2} x^{2n-2}$$

به طوری که

$$c_i = \sum_{\max\{0,i-(n-1)\} \le k \le \min\{i,n-1\}} a_k b_{i-k}$$

برای محاسبه حاصل حاصل ضرب حداکثر n^2 تا عمل ضرب نیاز می باشد پس می توان حاصل ضرب را به سادگی با الگوریتمی با پیچیدگی $O(n^2)$ محاسبه نمود. که در خیلی از مواقع چندان کارآمد نمی باشد.

اما می توان حاصل ضرب را بسیار کارآمد تر نیز محاسبه نمود همانطور که گفته شد این امر به وسیله تبدیلات فوریه میسر می شود یا به عبارت ساده تر همان الگوریتم FFT.

تبدیل فوریه گسسته (DFT)

قبل از هر چیز نیاز است با تبدیل فوریه گسسته آشنا شویم.

فرض کنید ω_n همان nامین ریشه واحد تعریف شده در انتهای فصل قبل باشد یعنی $\omega_n = e^{i\frac{2\pi}{n}}$ تبدیل فوریه گسسته نگاشتی است که بردار a را به $\hat{a}$ نگاشت می کند:

$$a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \to \hat{a} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

به طوری که:

$$\hat{a}_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \; j = 0, \ldots, n-1$$

همچنین می توان آن این نگاشت را به صورت عملگری با استفاده از یک ماتریس نشان داد:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}$$

که ماتریس فوق یک ماتریس واندرموند می باشد و آن را V_n در نظر می گیریم.

همچنین DFT چندجمله ای

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

را در n نقطه $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ محاسبه می نماید به عبارت دیگر:

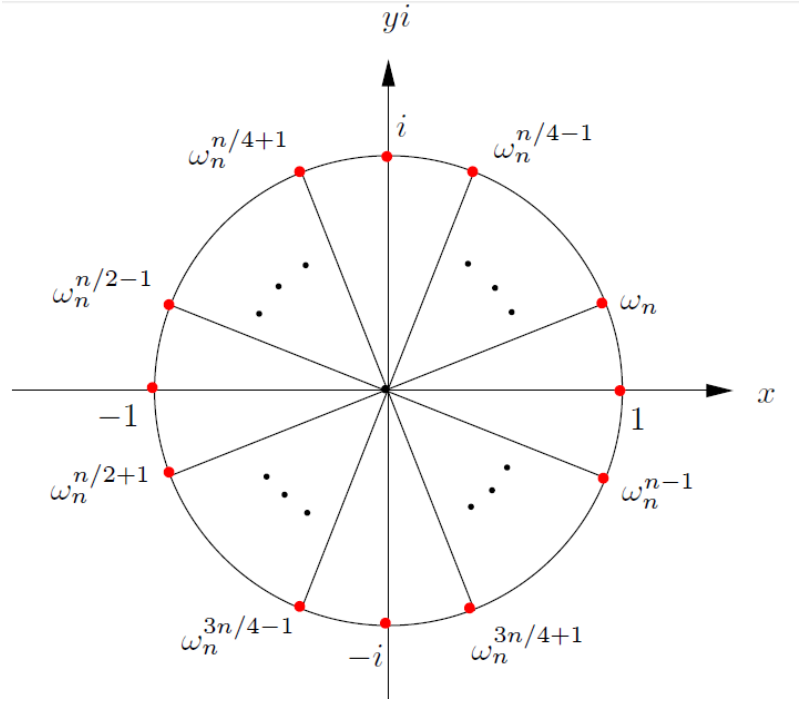
$$\hat{a}_k = p(\omega_n^k), \forall k \quad 0 \le k \le n-1$$

برای سادگی فرض می کنیم که n توانی از دو باشد.

برای حالت های دیگر به سادگی می توان جمله های با درجه بیشتر اما با ضریب صفر به چند جمله ایمان اضافه نماییم آن چنان که:

$$a_n = a_{n+1} = \cdots = a_{2^{\lceil \log_2 n \rceil}-1} = 0$$

توان های ω_n در صفحه مختلط به صورت زیر می باشند:



چند ویژگی زیر برای ادامه بحث و استفاده در الگوریتم FFT مورد نیاز هستند:

$$\omega_n^n = 1$$
$$\omega_n^{n+k} = \omega_n^k$$
$$\omega_n^{\frac{n}{2}} = -1 \qquad (**)$$
$$\omega_n^{\frac{n}{2}+k} = -\omega_n^k$$

حال با استفاده از تکنیک تقسیم و غلبه $p(x)$ را برای همه ریشه ها محاسبه می نماییم.

