نظریه علوم کامپیوتر

نیم‌سال دوم ۱۳۹۹-۱۴۰۰

مدرس: دکتر جواد ابراهیمی، دستیار آموزشی: ساجد کریمی

دانشکده علوم ریاضی

تمرین سری اول

نام و نام‌خانوادگی: امید یعقوبی

شماره دانشجویی: -

پرسش ۱

هر الگوریتم جویباری که به صورت قطعی F_0 یا همان Number Of Distinct Elements را حساب کند حداقل به $\Omega(nl)$ فضا احتیاج دارد.
برای اثبات فرض می کنیم : $|\Sigma| \geq n^2$

می گوییم دو Stream با نامهای x و y جداشدنی (از n) $''distinguishable''$ [۱، ص. ۳] برای مساله ی DE هستند اگر وجود داشته باشد Stream با نام z به شکلی که نتیجه ی DE برای $x.z$ متفاوت باشد با $y.z$ یا به عبارتی تعداد عناصر متمایز $x.z$ متفاوت باشد با تعداد عناصر متمایز $y.z$ و اندازه ی $x.z$ برابر با $y.z$ و برابر با n باشد. همچنین $streaming\ distinguisher$ [۱، ص. ۳] آن را با $D_{n,\Sigma}$ نشان می دهیم.
هر مجموعه به شکل $S = \{s_1, s_2, ..., s_{\frac{n}{2}}\}$ با اندازه ی $\frac{n}{2}$ که در آن $S \subseteq \Sigma$ یک دنباله به شکل $X_S = s_1, s_2, ..., s_{\frac{n}{2}}$ می سازد. اگر نتیجه ی مساله DE برای ورودی دلخواه x را به به شکل $DE(x)$ بنویسیم. واضح است که $DE(X_S) = \frac{n}{2}$.
تمام زیرمجموعه های $\frac{n}{2}$ عضوی از Σ، دنباله هایی متمایز به شکل بالا را می سازند. حال ادعا می کنیم که رشته های متناظر با این دنباله ها یک $Streaming\ Distinguisher$ برای مساله DE با ورودیهای به اندازه n هستند. یا به عبارتی این زیرمجموعه ها همان $D_{n,\Sigma}$ را تشکیل می دهند.
برای اثبات، دو دنباله دلخواه X_S و X_T را در نظر بگیرید. می گوییم z با مقدار X_S برای این دو وجود دارد به شکلی که $DE(x.z) \neq DE(y.z)$ چراکه $DE(X_S.X_S) = \frac{n}{2}$ ولی $DE(X_T.X_S) > \frac{n}{2}$. چون با فرض $S \neq T$ می دانیم $\exists s_i : s_i \in S \wedge s_i \notin T$ ، اگر این حکم برقرار نباشد یعنی همه ی اعضای S در T هستند و همچنین اندازه ی هر دو $\frac{n}{2}$ است که یعنی $S = T$ که با فرض متمایز بودن این دو در تناقض است. حال از آنجا که هیچ عضو جدیدی به X_S اضافه نشده تعداد اعضای متمایز آن کماکان $\frac{n}{2}$ باقی می ماند در صورتی که حداقل یک عضو به X_T اضافه شده است که باعث می شود تعداد عناصر متمایز $X_T.X_S$ اکیدا بیشتر از تعداد عناصر متمایز $X_S.X_S$ باشد.
چون S و T دلخواه بودند پس این حکم برای هر دو زیرمجموعه انتخوابی برقرار است. در نتیجه $D_{n,\Sigma}$ همان رشته های متناظر با دنباله های ساخته شده توسط زیرمجموعه های $\frac{n}{2}$ عضوی از Σ است. تعداد این زیرمجموعه ها نیز همان $\binom{|\Sigma|}{\frac{n}{2}}$ می باشد.
توجه شود که 2^l که متون اصلی با آن کار می کنند و l را $log_2|\Sigma|$ می گیرند همان $|\Sigma|$ است چون:

$$2^l = 2^{log_2|\Sigma|} = |\Sigma|^{log_2 2} = |\Sigma|^1 = |\Sigma|$$

با دانستن این که $\binom{n}{k} \geq (\frac{n}{k})^k$ و این فرض که $|\Sigma| \geq n^2$ داریم :

$$\binom{|\Sigma|}{\frac{n}{2}} \geq (\frac{2.|\Sigma|}{n})^{\frac{n}{2}} \geq |\Sigma|^n$$

پس $|D_{\Sigma,n}| \geq |\Sigma|^n$ در نتیجه $|\Sigma|^n$ رشته ی جدایی پذیر $(distinguishable\ string)$ [۱] برای مساله DE با ورودیهای با اندازه ی n وجود دارد. براساس قضیه شماره ی ۴ [۱، ص. ۴ قضیه ۴] هر الگوریتم جویباری برای حل مساله DE به حداقل $log(|\Sigma|^n)$ حافظه احتیاج دارد. از آنجا که $log(x^y) = y.log(x)$ پس هر الگوریتمی که این مساله را حل می کند به حداقل $n.log(|\Sigma|)$ فضا احتیاج دارد. چون l را $log(|\Sigma|)$ گرفتیم پس هر الگوریتم حل قطعی این مساله به حافظه ی $\Omega(nl)$ نیاز دارد.
■

می توان دو الگوریتمی داد که این مساله را با $O(nl)$ حل کند.

