۲ . ۵ نمایش object ها:

ما علاوه بر اعداد، object های دیگه‌ای هم هستند که می‌توانیم آن‌ها را نشان دهیم. برای این کار به توابع encode و decode نیاز داریم تا object را به string و string را به object تبدیل کنیم.

Definition ۲.۱۳ نمایش string ای:

فرض کنید که O هر مجموعه دلخواهی باشد. یک representation scheme برای O، تابع‌های دوتایی به صورت E و D هستند که به صورت زیر تعریف می‌شوند:

$$E : O \to \{0,1\}^* \qquad E\text{: تابع یک‌به‌یک}$$

$$D : \{0,1\}^* \xrightarrow[p]{} O \qquad D\text{: تابع پوشا}$$

$$D\big(E(o)\big) = o \qquad \text{معکوس هم}$$

Lemma ۲.۱۴:

برای نمایش یک مجموعه مانند O، ما تنها نیاز به یک تابع encoding یک‌به‌یک داریم. (تابع decode از روی همین یک‌به‌یک پیدا میشه)

اثبات:

فرض کنید o_0 یک عضو از مجموعه O باشد. می‌دانیم که هر string باینری‌ای (هر عضو مجموعه {0, 1}*) با توجه به یک‌به‌یک بودن encoding یا باید حاصل map شدن یکی از اعضای O به string باشه یا اینکه برای مپ شدن هیچکدام از اعضای O باشه. حالا یک تابع D تعریف می‌کنیم که string های باینری ({0, 1}*) را به عنوان ورودی بگیره، و در خروجی اون عضوی را بدهد که آن نمایش باینری را داشت (E آن عضو را به آن string باینری map کرده بود)؛ اگر هم هیچ عضوی از O، اون نمایش باینری را نداشت، خروجی تابع D، o_0 باشد.

هر تابع decoding ای که partial (روی تمام ورودی تعریف نشده باشد) باشد را هم می‌توان با قرار دادن یک خروجی الکی برای جاهایی که ورودی در آن تعریف نشده است، به تابع total تبدیل کرد.

۲.۵.۱ نمایش متناهی (finite representation):

اگر مجموعه O متناهی باشد، می‌توانیم تمام اعضای مجموعه O را با string های باینری‌ای با حداکثر طول n نشون بدیم.

تعداد کل رشته‌های متمایزی که با طول حداکثر n داریم، میشه جمعه رشته‌هایی به طول صفر، با رشته‌هایی به طول یک با ... با رشته‌هایی به طول n. یعنی:

$$|\{0,1\}^0| + |\{0,1\}^1| + \cdots + |\{0,1\}^n| = \sum_{i=1}^{n} 2^i = 2^{n+1} - 1$$

:*Lemma 2.16*

برای هر دو تا مجموعه متناهی و غیر تهی *S* و *T* یک تابع یک‌به‌یک از *S* به *T* وجود دارد (*F: S -> T*) اگر و تنها اگر $|T| \leq |S|$ باشد.

اثبات:

برای حالتی که $|T| \leq |S|$ برقرار باشد هم اعضای هر دو مجموعه *S* و *T* را به ترتیب می‌نویسیم و تابع *F* را به این صورت در نظر می‌گیریم که عضو اول *S* را به عضو اول *T* *map* کند. و عضو دوم*S* را به عضو دوم *T* و ... . طبیعتا چون مجموعه هستند اعضا تکراری نیست و تابع یک‌به‌یک است و هیچ عضوی از *T* را دو عضو از *S* نسبت ندادیم.

رد شدن برای حالتی که $|T| > |S|$ باشد با اصل لانه کبوتری است. پس باید حتما $|T| \leq |S|$ باشد.

:*prefix-free encoding 2.5.6*

اگر نحوه *represent* کردن ما به گونه ای باشد که نمایش باینری هیچ دو عضو متفاوتی *prefix* همدیگر نباشد، می‌توانیم بگوییم که نمایش ما *prefix-free* است.