$p(x)$ را به دو چند جمله $p_0(x)$ و $p_1(x)$ تقسیم می نماییم و درجه هر کدام $\frac{n}{2}-1$ می باشد:

$$p_0(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{\frac{n}{2}-1}$$
$$p_1(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{\frac{n}{2}-1}$$

حال $p(x)$ را با استفاده از این دو چندجمله ای بازسازی می نماییم:

$$p(x) = p_0(x^2) + x p_1(x^2) \quad (*)$$

حال برای محاسبه $p(x)$ در ریشه های n ام مسئله به دو قسمت تقسیم می شود:

1- محاسبه $p_0(x)$ و $p_1(x)$ در $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$
2- ترکیب حاصل بر اساس (*)

توجه داشته باشید که $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$ شامل فقط $\frac{n}{2}$ ریشه مختلط واحد می باشند.

حال الگوریتم تقسیم و غلبه زیر $p(x)$ را برای همه ریشه ها طبق مطالبی که گفته شد محاسبه می نماید که پایه الگوریتم اصلی FFT می باشد:

```
RECURSIVE-DFT(a, n)
1   if n = 1
2       then return a
3   w_n ← e^{i 2π/n}
4   w ← 1
5   a^[0] ← (a_0, a_2, ..., a_{n-2})
6   a^[1] ← (a_1, a_3, ..., a_{n-1})
7   â^[0] ← RECURSIVE-DFT(a^[0], n/2)
8   â^[1] ← RECURSIVE-DFT(a^[1], n/2)
9   for k = 0 to n/2 − 1 do
10      â_k ← â_k^[0] + ω â_k^[1]
11      â_{k+n/2} ← â_k^[0] − ω â_k^[1]
12      ω ← ω w_n
13  return (â_0, â_1, ..., â_{n-1})
```

الگوریتم 3: سودوکد DFT

برای فهمیدن خط 11 سودوکد فوق لازم به ذکر است که در k امین مرحله از حلقه for در خط های 9-12 ، $\omega = \omega_n^k$ و داریم:

$$\begin{aligned}
\hat{a}_{k+\frac{n}{2}} &= \hat{a}_k^{[0]} - \omega_n^k \hat{a}_k^{[1]} \\
&= \hat{a}_k^{[0]} + \omega_n^{k+\frac{n}{2}} \hat{a}_k^{[1]} \\
&= p_0\left(\omega_n^{2k}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k}\right) \\
&= p_0\left(\omega_n^{2k+n}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k+n}\right) \\
&= p\left(\omega_n^{k+\frac{n}{2}}\right),
\end{aligned}$$

طبق (**)

برای محاسبه پیچیدگی الگوریتم فوق، فرض کنید $T(n)$ زمان اجرای الگوریتم فوق باشد خط های 1-6 زمان اجرای $\theta(n)$ دارند. خط های 7 و 8 هر کدام زمانی معادل $T(n/2)$ مصرف می کنند. 9-13 زمان $\theta(n)$ مصرف می کنند. بنابراین می توان گفت:

$$T(n)=2T(n/2)+\theta(n)$$

پس:

$$T(n)=\theta(n \log_2 n).$$

معکوس DFT

فرض کنید می خواهیم a را از روی $\hat{a}$ محاسبه نماییم از آنجایی که ماتریس واندرموند معرفی شده با n ریشه متمایز ساخته شده است همواره دترمینان ناصفر دارد. لذا وارون پذیر است پس:

$$a = V_n^{-1}\hat{a}$$

به عبارت دیگر قصد داریم ضرایب چند جمله ای $p(x)$ را از روی n نقطه $p(\omega_n^0), p(\omega_n^1), ..., p(\omega_n^{n-1})$ محاسبه نماییم که این محاسبه با زمان $\theta(n \log_2 n)$ امکان پذیر است. که در ادامه نشان خواهیم داد.

وارون ماتریس واندرموند V_n به صورت زیر است:

$$V_n^{-1} = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{pmatrix}$$

در این جا برای محاسبه معکوس از اثبات ها و محاسبات حجیم پرهیز کردیم اما مبنای محاسبه ماتریس معکوس فوق مطالب ذکر شده در فصل سوم می باشد.

بر اساس مطالب بخش قبل همچنان می توان الگوریتم 1 را اجرا کرد با این تفاوت که به جای a قرار می دهیم $\hat{a}$ و بلعکس همچنین به جای ω_n قرار می دهیم ω_n^{-1} (همان ω_n^{n-1}است) و نتیجه را در $1/n$ ضرب می نماییم.