الگوریتم اول:

از آنجا که هر عضو از Σ با $log(|\Sigma|)$ بیت قابل نمایش است. پس کافیست تمام ورودی را به شرطی که پیشتر ذخیره نشده باشد ذخیره کنیم. حال برای عنصر i بررسی می کنیم که آیا در $i-1$ عنصر قبلی آمده است یا خیر. از آنجا که در F_0 تنها نگهداری رخداد $a_i \in \Sigma$ کافیست. در بدترین حالت تمامی اعضای Σ در Stream لیست شده اند. یعنی استفاده از حافظه در بدترین حالت برابر $n.log(|\Sigma|)$ خواهد بود که همان $O(nl)$ است. پس از اتمام ورودی. از روی طول حافظه ای که تا به حال آن را ذخیره کردیم می توانیم F_0 را حساب کنیم. اگر $M' = Total\ Memory - Free\ Memory$ آن گاه $F_0 = \frac{M'}{log(|\Sigma|)}$. دلیل آن هم واضح است زمانی که در بدترین حالت همه ی حافظه استفاده می شود. یعنی n عنصر متمایز داشتیم و از nl حافظه استفاده کردیم حال $n = \frac{nl}{l}$ که $l = log(|\Sigma|)$ است.

خود M' هم معمولا به راحتی قابل محاسبه است. برای مثال در برخی پیاده سازی ها اشاره گری به اولین خانه ی خالی یا آخرین خانه ی پر داریم. اگر اشاره گر به آخرین خانه ی پر داشته باشیم که نام آن p باشد و حافظه از 0 شروع شده باشد به راحتی $M' = p + 1$ می باشد. البته این ریزه کاریهای پیاده سازی خارج از محیط تمرین است. تنها اشاره ای به آن شد که به دست آوردن M' در $Stopping\ Rule$ کار پیچیده ای نیست. حتا در مدلهای انتزاعی تر اگر بتوان بین خانه ای که روی آن چیزی نوشته شده و $blank$ تفاوت گذاشت این کار به سادگی بدون افزایش پیچدگی فضایی به صورت مجانبی ممکن است. ولی برای نداشتن هرگونه ابهام روش کلاسیکتر به عنوان روش دوم گفته می شود.

الگوریتم دوم:

در این الگوریتم با همان ایده ی الگوریتم قبلی، هر عضو از Σ با $log(|\Sigma|)$ بیت قابل نمایش است. در اینجا علاوه بر نگهداری عناصر دیده شده متمایز، یک شمارنده نیز نگه میداریم. در نهایت جواب F_0 همان عددی است که شمارنده آن را نشان می دهد. با دیدن عنصر i بررسی می شود که در $i-1$ عنصر قبلی آمده است یا نه. اگر نیامده بود: ابتدا آن را در حافظه ذخیره می کنیم سپس شمارنده را یک واحد افزایشی می دهیم. در بدترین حالت حافظه مورد نیاز برای ما، حافظه نگهداری n عنصر با اندازه $log(|\Sigma|)$ است بعلاوه ی یک شمارنده که می تواند حداکثر از $log(n)$ حافظه استفاده کند. پس پیچیدگی فضایی در این الگوریتم برابر $O(n.log(|\Sigma|) + log(n))$ است. از آنجا که $|\Sigma| \geq n^2$ پس همیشه $n.log(|\Sigma|) \geq log(n)$ و در نتیجه پیچییدگی این الگوریتم هم از مرتبه $O(n.log(|\Sigma|))$ یعنی همان $O(nl)$ است.

پرسش ۲

آ)

به صورت شهودی می توان حس کرد که در یک هشِ یکنواخت، هرچه تعداد المان های متمایز بیشتر می شود، $Minimum$ بیشتر به سمت کوچک شدن حرکت می کند. از آنجا که فضای مپ شده ی K المان به صورت یکنواخت در بازه ی $[0, 1]$ پخش شده است، پس مقدار $Minimum$ به صورت شهودی برابر $\frac{1}{K+1}$ می باشد. [۲]

حال آن را به دو روش اثبات می کنیم (هر دو روش از منبع [۲] می باشد).