ما می‌توانیم هر *representation* ای را به *prefix-free* تبدیل کنیم.

:*prefix-free encoding Definition 2.1*

تعریف ریاضی *prefix* بودن دو تا *string* باینری *y* و *y’* به این صورت است که *|y|≤|y’|* باشد و برای تمامی *i* های کوچک تر از *|y|*، $y_i = y'_i$ برقرار باشد.

تابع *E* یک *encoding* است و تنها وقتی می‌گوییم *E*، *prefix-free* است که هیچ دو عضو متمایز *o* و *o’* ای وجود نداشته باشند که *E(o)* و *E(o’)*، *prefix* یکدیگر باشند.

یادآوری: O^* برابر است با تمام لیست‌هایی با طول متنهای که هر عضو این لیست‌ها، عضوی از *O* باشد.

: *Prefix-free implies tuple encoding – Theorem 2.18*

فرض کنیم که $E : O \to \{0,1\}^*$ یک تابع *prefix-free* باشد. تابع (*mapping*) $\bar{E} : O^* \to \{0,1\}^*$یک‌به‌یک است و به صورت $\bar{E}(o_0, o_1, \dots, o_{k-1}) = E(o_0)\, E(o_1) \dots E(o_{k-1})$ تعریف می‌شود.

ایده اثبات: استفاده از اینکه *encoding* هیچ دو عضوی *prefix* یکدیگر نیستند.

اثبات:

برای این اثبات از برهان خلف استفاده می‌کنیم. برای این کار فرض می‌کنیم که دو تا لیست (*tuple*) مجزا (غیر یکسان) مانند $(o_0, o_1, \dots, o_{k-1})$ و $(o'_0, o'_1, \dots, o'_{k-1})$داشته باشیم به طوریکه $\bar{E}(o_0, o_1, \dots, o_{k-1}) = \bar{E}(o'_0, o'_1, \dots, o'_{k-1})$ باشد.

مقدار $\bar{E}(o_0, o_1, \ldots, o_{k-1})$ را با $\bar{x}$ نشان می‌دهیم. فرض می‌کنیم که *i* اولین *index* ای باشد که در آن $o_i \neq o'_i$ باشد. (چون در ابتدا گفتیم که این دو تا لیست از هم مجزا هستند، پس حداقل یک عضو غیر یکسان دارند) در حالی که *i<k* ، ما می‌توانیم $\bar{x}$ را به دو روش بنویسیم:

$$\bar{x} = \bar{E}(o_0, o_1, \ldots, o_{k-1}) = x_0 \ldots x_i\, E(o_i)\, E(o_{i+1}) \ldots E(o_{k-1})$$

$$\bar{x} = \bar{E}(o'_0, o'_1, \ldots, o'_{k-1}) = x_0 \ldots x_i\, E(o'_i)\, E(o'_{i+1}) \ldots E(o'_{k-1})$$

حالا اگر $x_0 \ldots x_i$ های ابتدای هر دو حالت را برداریم، پس باید $E(o_i)\, E(o_{i+1}) \ldots E(o_{k-1}) = E(o'_i)\, E(o'_{i+1}) \ldots E(o'_{k-1})$ شود که اگر فقط اولین قسمت از هر دو *string* را مورد توجه قرار دهیم، $E(o_i)s = E(o'_i)\, s'$ خواهد بود. حالا برای اینکه این دو مقدار با یکدیگر برابر باشند (با توجه به اینکه o_i و o'_i با یکدیگر یکسان نیستند) پس باید یکی *prefix* دیگری باشد. پس به تناقض می‌خوریم در این حالت (در حالت *i<k*).

در حالتی که *k= i* باشد و همچنین *k’>k* باشد، پس داریم:

$$\bar{x} = E(o_0) \ldots E(o_{k-1}) = E(o_0) \ldots E(o_{k-1}) E(o'_k)\, E(o'_{k+1})\, E\left(o'_{k'-1}\right)$$

که در این حالت پس باید $E(o'_k)\, E(o'_{k+1})\, E\left(o'_{k'-1}\right)$ برابر با *string* خالی (“”) باشد که خب طبیعتا **این *string* خالی، *prefix* همه *string* هاست**. پس به تناقض می‌خوریم در این حالت هم.