ضرب سریع دو چند جمله ای

با مطالب ذکر شده در مورد DFT آماده شده ایم که به الگوریتم اصلی FFT بپردازیم.

همان طور که در ابتدای فصل اشاره شد هدف این است که دو چند جمله ای را با پیچیدگی زمانی کارآمدی در یکدیگر ضرب نماییم:

$$p(x) = a_0 + a_1 x + ... + a_{n-1} x^{n-1}$$
$$q(x) = b_0 + b_1 x + ... + b_{n-1} x^{n-1}$$

که حاصل ضربشان به صورت زیر می باشد:

$$p(x).q(x) = c_0 + c_1 x + ... + c_{2n-2} x^{2n-2}$$

الگوریتم زیر با $\theta(n \log n)$ حاصل ضرب دو چند جمله ای فوق را محاسبه می نماید:

1- $p(x)$ و $q(x)$ را در $2n$ نقطه $\omega_{2n}^0, \omega_{2n}^1, \omega_{2n}^2, ..., \omega_{2n}^{2n-1}$ با استفاده از DFT محاسبه نمایید.
2- مقدار $p(x).q(x)$ را در $2n$ نقطه $\omega_{2n}^0, \omega_{2n}^1, \omega_{2n}^2, ..., \omega_{2n}^{2n-1}$ به دست آورید.
3- به وسیله معکوس DFT ضرایب $c_0, c_1, c_2, \ldots, c_{2n-2}$ از چندجمله ای $p.q$ را محاسبه نمایید.

الگوریتم 2 : FFT

لازم به ذکر است که:

خط اول زمان اجرای $\theta(n \log n)$ دارد.

خط دوم زمان اجرای $\theta(n)$ دارد. دلیل محاسبه $p(\omega).q(\omega)$ در $2n$ ریشه را می توان به صورت زیر شرح داد:

$$
\begin{aligned}
(p.q)(\omega_{2n}^{0}) &= p(\omega_{2n}^{0}).q(\omega_{2n}^{0}) \\
(p.q)(\omega_{2n}^{1}) &= p(\omega_{2n}^{1}).q(\omega_{2n}^{1}) \\
(p.q)(\omega_{2n}^{2}) &= p(\omega_{2n}^{2}).q(\omega_{2n}^{2}) \\
&\vdots \\
(p.q)(\omega_{2n}^{2n-1}) &= p(\omega_{2n}^{2n-1}).q(\omega_{2n}^{2n-1})
\end{aligned}
$$

در خط سوم مقدار $p.q$ هر ریشه را در اختیار داریم پس با استفاده از معکوس DFT طبق آنچه در بخش قبل گفته شد می توان ضرایب $p.q$ را با زمان اجرای $\theta(n \log n)$ محاسبه نمود.

پس اکنون قادریم که حاصل ضرب دو چند جمله ای را در زمان اجرای $\theta(n \log n)$ محاسبه نماییم.