روش اول:

چون h یکنواخت است داریم:

$$\mathbb{E}[Y] = \int_0^1 P[Y \geq z]dz = \int_0^1 (1-z)^K dz = \left(-\frac{(1-z)^{K+1}}{K+1}\right)\Big|_0^1 = \frac{1}{K+1}$$

روش دوم:

این گونه استدلال می کنیم که $\mathbb{E}[Y]$ برابر است با احتمال اینکه هشِ $K+1$ امین المان برابر با $Minimum$ باشد. بر اساس *تقارن* احتمال این که این عدد $Minimum$ باشد با بقیه برابرِ است. پس احتمال این که عنصر $K+1$ کمینه باشد برابر $\frac{1}{K+1}$ و برابر با $\mathbb{E}[Y]$ است.

■

پرسش ۳

آ)

مانند پیش می دانیم که $|\Sigma| = 2^l$. اگر بتوانیم ورودی را $|\Sigma|$ بار بخوانیم، می توانیم DE را با حافظه ی $O(log(|\Sigma|))$ حل کنیم. می دانیم که Σ قابل ترتیب گذاری است. اگر هر مرحله از خواندن را با Run_j نشان دهیم که $j \in \{1, 2, ..., |\Sigma|\}$ و Σ را به صورت $\Sigma = \{a_1, a_2, ..., a_{|\Sigma|}\}$ نشان دهیم. در مرحله Run_j ابتدا a_j را در حافظه با هزینه ی $log(|\Sigma|)$ نگه میداریم. یک $counter$ هم برای نگهداری تعداد المانهای متمایز دیده شده تا به حال با حافظه ی $log(|\Sigma|)$ داریم. چراکه در بدترین حالت تعداد عناصر متمایز برابر تعداد الفبای ماست. یک بیت $flag$ هم با نام B داریم که اگر تا به حال a_j در مرحله Run_j دیده شده بود 1 و در غیر این صورت 0 است. در ابتدای هر Run_j هم مقدار این $flag$ به 0 آپدیت می شود. سپس در هر مرحله x_i را با a_j که پیشتر آن را ذخیره کردیم مقایسه می کنیم. اگر $a_j = x_i \; and \; B = 0$ آن گاه مقدار B به یک تغییر داده و $counter$ را نیز یکی افزایش می دهیم. توجه شود که در x_i های بعدی اگر پیشتر a_j دیده شده بود دیگر به $counter$ اضافه نمی شود. چراکه B از 0 به 1 تغییر پیدا کرده است. به این ترتیب در $R_{|\Sigma|}$ رخداد آخرین عضو از Σ هم بررسی شده و کل حافظه مصرفی نیز برابر $2.log(|\Sigma|) + 1$ است که برابر $O(log(|\Sigma|))$ است. از آنجا $|\Sigma| = 2^l$ پس حافظه مصرفی برابر $O(log(2^l)) = O(l)$ است.

ب)

اگر $p \in \{1, 2, ..., |\Sigma|\}$ آن گاه Σ را به p دسته ی $\frac{|\Sigma|}{p}$ تایی افراز می کنیم. اگر دسته ی kام را با S_k نشان دهیم، در هر Run_k مساله را برای الفبای $S_k \subseteq \Sigma$ که $|S_k| = \frac{|\Sigma|}{p}$ ، با همان شیوه پیشین انجام می دهیم. این کار را با اضافه کردن چند ریزه کاری به آنچه در *"الگوریتم۲"* پرسش یک انجام دادیم انجام می دهیم. به این ترتیب پس از p اجرا، برای همه ی دسته ها، عضویت اعضای استریم، یعنی x_iها در آن دسته ها بررسی شده و این کار با حافظه ی $O(\frac{|\Sigma|}{p}log(|\Sigma|))$ که همان $O(\frac{2^l}{p}l)$ است انجام می شود. آرایه ای که عناصر تا به حال دیده شده و درون الفبای S_k را نگه می دارد $currentList$ نام گذاری می کنیم. حال باید بدانیم که در کدام دسته ایم و اگر x_i فعلی در این دسته قرار نمی گرفت آن را $ignore$ کنیم، در غیر این صورت یعنی زمانی که $x_i \in S_k$ باید شرط دوم یعنی $x_i \in currentList$ بررسی شود، اگر این شرط برقرار نبود یعنی x_i که $x_i \in S_k$ بار اول است که دیده می شود. پس آن را به $currentList$ اضافه کرده و $counter$ را یکی افزایش می دهیم. برای بررسی شرط $x_i \in S_k$ در اجرای Run_k نگهداری $Min(S_k)$ و $Max(S_k)$ که آنها را با Min_k و Max_k نشان می دهیم کافی است. اگر $Min_k \leq x_i \leq Max_k$ یعنی $x_i \in S_k$ پس شرط $x_i \in currentList$ را بررسی می کنیم، در غیر این صورت آن را $ignore$ می کنیم. دسترسی به Max_k و Min_k مانند دسترسی به jامین اندیس Σ مانند آنچه در *"آ"* دیدیم می باشد. به علت قابل اندیس گذاری بودن Σ این کار انجام پذیر است. برای راحتی پیاده سازی k را هم که رنجی بین 1 تا p دارد نگهداری می کنیم. به این ترتیب 3 متغیر Min_k و Max_k و k به پیچیدگی $O(n.log(|\Sigma|))$ اضافه می شود. از آنجا که مقدار هر سه در رنج $[1, ..., |\Sigma|]$ قرار می گیرد الگوریتم دارای پیچیدگی حافظه ای $O(n.log(|\Sigma|)) + 3.log(|\Sigma|)$ است از آنجا که طبق فرض سوال $|\Sigma| \geq n^2$ پس این الگوریتم دارای پیچیدگی فضایی $O(\frac{|\Sigma|}{p}log(|\Sigma|))$ می باشد که همان $O(\frac{2^l}{p}.l)$ است.