:*prefix-freeness of list representation Remark 2.19*

حتی اگر *E* ما برای نمایش یک مجموعه *O*، *prefix free* باشد، دلیل نمی‌شود که $\bar{E}$ ما هم *prefix-free* باشد. **در اصل $\bar{E}$ ، دیگر *free_prefix* نیست**. چون نمایش *(o, o’)*، قطعا *prefix* برای نمایش *(o, o’, o’’)* می‌باشد. (منظور از نمایش همان حاصل تابع $\bar{E}$ می‌باشد)

هر نمایش غیر *prefix-free* ای را می‌شود *Prefix-free* کرد. (پس طبیعتا نمایش لیستی را هم می‌شود *prefix-free* کرد)

2.3.5 تبدیل کردن نمایش‌ها به *prefix-free* :

اگر یک *representation* ای طول خروجی‌هایش ثابت باشد (یعنی به ازای تمام ورودها، خروجی‌هایی با طول ثابت بدهد) یا به تعریف ریاضی، حالت $E : O \to \{0,1\}^n$ را داشته باشد، به صورت طبیعی *prefix-free* است. (چون طول خروجی‌ها با یکدیگر برابر است پس برای *prefix* بودن باید دقیقا یکی شوند که خب در این صورت دیگر یک‌به‌یک بودن نقض می‌شود)

:*Lemma 2.20*

فرض کنیم که $E : O \to \{0,1\}^*$ یک تابع یک‌به‌یک باشد. پس یک تابع *encoding* یک‌به‌یک *prefix-free* $\bar{E}$ وجود دارد به طوریکه $|\bar{E}(o)| \leq 2|E(o)| + 2$ باشد.

اثبات:

ایده پشت این حرکت این است که طول رشته خروجی را دو برابر کنیم، به طوریکه *0* تبدیل بشه به *00*، *1* تبدیل بشه به *11* و در انتهای رشته هم *01* بگذاریم. (فرض می‌کنیم که دو تا رشته داریم که یکی *prefix* دیگری است. حالا پس از این کار، تا دو برابر طول رشته کوچکتر که می دانیم همچنان *prefix* رشته به طول بزرگتر است ولی آن *01* انتهایی کار را خراب می‌کند. چون در رشته به طول بزرگتر ما یا *11* داریم یا *00* چون اگر *01* داشته باشیم یعنی رشته‌ها هم طول هستند و یکی بودند و یک‌به‌یک بودن نقض می‌شود. پس قطعا دیگر رشته به طول کوچکتر *prefix* دیگری نیست.)

پس برای تبدیل هر خروجی از نحوه نمایش غیر ***prefix-free*** از تابع زیر استفاده می‌کنیم:

$$PF: \{0,1\}^* \rightarrow \{0,1\}^*$$

$$PF(x) = x_0 x_0 x_1 x_1 \dots x_{n-1} x_{n-1}\ 01$$

Prefix-freeness در اصل شرط قوی‌تر و جامع‌تری به نسبت شرط یک‌به‌یک است. (چون اگر در اصل خروجی برایدو ورودی متفاوت یکسان شود، پس در اصل چون یک ریشته ***prefix*** خودش است، پس در اصل نمایش و ***representation*** یا همان خروجی این دو عضو متفاوت، ***prefix*** هم می‌شوند.)

اثبات با کد پایتون: صفحه ۱۰۶ و ۱۰۷

تبدیل ***encoding*** تکی (ورودی یک عضو از مجموعه، نه لیستی از اعضا) به ***prefix-free*** :

ورودی ***encode***: تابعی برای ***encode*** کردن که در حالت عادی داریم.

ورودی ***decode***: تابعی برای ***decode*** وردن که در حالت عادی داریم.