پیاده سازی

کد زیر پیاده سازی الگوریتم FFT به زبان $C++$ می باشد:

```
#include<bits/stdc++.h>
using namespace std;

typedef vector<int> VI;
double PI = acos(0) * 2;

class complex
{
public:
  double a, b;
  complex() {a = 0.0; b = 0.0;}
  complex(double na, double nb) {a = na; b = nb;}
  const complex operator+(const complex &c) const
  {return complex(a + c.a, b + c.b);}
  const complex operator-(const complex &c) const
  {return complex(a - c.a, b - c.b);}
  const complex operator*(const complex &c) const
  {return complex(a*c.a - b*c.b, a*c.b + b*c.a);}
  double magnitude() {return sqrt(a*a+b*b);}
  void print() {printf("(%.3f %.3f)\n", a, b);}
};

class FFT
{
public:
  vector<complex> data;
  vector<complex> roots;
  VI rev;
  int s, n;

  void setSize(int ns)
  {
    s = ns;
    n = (1 << s);
    int i, j;
    rev = VI(n);
    data = vector<complex> (n);
    roots = vector<complex> (n+1);
    for (i = 0; i < n; i++)
      for (j = 0; j < s; j++)
     if ((i & (1 << j)) != 0)
       rev[i] += (1 << (s-j-1));
    roots[0] = complex(1, 0);
    complex mult = complex(cos(2*PI/n), sin(2*PI/n));
    for (i = 1; i <= n; i++)
      roots[i] = roots[i-1] * mult;
  }
```

```
  void bitReverse(vector<complex> &array)
  {
    vector<complex> temp(n);
    int i;
    for (i = 0; i < n; i++)
      temp[i] = array[rev[i]];
    for (i = 0; i < n; i++)
      array[i] = temp[i];
  }

  void transform(bool inverse = false)
  {
    bitReverse(data);
    int i, j, k;
    for (i = 1; i <= s; i++) {
      int m = (1 << i), md2 = m / 2;
      int start = 0, increment = (1 << (s-i));
      if (inverse) {
     start = n;
     increment *= -1;
      }
      complex t, u;
      for (k = 0; k < n; k += m) {
     int index = start;
     for (j = k; j < md2+k; j++) {
       t = roots[index] * data[j+md2];
       index += increment;
       data[j+md2] = data[j] - t;
       data[j] = data[j] + t;
     }
      }
    }
    if (inverse)
      for (i = 0; i < n; i++) {
     data[i].a /= n;
     data[i].b /= n;
      }
  }

  static VI convolution(VI &a, VI &b)
  {
    int alen = a.size(), blen = b.size();
    int resn = alen + blen - 1;    // size of the resulting
array
    int s = 0, i;
    while ((1 << s) < resn) s++;   // n = 2^s
    int n = 1 << s; // round up the the nearest power of two

    FFT pga, pgb;
    pga.setSize(s); // fill and transform first array
```

```
    for (i = 0; i < alen; i++) pga.data[i] = complex(a[i],
0);
    for (i = alen; i < n; i++)     pga.data[i] = complex(0,
0);
    pga.transform();

    pgb.setSize(s); // fill and transform second array
    for (i = 0; i < blen; i++)     pgb.data[i] =
complex(b[i], 0);
    for (i = blen; i < n; i++)     pgb.data[i] = complex(0,
0);
    pgb.transform();

    for (i = 0; i < n; i++)   pga.data[i] = pga.data[i] *
pgb.data[i];
    pga.transform(true); // inverse transform
    VI result = VI (resn);    // round to nearest integer
    for (i = 0; i < resn; i++)     result[i] = (int)
(pga.data[i].a + 0.5);

    int actualSize = resn - 1;     // find proper size of
array
    while (result[actualSize] == 0)
      actualSize--;
    if (actualSize < 0) actualSize = 0;
    result.resize(actualSize+1);
    return result;
  }
};
int main()
{
  VI a = VI (10);
  for (int i = 0; i < 10; i++)
    a[i] = (i+1)*(i+1);
  VI b = FFT::convolution(a, a);
  /* 1 8 34 104 259 560 1092 1968 3333
     5368 8052 11120 14259 17104 19234 20168 19361 16200
10000*/
  for (int i = 0; i < b.size(); i++)
    printf("%d ", b[i]);
  return 0;
}
```

الگوریتم 4: پیاده سازی FFT ، آرایه a که شامل ضرایب یک چند جمله ای است به توان دو می رسد.

معرفی NTT

برای مطلب پایانی الگوریتم NTT را نیز معرفی می نماییم.

این الگوریتم شباهت زیادی به الگوریتم FTT دارد و ریشه های یکسانی دارند با این تفاوت که NTT به جای استفاده از اعداد و صفحه مختلط از حلقه های جبری $\mathbb{Z}_n$ و قضایای فرما استفاده می نماید و حاصل ضرب دو چند جمله ای را محاسبه می نماید.

همچنین پیچیدگی زمانی یکسانی با الگوریتم FFT دارد.

فصل ششم: مسئله دزد حریص

صورت مسئله

دزدی به یک مغازه برای دزدی رفته است و یک کوله پشتی به همراه دارد که دقیقا به اندازه k شی ظرفیت دارد، مغازه دقیقا n شی دارد که هر کدام ارزش های متفاوتی دارند فرض کنید شی i ام ارزش a_i دارد همچنین از هر شی به تعداد نامتناهی موجود است. دزد بسیار حریص است و کوله پشتی را حتما پر می کند یعنی دقیقا k شی برمیدارد(نه لزوما متمایز) . سود دزد برابر است با مجموع ارزش اشیاء داخل کوله پشتی. تمام مقادیر ممکن برای سود حاصل از دزدی را بیابید به عبارت دیگر تمام سود های ممکنی که دزد برای هر نوع انتخاب k شی خود به دست می آورد را بیابید. فرض کنید حداکثر مقدار a_i ها برابر با m باشد.