پ)

ایده: ابتدا n را در Run_1 حساب کنیم. استریم را به p دسته ی $\frac{n}{p}$ تایی افراز می کنیم. به این شکل که در Run_{k+1} تنها kامین دسته ی $\frac{n}{p}$ تایی را مورد توجه قرار می دهیم و بقیه را $skip$ کنیم. گویی پنجره ای به طول $\frac{n}{p}$ داریم و در هر اجرا فقط به اعضایی که در آن پنجره قرار می گیرند توجه می کنیم. پس از هر اجرا نیز پنجره را $\frac{n}{p}$ ام به جلو شیفت می دهیم. به این شکل حافظه ی اصلی مورد استفاده در هر دور خواندن، $\frac{n}{p}$ خانه ی $log(|\Sigma|)$ بیتی می باشد. در حقیقت این الگوریتم به $p+1$ دور خواندن احتیاج دارد که $+1$ برای به دست آوردن n است.(این ایده در ادامه به تفضیل توضیح داده خواهد شد.)

ابتدا در اجرای اول n را به دست می آوریم. چون $2p$ بار می توانیم استریم را بخوانیم و $p \in \{1, 2, ..., |\Sigma|\}$ پس دست کم می توانیم 2 بار استریم را بخوانیم. زمانی که پیچیدگی حافظه تابعی از n است یعنی ما توانایی $alocate$ کردن حافظه براساس طول ورودی را به صورت پویا داریم. ابتدا آرایه ای به طول $\frac{n}{p}$ با نام $currentList$ می سازیم. یک شمارنده با نام $currentIndex$ تعریف می کنیم که در هر Run_k به صورت مستقل اندیس فعلی x_i را نگه می دارد. به عبارتی مقدار $currentIndex$ در x_i برابر i خواهد بود و پس از دیدن هر x_i در استریم یک واحد به آن اضافه می شود. همچنین در

ابتدای هر Run مقدار آن به 0 به روز رسانی خواهد شد. همچنین مقدار ثابت $increaseRate$ را پس از Run_1 به این شکل تعریف می کنیم: $increaseRate = \frac{n}{p}$ و در نهایت یک متغیر با نام و مقدار $skipBorder = 0$ تعریف می کنیم. همچنین متغیر $result$ را که تعداد المانهای متمایز را نشان میدهد تعریف می کنیم.
حال از Run_2 تا Run_{p+1} به این شکل الگوریتم را پیش می بریم: در Run_k ابتدا $currentIndex$ را با مقدار 0 به روز رسانی می کنیم. حال با دیدن هر x_i ابتدا $currentIndex$ را یک واحد افزایش می دهیم. سپس شرط زیر را چک می کنیم:

$$\underbrace{currentIndex > skipBorder \ \& \ (currentIndex - skipBorder) \leq increaseRate}_{\alpha}$$

اگر این شرط برقرار نبود یعنی این دسته باید $skip$ شود. پس به ازای هر x_i که در این شرط صدق نمی کرد هیچ کاری انجام نمی دهیم. در غیر این صورت، یعنی زمانی که این شرط برقرار بود، گویی در دسته ای هستیم که باید المانهای متمایز آن را بشماریم. برای این کار مانند "پرسش ۱" عضویت المان x_i را در $currentList$ چک می کنیم. در صورتی که $x_i \notin currentList$ ابتدا آن را به $currentList$ اضافه می کنیم و سپس $result$ را به $result + 1$ به روز رسانی می کنیم. زمانی که $currentIndex$ برابر n شد (مقدار n به راحتی از $increaseRate$ قابل محاسبه است) ، پس از انجام همین کارهای معمول ، باید $skipBorder$ را به $skipBorder + increaseRate$ به روز رسانی کرده و برای دورهای بعد هم به همین ترتیب عمل می کنیم.
توجه شود که در شرط α ، عضویت $currentIndex$ در مرحله ی Run_{k+1} در رنج $((k-1).\frac{n}{p}, k.\frac{n}{p}]$ چک می شود. ابتدا می بینیم که $currentIndex$ از $skipBorder$ اکیدا بزرگتر باشد. مقدار $skipBorder$ به ما می گوید که برای چه میزان از استریم تا به حال DE محاسبه شده است. اگر $currentIndex$ از $skipBorder$ اکیدا بزرگتر باشد یعنی در دسته ای هستیم که هنوز DE برای آن محاسبه نشده، ولی ممکن است در دسته ای باشیم که نباید DE برای آن در این Run_{k+1} محاسبه شود! منظور دسته هایی است که باید در مراحل بعد از Run_{k+1} محاسبه شوند.
اگر این بخش از α درست بود شرط زیر را چک می کنیم:

$$(currentIndex - skipBorder) \leq increaseRate$$

از آن جا که:

$$increaseRate = \frac{n}{p}$$

پس شرط زیر چک می شود:

$$(currentIndex - skipBorder) \leq \frac{n}{p}$$

توجه شود که ابتدا اکیدا بزرگتر بودن $currentIndex$ از $skipBorder$ چک شده است، پس امکان ندارد که بخش اول α درست بوده باشد و این بخش منفی باشد. حال اگر این شرط درست باشد یعنی در دسته ی kام که در مرحله Run_{k+1} بررسی می شود هنوز المانهایی هستند که برای آنها محاسبه DE انجام نشده و باید محاسبه شود. پس مانند آنچه در "پرسش ۱" داشتیم محاسبه را انجام می دهیم. زمانی که این شرط برقرار نباشد ولی شرط اول برقرار باشد یعنی این دسته باید در مراحل بعدی محاسبه شود و به راحتی آن را $skip$ می کنیم.

در این الگوریتم در مرحله دوم خواندن، $\frac{n}{p}$ تای اول بررسی می شود، در مرحله سوم $\frac{n}{p}$ تای دوم، به همین ترتیب در مرحله ی $p+1$ آخرین دسته ی $\frac{n}{p}$ تایی بررسی خواهد شد.

درستی: از آنجا که استریم را به درستی افراز کردیم، پس هیچ دسته ای با دسته ی دیگر اشتراکی ندارد و برای هر دسته DE دقیقا یکبار محاسبه می شود. یعنی محاسبه برای دسته های دو به دو مجزا ($Mutualy\ Exclusive$) انجام می شود. همچنین این افراز همه ی استریم را به درستی پوشش می دهد. یعنی محاسبه به صورت ریزمحاسبه هایی روی هم کامل ($Collectively\ Exhaustive$) انجام می شود.

آنالیز: اگر متغیرها و آرایه ای را که داشتیم به همراه سایزی که اشغال می کنند لیست کنیم و شرط $|\Sigma| \geq n^2$ را در نظر بگیریم، داریم:

$$\left.\begin{array}{l} Size\ of\ currentList = \frac{n}{p}.log(|\Sigma|) \\ Size\ of\ currentIndex = log(n) \\ Size\ of\ increaseRate = log(n) \\ Size\ of\ skipBorder = log(n) \\ Size\ of\ result = log(|\Sigma|) \end{array}\right\} (\frac{n}{p}+1).log(|\Sigma|) + 3.log(n) = O(\frac{n}{p}.log(|\Sigma|)) = O(\frac{n}{p}.l)$$

(اگر بخواهیم خیلی حساس باشیم برای فهم k در Run_k نیز باید یک $counter$ داشته باشیم که حافظه ی $log(|\Sigma|)$ می خواهد و در مرتبه بی تاثیر است، برای سادگی فرض می کنیم که می دانیم در Run چندم هستیم)

پرسش ۴

یک $Deterministic\ Communication\ Protocol$ روی دامنه ی $X \times Y$ و برد Z یک درخت دودویی است که هر $Internal\ Node$ با نام v دارای برچسب تابع $a_v : X \to \{0,1\}$ یا $b_v : Y \to \{0,1\}$ می باشد. به عبارتی هر نود دارای برچسب تابعی متمایز با دیگر نودهاست. همچنین بر این مبنا که نوبت آلیس باشد یا باب این تابع با آرگومان ورودی خود خروجی صفر یا یک دارد. [۴]
توجه شود که با وجود اینکه در این کلاس و منبع [۵] با مدل ساده شده ی پروتوکول که به شکل دوتایی (A, B) بود کار کردیم و می دانستیم که بیتهای فرد (اگر از 1 شروع کنیم) توسط آلیس و بیتهای زوج توسط باب فرستاده می شود، ولی مدل عمومی تر از پروتوکول به شکل سه تایی (A, B, N) می باشد که N تابع $NEXT : \{0,1\}^* \to \{A, B, STOP\}$ تعریف می شود. این تابع بر اساس $Communication\ History$ (*بیتهای رد و بدل شده تا به حال*) نفر بعدی را تصمیم می گیرد یا $Communication$ را پایان می دهد [۴]. پس این که برچسب نودها یک $level$ در میان a_v یا b_v هستند عمومی نیست ولی برای سادگی اینجا نیز همین فرض برقرار خواهد بود و اثباتهای پیش رو بدون از دست دادن عمومیت $(w.l.o.g.)$ در تعریف عمومی نیز برقرارند.
مقدار پروتوکول p روی ورودی (x, y) برچسب برگی است که برای رسیدن به آن از ریشه روی درخت حرکت می کنیم. در هر گره ی داخلی $(Intrernal\ Node)$ با نام v دو حالت داریم:
حالت اول: برچسب نود تابع a_v است. به چپ می رویم اگر که $a_v(x) = 0$ و به راست می رویم اگر که $a_v(x) = 1$
حالت دوم: برچسب نود تابع b_v است. به چپ می رویم اگر که $b_v(y) = 0$ و به راست می رویم اگر که $b_v(y) = 1$ [۴]