خروجی ***pfencode***: یک تابع می‌سازد. کار این تابع این است که ابتدا با استفاده از تابع ***encode*** ورودی، را ***encode*** می‌کند سپس با روش توضیح داده شده، خروجی تابع ***encode*** را به ***Prefix-free*** تبدیل می‌کند. (با تبدیل ***0*** به ***00*** ، ***1*** به ***11*** و گذاشتن ***01*** در انتها)

خروجی ***pfdecode***: یک تابع می‌سازد. کار این تابع این است که ابتدا روی ورودی پیمایش می‌کند سپس از ابتدا شروه می‌کند و تمام ***00*** ها را به ***0***، تمام ***11*** ها را به ***1*** تبدیل می‌کند و آن ***01*** انتهایی را هم ایگنور می‌کند. سپس خروجی را به تابع ***decode*** ورودی می‌دهد.

خروجی ***pfvalid***: چک می‌کند که آیا ***string*** ورودی حاصل از تبدیل گفته شده برای ***prefix-free*** ساختن می‌باشد یا خیر. (یعنی اول زوج بودن طول ***string*** را بررسی می‌کند. **و** (***and***) بعد از آن از ابتدا شروع می‌کند و میبیند هر دو تا بیت پشت هم (به جز دو تا بیت آخر) یکسان هستند یا خیر **و** (***and***) همچنین آیا در انتها دو تا بیت آخر ***01*** هستند یا خیر.)

```
def prefixfree(encode, decode):
    def pfencode(o):
        L = encode(o)
        return [L[i//2] for i in range(2*len(L))]+[0,1]
    def pfdecode(L):
        return decode([L[j] for j in range(0,len(L)-2,2)])
    def pfvalid(L):
        return (len(L) % 2 == 0 ) and all(L[2*i]==L[2*i+1]
         ↪ for i in range((len(L)-2)//2)) and
         ↪ L[-2:]==[0,1]

    return pfencode, pfdecode, pfvalid

pfNtS, pfStN , pfvalidN = prefixfree(NtS,StN)
```

تبدیل ***encoding*** لیستی (ورودی یک لیستی از اعضا) به ***prefix-free*** :

ورودی ***pfencode***: دقیقا همان خروجی تابع قبلی

ورودی ***pfdecode***: دقیقا همان خروجی تابع قبلی

ورودی ***pfvalid***: دقیقا همان خروجی تابع قبلی

خروجی ***encodelist*** : از همان تابع ورودی ***pfencode*** استفاده می‌کند و همه عضوهایی که باید ***encode*** شوند را به صورت تکی و به ترتیب ***encode*** می‌کند سپس آن‌ها را بهم می‌چسباند.

خروجی *decodelist* : دو تا متغیر *i* و *j* را در نظر می‌گیرد که نشان‌دهنده بازه از *string* ورودی هستند. سپس مقدار *j* را همواره (تا قبل از اینکه به انتهای *string* ای برسد که می‌خواهد آن را *decode* کند) سپس *string* ورودی را (که می‌خواهد آن را *decode* کند را) از بازه *i* تا *j* اش را به تابع *pfvalid* می‌دهد، اگر نتیجه مثبت بود، یعنی آن بازه *I* تا *j* شامل یک ورودی از مجموعه ورودی اصلی بوده و آن را به تابع *pfdecode* می‌دهد و مقدار *i* را برابر با *j* می‌گذارد تا به سراغ بازه بعدی برود. در هر حالت هم مقدار *j* را یکی افزایش می‌دهد.

```python
def represlists(pfencode,pfdecode,pfvalid):
    """
    Takes functions pfencode, pfdecode and pfvalid,
    and returns functions encodelists, decodelists
    that can encode and decode lists of the objects
    respectively.
    """

    def encodelist(L):
        """Gets list of objects, encodes it as list of
        ↪ bits"""
        return "".join([pfencode(obj) for obj in L])

    def decodelist(S):
        """Gets lists of bits, returns lists of objects"""
        i=0; j=1 ; res = []
        while j<=len(S):
            if pfvalid(S[i:j]):
                res += [pfdecode(S[i:j])]
                i=j
            j+= 1
        return res

    return encodelist,decodelist
```

: *representing letters and text 2.5.5*

ASCII یک نحوه *represent* کردن با **طول ثابت** خروجی (*fixed-length*) است که پس به صورت خودکار هم ***prefix-free*** است.