در ادامه در مورد راه حل های این مسئله بحث خواهیم کرد.

راه حل بدیهی

حداکثر سودی که دزد می تواند کسب کند برابر است با mk پس تمام سود های ممکن اعدادی متمایز در بازی $[0,mk]$ می باشند. یک راه حل بدیهی این است که روی k استقرا بزنیم ، فرض کنید که جواب را برای کوله پشتی با ظرفیت p بدانیم که به صورت یک مجموعه حداکثر mk عضوی است برای به دست آوردن جواب برای $p+1$ کافیست شی $p+1$ ام را یکی از n شی به دلخواه بگذاریم و سپس برای هر شی هر عضو از مجموعه در مرحله pام را با ارزش آن شی جمع نموده در در مجموعه جدید برای کوله پشتی $p+1$ ظرفیتی قرار دهیم به این شکل جواب برای $p+1$ با زمان nmk از روی حالت p ظرفیتی به دست می آید.

پس برای محاسبه مسئله برای کوله پشتی k عضوی زمان $O(nmk^2)$ لازم است.

اما همان طور که مشاهده می کنید این راه حل چندان کارآمد نمی باشد. ایده ما برای حل این مسئله با زمانی کارآمد تر استفاده از مباحث دو فصل قبلی و ترکیب آن ها می باشد که مسئله را بهینه تر می کند.

راه حل با استفاده از FFT

چند جمله ای $p(x)=ax^3+bx^4+cx^6$ را در نظر بگیرید.

فرض کنید می خواهیم چندجمله ای $(p(x))^3$ را محاسبه کنیم :

$$(ax^3+bx^4+cx^6)(ax^3+bx^4+cx^6)(ax^3+bx^4+cx^6)=$$
$$(a.a.a)x^{(3+3+3)}+(3.a.a.b)x^{(3+3+4)}+\cdots+(6.a.b.c)x^{(3+4+6)}+\cdots+(c.c.c)x^{(6+6+6)}$$

همان طور که مشاهده می کنید برای محاسبه جملات چندجمله ای حاصل باید از هر یک از چند جمله های های $p(x)$ در ضرب $\underbrace{(ax^3+bx^4+cx^6)}_{p(x)}\underbrace{(ax^3+bx^4+cx^6)}_{p(x)}\underbrace{(ax^3+bx^4+cx^6)}_{p(x)}$ دقیقا یک جمله را انتخاب کنیم و سپس سه جمله را در هم ضرب نموده که باعث می شود توان های این سه جمله با یکدیگر جمع شده و توان یکی از جملات در حاصل ضرب را ایجاد نمایند همچنین ضرایب این جملات نیز در هم ضرب می شوند. اگر هم چندین بار جمله هایی با توان های یکسان ساخته شد ضریب حاصل ضرب آن ها با هم جمع می شود و در نهایت یک جمله در $(p(x))^3$ ایجاد نمایند.

به واقع باید تمام انتخاب های سه تایی متمایز ممکن از جمله های هر یک از سه $p(x)$ را در نظر گرفته جملات را در هم ضرب نموده و به صورت αx^{β} در آورده و همه را با هم جمع نماییم.

حال اگر مسئله دزد حریص را با چند جمله ای $p(x)$ به فرم $p(x)=x^{a_1}+x^{a_2}+\cdots+x^{a_n}$ تبدیل نماییم و به توان k برسانیم چند جمله ای $(p(x))^k$ حاصل می شود که دارای جملاتی می باشد که توانشان جمع k تا از a_i ها (نه لزوما متمایز) می باشند. همچنین مجموع تمام انتخاب k تایی از a_i ها را هم به صورت توانی از یک جمله در $(p(x))^k$ ظاهر شده می یابیم.

پس تنها کافیست چند جمله ای $p(x)$ را تشکیل داده و ضریب جملات x^{a_i} را برابر یک قرار داده و ضریب سایر جملات را برابر صفر قرار دهیم درجه $p(x)$ نیز برابر با m خواهد بود.

جواب نیز توان تمام جملاتی می باشد که در چند جمله ای $(p(x))^k$ ظاهر شده اند و ضریبی ناصفر دارند.

حال کافیست $(p(x))^k$ را با اجرای k-1 بار الگوریتم FFT به دست آوریم.

حداکثر درجه حاصل ضرب mk خواهد بود پس هر بار اجرای FFT از زمان $O(mk\ log(mk))$ خواهد بود که k-1 بار اجرای آن لازم است پس زمان کل محاسبه حاصل ضرب از $O(mk^2\ log(mk))$ خواهد بود همان طور که مشاهد می کنید این راه حل از راه حل بدیهی قسمت قبل کارآمد تر می باشد.