توجه شود که چون پروتوکول قطعی $(Deterministic)$ است، مسیر رسیدن به هر برگ برای یک ورودی خاص (x, y) یکتاست. هزینه پروتوکول p روی ورودی (x, y) که آن را با $cost(p)(x, y)$ نشان می دهیم، طول مسیر طی شده از ریشه به برگ متناظر است. همچنین هزینه ی یک پروتوکول که آن را با $cost(p)$ نشان می دهیم، طول بلندترین مسیر از ریشه به برگ است. اگر درخت یک پروتوکول را با T_p نشان دهیم، به وضوح هزینه ی پروتوکول p همان $height(T_p)$ است.

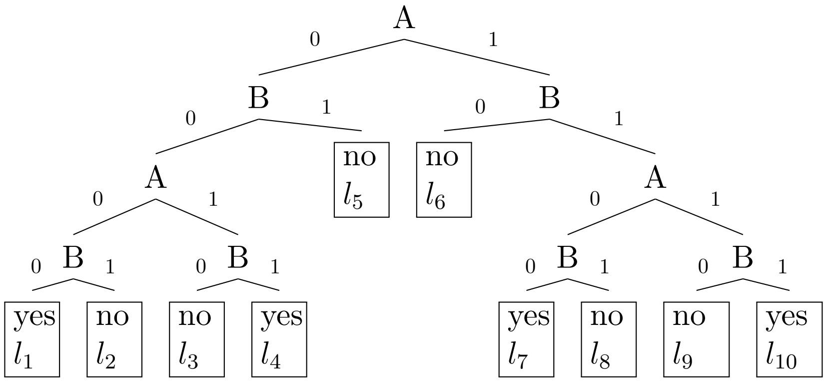
اثبات ([۳، ص. ۵۳]) :
مشاهده ۱: بین مجموعه $Communication\ History$ های ممکن یک پروتوکول خاص با نام p که از این پس این مجموعه را CH_p می نامیم و برگهای T_p تناظر یک به یک وجود دارد. کافیست به هر یال (u, v) مقدار خروجی تابع u را نسبت دهیم. به این ترتیب هر مسیر ریشه به برگ یک رشته ی باینری یکتا می دهد که CH متناظر است. همچنین برای یک CH خاص با نمایش $(b_1, b_2, ..., b_{r-1})$ از ریشه حرکت می کنیم و در هر مرحله با خواندن b_i اگر $b_i = 0$ به چپ می رویم و اگر $b_i = 1$ بود به راست می رویم. به این ترتیب حتما به یک برگ می رسیم چراکه در انتهای هر CH تابع حتما محاسبه شده است و برگی که به آن رسیدیم متناظر همان CH است. در نتیجه اگر برگهای T_p را با $Leaves(T_p)$ نشان دهیم آن گاه $CH_p \leftrightarrow Leaves(T_p)$ برقرار است.

مشاهده ۲: می توان گفت که $Leaves(T_p)$ فضای ورودی $X \times Y$ را افراز می کند. به این شکل که اعضای کلاس $[l]$ که l برگی از T_p است، شامل ورودیهایی است که با حرکت روی T_p ما را به برگ l می برند. اگر ماتریس M را که در صورت این پرسش معرفی شده با نام دقیق تری مثل M_f نامگذاری کنیم. آنگاه برای هر ورودی (x, y) در فضاِی ورودی درایه ی متناظر آن یعنی $M_f[x, y]$ برابر $f(x, y)$ است. از آنجا که $Leaves(T_p)$ فضای ورودی را افراز می کند و هر ورودی دارای متناظری در ماتریس M_f است، در نتیجه می توان گفت که $Leaves(T_p)$ ماتریس M_f را افراز می کند. پس از یک مثال، نشان می دهیم که هر کلاس هم ارزی $[l]$ یک $rectangle$ است. همچنین با این مشاهده نتیجه می گیریم که هر پروتوکول p برای تابع f یک افراز را روی ماتریس M_f القا $(induce)$ می کند.

مثال ۱: تابع $EQUALITY$ برای $n = 2$ را در نظر بگیرید. M_{EQ} برابر ماتریس I_4 است:

$$M_{EQ}\ with\ n = 2 : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

برچسب سطر و ستون ماتریس را ترتیب عادی در نظر بگیرید یعنی $(00, 01, 10, 11)$. حال پروتوکول زیر را که مساله $EQUALITY$ را حل می کند در قالب درخت دودویی داریم:



توجه شود که نیازی نیست حتما یک پروتوکول بهینه اینجا بیاوریم. در این پروتوکول آلیس و باب خیلی ساده شروع می کنند به ترتیب بیتهای خود را فرستادن و همزمان مقایسه کردن آنها. به همین علت این پروتوکول بهینه نیست. بر روی درخت به این شکل به پروتوکول نگاه شود که تابع متناظر با ریشه بیت اول X را به عنوان خروجی برمی گرداند. حال اگر بیت اول برای مثال 1 بود، آن گاه به راست می رویم و تابع متناظر راست بیت اول Y را به عنوان خروجی بر می گرداند. پروتوکول ساده ای است و به دست آوردن تک تک تابع های هر نود آن هم ساده است. حال می بینیم که پروتوکول چگونه ماتریس $M_{EQ}\ with\ n = 2$ را افراز می کند.
برای هر ورودی کلاس هم ارزی را به دست می آوریم (ورودی ها به فرم (x, y) هستند):

۱. $(00, 00) \in [l1]$ ۲. $(00, 01) \in [l2]$ ۳. $(00, 10) \in [l5]$ ۴. $(00, 11) \in [l5]$ ۵. $(01, 00) \in [l3]$
۶. $(01, 01) \in [l4]$ ۷. $(01, 10) \in [l5]$ ۸. $(01, 11) \in [l5]$ ۹. $(10, 00) \in [l6]$ ۱۰. $(10, 01) \in [l6]$
۱۱. $(10, 10) \in [l7]$ ۱۲. $(10, 11) \in [l8]$ ۱۳. $(11, 00) \in [l6]$ ۱۴. $(11, 01) \in [l6]$ ۱۵. $(11, 10) \in [l9]$
۱۶. $(11, 11) \in [l10]$

همه ی کلاسهای هم ارزی را لیست می کنیم، اینبار اعضای هر کلاس را با $M_{EQ}[i, j]$ به جای (x, y) نشان می دهیم.

$[l_1] = \{M_{EQ}[1, 1]\}$
$[l_2] = \{M_{EQ}[1, 2]\}$
$[l_3] = \{M_{EQ}[2, 1]\}$
$[l_4] = \{M_{EQ}[2, 2]\}$
$[l_5] = \{M_{EQ}[1, 3], M_{EQ}[1, 4], M_{EQ}[2, 3], M_{EQ}[2, 4]\}$
$[l_6] = \{M_{EQ}[3, 1], M_{EQ}[3, 2], M_{EQ}[4, 1], M_{EQ}[4, 2]\}$
$[l_7] = \{M_{EQ}[3, 3]\}$
$[l_8] = \{M_{EQ}[3, 4]\}$
$[l_9] = \{M_{EQ}[4, 3]\}$
$[l_{10}] = \{M_{EQ}[4, 4]\}$

شکل افراز القا شده روی ماتریس $M_{EQ}\ with\ n = 2$:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

روی شکل $EQ\_Monochromatic$ بودن همه ی مستطیلها ی حاصل از افراز مشخص است. در ادامه پس از اثبات دو لِم، اثبات می کنیم که این اتفاق همیشه خواهد افتاد.

لم ۱: برای هر برگ l_i در T_p اعضای کلاس هم ارزی $[l_i]$ یک $Combinatorial\ rectangle$ تشکیل می دهند. [۶]

توجه شود که در منابع مختلف برای اثبات معادلهای مختلف این لِم از استقرا روی طول بیتهای جا به جا شده تا به حال (مانند [۳]) یا استقرا روی عمق T_p (مانند [۶]) استفاده می کنند. در "*مشاهده ۱*" دیدیم که این دو باهم در تناظرند و اینجا با استقرا روی عمق بر اساس [۶] این لم را اثبات میکنیم.

اثبات: در اینجا حکم قوی تری را اثبات می کنیم؛ برای هر نود v در T_p اگر مجموعه ورودیهایی که از این نود رد می شوند را R_v بگیریم، آن گاه R_v روی ماتریس M_f یک $Combinatorial\ recatngle$ است. (توجه شود که این لم کاری به تکرنگی ندارد و لِم بعدی به آن خواهد پرداخت)

استقرا روی عمق T_p می زنیم. در حالت پایه روی نود ریشه v_0 هستیم. در این حالت به وضوح تمام رشته ها در فضای ورودی درون R_{v0} قرار می گیرند. پس $R_{v0} = M_f$ و M_f خود یک $Combinatorial\ recatngle$ است. پس برای حالت پایه برقرار است.