راه حل با استفاده از FFT و ترکیب آن با توان رسانی لگاریتمی در حلقه های جبری

می دانیم که چندجمله ای $p(x)$ ایجاد شده در قسمت قبل عضوی از حلقه چندجمله ای ها روی میدان اعداد صحیح می باشد. لذا می توان $p(x)$ را به عنوان عضوی از یک حلقه جبری در نظر گرفت پس می توان $(p(x))^k$ را با زمان لگاریتمی محاسبه کرد در واقع به جای k-1 بار اجرای الگوریتم FFT که در قسمت قبل مشاهده کردیم تنها $\log k$ بار اجرا کافی است. لذا زمان محاسبه $(p(x))^k$ از $O(mk\ log(k)\ log(mk))$ خواهد بود که راه حل مسئله را بسیار کارآمد می کند.

در ادامه یک پیاده سازی برای این راه حل ارائه خواهیم نمود.

```
#include<iostream>
#include<cstdio>
#include<vector>
#include<cmath>
using namespace std;

typedef vector<int> VI;
double PI = acos(0) * 2;

class complex
{
public:
  double a, b;
  complex() {a = 0.0; b = 0.0;}
  complex(double na, double nb) {a = na; b = nb;}
  const complex operator+(const complex &c) const
  {return complex(a + c.a, b + c.b);}
  const complex operator-(const complex &c) const
  {return complex(a - c.a, b - c.b);}
  const complex operator*(const complex &c) const
  {return complex(a*c.a - b*c.b, a*c.b + b*c.a);}
  double magnitude() {return sqrt(a*a+b*b);}
  void print() {printf("(%.3f %.3f)\n", a, b);}
};

class FFT
{
```

```
public:
  vector<complex> data;
  vector<complex> roots;
  VI rev;
  int s, n;

  void setSize(int ns)
  {
    s = ns;
    n = (1 << s);
    int i, j;
    rev = VI(n);
    data = vector<complex> (n);
    roots = vector<complex> (n+1);
    for (i = 0; i < n; i++)
      for (j = 0; j < s; j++)
     if ((i & (1 << j)) != 0)
       rev[i] += (1 << (s-j-1));
    roots[0] = complex(1, 0);
    complex mult = complex(cos(2*PI/n), sin(2*PI/n));
    for (i = 1; i <= n; i++)
      roots[i] = roots[i-1] * mult;
  }

  void bitReverse(vector<complex> &array)
  {
    vector<complex> temp(n);
```

```
  int i;
  for (i = 0; i < n; i++)
    temp[i] = array[rev[i]];
  for (i = 0; i < n; i++)
    array[i] = temp[i];
}

void transform(bool inverse = false)
{
  bitReverse(data);
  int i, j, k;
  for (i = 1; i <= s; i++) {
    int m = (1 << i), md2 = m / 2;
    int start = 0, increment = (1 << (s-i));
    if (inverse) {
   start = n;
   increment *= -1;
    }
    complex t, u;
    for (k = 0; k < n; k += m) {
   int index = start;
   for (j = k; j < md2+k; j++) {
     t = roots[index] * data[j+md2];
     index += increment;
     data[j+md2] = data[j] - t;
     data[j] = data[j] + t;
   }
```

```
      }
    }
    if (inverse)
      for (i = 0; i < n; i++) {
     data[i].a /= n;
     data[i].b /= n;
      }
  }


  static VI convolution(VI &a, VI &b)
  {
    int alen = a.size(), blen = b.size();
    int resn = alen + blen - 1;    // size of the resulting
array
    int s = 0, i;
    while ((1 << s) < resn) s++;   // n = 2^s
    int n = 1 << s; // round up the the nearest power of two
    FFT pga, pgb;
    pga.setSize(s); // fill and transform first array
    for (i = 0; i < alen; i++) pga.data[i] = complex(a[i],
0);
    for (i = alen; i < n; i++)     pga.data[i] = complex(0,
0);
    pga.transform();


    pgb.setSize(s); // fill and transform second array
    for (i = 0; i < blen; i++)     pgb.data[i] =
complex(b[i], 0);
```

```
    for (i = blen; i < n; i++)     pgb.data[i] = complex(0,
0);

    pgb.transform();

    for (i = 0; i < n; i++)   pga.data[i] = pga.data[i] *
pgb.data[i];

    pga.transform(true); // inverse transform

    VI result = VI (resn);    // round to nearest integer

    for (i = 0; i < resn; i++)     result[i] = (int)
(pga.data[i].a + 0.5);

    int actualSize = resn - 1;     // find proper size of
array

    while (result[actualSize] == 0)

      actualSize--;

    if (actualSize < 0) actualSize = 0;

    result.resize(actualSize+1);

    return result;

  }

};

vector<int> f,mul,bt;

int n,k;

int main()

{

  cin>>n>>k;

  for(int i=0;i<n;i++)

    {

      int x;

      scanf("%d",&x);

      for(int j=f.size();j<=x;j++)
```

```
      f.push_back(0);
       f[x]=1;
     }
  mul.push_back(1);
  while(k)
     {
       bt.push_back(k&1);
       k>>=1;
     }
  for(int i=bt.size()-1;i>=0;i--)
     {
       // cout<<bt[i]<<" ";
       mul=FFT::convolution(mul,mul);
       if(bt[i])
      mul=FFT::convolution(mul,f);
       for(int i=0;i<mul.size();i++)
      if(mul[i])
        mul[i]=1;
     }
  for(int i=0;i<mul.size();i++)
     if(mul[i])
       printf("%d ",i);
  return 0;
}
```