برای قدم استقرا، فرض کنید برای نود w فرض برقرار است، اثبات می کنیم برای فرزند w با نام v نیز حکم بر قرار است. بدون از دست دادن عمومیت (w.l.o.g.) فرض کنید، v فرزند چپ w بوده و w برچسب a_v دارد. یعنی در w نوبت با آلیس بوده و جواب $a_v(x)$ هم برابر 0 بوده است. همچنین توجه شود که به دلیل rectangle بودن R_w می توانیم آن را به شکل $A_w \times B_w$ نشان دهیم. چرا که بر اساس تعریف:

$$R_w \subset X \times Y\ s.t.\ R_w = A_w \times B_w$$

که $A_w \subset X$ و $B_w \subset Y$ حال داریم:

$$R_v = R_w \cap \{(x,y)|a_w(x)=0\} = \underbrace{(A_w \cap \{x|a_w(x)=0\})}_{\alpha} \times B_w$$

از آنجا که $a_w(x)$ تنها بر اساس ورودی x خروجی می دهد پس در $R_w = A_w \times B_w$ روی A_w تاثیر می گذارد. از آنجا که α حاصل اشتراک A_w با مجموعه ای دیگر است، پس $\alpha \subseteq A_w$ از آنجا که $\subset$ یک رابطه ی $transitive$ است و بر اساس فرض استقرا $A_w \subset X$ پس $\alpha \subset X$ و همچنین بر اساس فرض می دانستیم که $B_w \subset Y$ و در نهایت $\alpha \times B_w \subset X \times Y$ که نشان می دهد R_v یک $Combinatorial\ rectangle$ است.
■

لم ۲: اگر پروتوکول p تابع f را محاسبه کند آن گاه برای هر $rectangle$ القا شده $(induced)$ توسط p در M_f این $rectangle$ تکرنگ یا $f\_Monochromatic$ است.

یک مستطیل دلخواه القا شده توسط p در M_f را در نظر بگیرید. در "*مشاهده ۲*" دیدیم که این مستطیل توسط یکی از اعضای $\{R_{li}\}_{l_i \in Leaves(T_p)}$ القا شده است. می دانیم که اعضای این افراز یا R_{li} ها همان کلاسهای هم ارزی l_i هستند. می دانیم که هر $[l_i]$ یک مجموعه است شامل ورودیهایی به فرم (x,y) که به l_i ختم می شوند. خروجی p هم برای این ورودیها ثابت ($constant$) است. همچنین این خروجی همان برچسب برگ یعنی $z_l \in Z$ است. از آنجا که p به درستی f را محاسبه می کند پس برای هر کدام از این (x,y) ها که به l_i ختم می شوند و عضو $[l_i]$ هستند نیز $f(x,y) = z_l$ در نتیجه برای همه ی (x,y) هایی که به l_i ختم می شوند هم $M_f[x,y] = z_l$ و در نهایت نتیجه میگیریم که مستطیل القا شده توسط $[l_i]$ یک مستطیل $f\_Monochromatic$ است.
■

قضیه نهایی: اگر f را تابعی بگیریم که در آن حداقل مستطیل $Monochromatic$ لازم برای افراز M_f برابر k باشد. اگر پیچیدگی ارتباطی f را با $D(f)$ نشان دهیم. آن گاه $log_2(k) \le D(f)$.

اثبات: یک پروتوکول قطعی با هزینه ی $cost(p)$ تعداد $Communication\ History$ متمایز آن حداکثر برابر $2^{cost(p)}$ است. متناظر آن T_p دارای $2^{cost(p)}$ برگ متفاوت است. اگر این پروتوکول f را محاسبه کند، بر اساس "لم۱" و "لم۲" ماتریس M_f را به $2^{cost(p)}$ مستطیل $f\_Monochromatic$ افراز می کند. بر اساس فرض حداقل تعداد این $f\_Monochromatic$ $rectangle$ ها برابر k است. پس $k \leq 2^{cost(p)}$ و در نتیجه $log_2(k) \leq cost(p)$. از آنجا که این حداقل هزینه ی ممکن برای محاسبه f است، پس $log_2(k) \leq D(f)$.

■

مراجع

[۱] Ryan Williams, Luca Trevisan, *Notes on Streaming Algorithms* ,M.I.T. 6.045 "https://people.csail.mit.edu/rrw/6.045-2020/notestream.pdf"

[۲] David R. Karger, *Streaming Algorithms* ,M.I.T. 6.854 "http://courses.csail.mit.edu/6.854/16/Notes/n5-streaming.html"

[۳] Tim Roughgarden, *Communication Complexity* ,Stanford CS369E "https://arxiv.org/pdf/1509.06257.pdf"

[۴] Alexander Sherstov, *Communication Complexity* ,UCLA CS289 "http://web.cs.ucla.edu/~sherstov/teaching/2012-winter/docs/lecture01.pdf"

[۵] Ryan Williams, *Notes on Communication Complexity* ,M.I.T. 6.045 "https://people.csail.mit.edu/rrw/6.045-2020/notestream.pdf"

[۶] Yaron Singer , *Communication Complexity* ,Harvard CS 284r "https://people.seas.harvard.edu/~yaron/SocialDataMining/lecture_notes/lecture2.pdf"