الگوریتم 5: پیاده سازی راه حل بهینه مسئله دزد حریص با استفاده از روش های توان رسانی لگاریتمی در حلقه های جبری و الگوریتم FFT

فصل هفتم: مسئله ترتیب آماری

آرایه a به طول n از اعداد صحیح و عدد x داده شده است. برای عدد k یک بازه از آرایه دارای ترتیب آماری k می باشد اگر دقیقا k عدد از این بازه کمتر از x باشند.

برای هر k بین 1 تا n تعداد بازه های با ترتیب آماری k را بیابید.

برای مثال اگر آرایه a=[1,2,3,4,5] باشد و x=3 باشد جواب به صورت {0 0 0 4 5} می باشد.(5 بازه با ترتیب آماری 1 وجود دارند که بازه های [2,3,4,5],[2,3,4],[2,3],[2],[1] هستند.

راه حل

آرایه s را به این صورت تعریف می کنیم که $s[i]$ برابر باشد با تعداد عناصر کمتر از x در بازه ی [1,i] ، آرایه s یک آرایه صعودی خواهد بود.

آرایه r را به این صورت تشکیل می دهیم که $r[i]$ برابر است با تعداد i های ظاهر شده در آرایه s.

جواب مسئله برای هر k در بازه ی 1 تا n را $ans[k]$ تعریف می کنیم که برابر است با:

$$\sum_{i,j,i-j=k} r[i].r[j]$$

آرایه v را به این صورت می سازیم که :

$$v[i]=r[n-i]$$

r و v را به صورت دو چند جمله ای در نظر گرفته و در یکدیگر ضرب می نماییم تا آرایه p حاصل شود یعنی:

$$p=r\times v$$

بنابراین

$$p[n+k]=\sum_{i,h,i+h=n+k} r[i].v[h]=$$
$$\sum_{i,j,i+n-j=n+k} r[i].r[j]=$$
$$\sum_{i,j,i-j=k} r[i].r[j]=ans[k]$$

تنها کافیست p را با استفاده از FFT محاسبه نماییم.

پس زمان اجرا برابر است با $O(n\ log(n))$.

فصل هشتم: معرفی چند مسئله دیگر

در این فصل به معرفی چند مسئله می پردازیم که بر خلاف ظاهرشان ، یافتن الگوریتم بهینه در نهایت به روش FFT منجر می شود که این خود از زیبایی های این الگوریتم می باشد.

مسئله درخت دودویی میوه دار

یک درخت دودویی T در اختیار داریم یک زیر درخت ریشه دار از T زیر درختی از T است که همبند باشد و شامل ریشه T نیز باشد. راس ها از 1 تا n شماره گذاری شده اند. و ریشه همواره راس 1 می باشد.

یک زیر درخت ریشه دار T دارای میوه است به طوری که:

- میوه ها فقط در راس ها هستند.
- ریشه دقیقا x میوه دارد.
- میوه های هر راس از مجموع میوه های فرزندانش کمتر نیست

تعداد زیردرخت های ریشه دار و میوه دار متمایز از درخت T چند تا می باشد؟

دو زیر درخت ریشه دار و میوه دار متمایزند اگر زیردرخت های برچسب دارشان یکسان نباشد و در صورتی که زیردرختان ریشه دارشان یکسان باشد ، حداقل یک راس یافت شود که تعداد میوه های متفاوتی در دو درخت داشته باشد.

مسئله فاصله بین رشته ای

دو رشته r و s با طول یکسان را در نظر بگیرد فاصله بین آن ها برابر است با حداقل عمل جایگزینی مورد نیاز برای این که دو رشته با یکدیگر یکسان شوند.

عمل جایگزینی: دو کاراکتر c_1 و c_2 به دلخواه انتخاب می شوند و در هر دو رشته تمام کاراکتر های c_2 با کاراکتر c_1 جایگزین می شوند.

رشته های S و T داده شده اند.

برای تمام زیر رشته های S به طول $|T|$ فاصله بین رشته ای آن ها و رشته T را محاسبه کنید.

مثال: فرض کنید $S=''abcdefa''$ و $T=''ddcb''$ دو رشته باشند آنگاه زیر رشته های هم اندازه T از S از راست به چپ به ترتیب جواب های زیر را دارند:

2 3 3 3

مسئله عدد روی تخته سیاه

عدد x روی تخته سیاه نوشته شده است هر مرحله قدم های زیر را به ترتیب انجام می دهیم:

- عدد x را پاک می کنیم
- عدد رندمی از بازه ی $[0,x]$ انتخاب می کنیم و به جای x روی تخته می نویسیم

بعد از m مرحله احتمال وجود هر یک از اعداد بر روی تخته سیاه در بازه $[0,n]$ را محاسبه کنید .

در ابتدا $n+1$ عدد $P_0, P_1, ..., P_n$ به شما داده می شود که P_i احتمال این است که اولین عدد بر روی تخته سیاه i باشد.

فصل نهم: نتایج

این پروژه حاصل تحقیق جامعی در مورد الگوریتم FFT بود.

همانطور که در فصل های قبل بحث شد این الگوریتم کاربرد های فراوانی دارد و می تواند مسائل متنوعی را حل کند.

چندجمله ای ها به مانند ماتریس ها توابعی بسیار انعطاف پذیر می باشند و از قدرت بالایی برای طراحی توابع متنوع برخوردارند از همین رو همان طور که در فصل قبل مشاهده شد مسائلی به وسیله الگوریتم FFT قابل تحلیل می باشند که ظاهر کاملا متفاوتی دارند.

در فصل هشتم مسائلی مطرح شد که ظاهر آنها یک مسئله ترکیبیاتی (مسئله درخت دودویی میوه دار) یا یک مسئله پردازش رشته (مسئله فاصله بین رشته ای) یا یک مسئله احتمالاتی (مسئله عدد روی تخته سیاه) بود در حالی که راه حل بهینه آنها با طراحی یک چندجمله ای و استفاده از الگوریتم FFT برای ضرب چند جمله ای ها در زمان بهینه میسر می شود.

همچنین با ایده ای که در فصل ششم مطرح کردیم ، توان رسانی چند جمله ای ها نیز در زمانی بهینه میسر شد.

تمام این ابزارهای معرفی شده می توانند کاربرد وسیعی در مسائل دنیای علوم کامپیوتر داشته باشند که سعی شد بخش هایی از این کاربردها را در فصل های گذشته مطرح کنیم.

همچنین باری دیگر اهمیت ریاضیات محض در دنیای علوم کامپیوتر به تصویر کشیده شد و این بار در شاخه به ظاهر نه چندان مرتبط آنالیز مختلط، همچنین جبرخطی و جبرمجرد نیز نقش مهمی در رسیدن به الگورتیم FFT و کاربرد های آن ایفا کردند.

«بی شک این الگوریتم یکی از زیباترین الگوریتم های تاریخ علم علوم کامپیوتر می باشد»

فصل دهم: منابع

1- James Ward Brown, Ruel V.Churchill, Complex Variables and Applications, McGraw Hill, 8th edition.

2- جین هو کواک، سانگ پیو هونگ، جبرخطی، مرکز نشر دانشگاهی، چاپ اول 1393 ، ترجمه محمدرضا درفشه، نگار شهنی کرمزاده

3- https://proofwiki.org/wiki/Inverse_of_Vandermonde_Matrix

4- توماس کورمن، چارلز لیزرسون، رونالد ریوست، کلیفورد استین، مقدمه ای بر الگوریتم ها، نشر «نص» ، ویراست سوم، چاپ دوم، ترجمه مهندس دهقان طرزه و دکتر یحیی تابش

5- Lowa state University website, https://www.cs.iastate.edu/