:representing vectors, matrices and images 2.5.6

وقتی بتوانیم اعداد را نشان بدهیم، پس می‌توانیم لیستی از اعداد را هم نشان بدهیم و *vector* ها هم لیستی از اعداد هستند، پس *vector* ها هم قابل نمایش هستند.

:representing graphs 2.5.7

گرافی با *N* راس، می‌تواند با ماتریس مجاورت *n*n* نمایش داده شود. (به طوریکه درایه *i* و *j* برابر با یک است، اگر بین این دو راس *i* و *j*، یالی وجود داشته باشد. وگرنه مقدار آن درایه برابر با صفر است)

پس ما می‌توانیم یک گراف جهت‌دار با *n* راس را با یک *string* ای مثلی *A* که متعلق به $\{0,1\}^{n^2}$ است، نمایش دهمی به طوریکه $A_{i,j} = 1$ است اگر و تنها اگر یک یال از راس *i* به راس *j* داشته باشیم.

یک راه دیگر برای نمایش گراف‌ها هم، لیست مجاورت (*adjacency list*) است. (برای هر گره یک لیستی نگه می‌داریم که شامل همسایه‌های خروجی آن در گراف جهت دار است.)

: *representing lists and nested lists 2.5.8*

اگر ما روشی برای نمایش اعضای مجموعه *O* داشته باشیم، پس می‌توانیم با تبدیل ***prefix-free*** ، لیستی از این اعضا را هم نمایش دهیم. حتی با یک حرکت (*trick*) می‌تونیم لیست‌های تودرتو را هم هندل کنیم. ایده آن هم این است که نمایش ما برای اعضای مجموعه *O* به صورت $E : O \rightarrow \{0,1\}^*$ باشد. سپس یک الفبای (*alphabet*) جدید به صورت $\sum = \{0, 1, [,], ,\}$ تعریف می‌کنیم. سپس هرجا خواستیم یک لیست درونی تعریف کنیم، اعضای داخل آن را بین *[]* قرار می‌دهیم. مثلا: $(o_1, (o_2, o_3))$ را با حالتی مانند *"[0011, [10011, 00111]]"* نمایش می‌دهیم.

: *notation 2.5.9*

وقتی که ما می‌گوییم که *A* یک الگوریتم است که عمل ضرب بر روی اعداد طبیعی را محاسبه می‌کند، منظور اصلی‌مان این است که *A* یک الگوریتم است که تابع $F : \{0,1\}^* \rightarrow \{0,1\}^*$ را محاسبه می‌کند به طوریکه برای هر دوتایی *a* و *b* عضو اعداد طبیعی، اگر $x \in \{0,1\}^*$ نمایش دهنده دوتایی *a* و *b* باشد (نمایش دهنده (*a,b*) باشد)، پس *F(x)* نمایشگر *string* ای است که حاصل *a.b* می‌باشد.

: *defining computational task as mathematical functions 2.6*

Computational process یک پراسسی است که به عنوان ورودی *string* ای از بیت‌ها (صفر و یک) را دریافت می‌کند و به عنوان خروجی *string*ای از بیت‌ها تولید می‌کند. این *computational task* ها در اصل همان *specification* است. (یک سطح انتزاعی یا *abstract*)

: *Boolean functions and languages – Remark 2.23*

Boolean function ها در اصل نوعی از *computational task* ها هستند که **خروجی آن‌ها یک بیت می‌باشد.** **(صفر یا یک)** که این مانند جواب دادن به یک سوال *yes/no* (آره یا نه، هست یا نیست و ...) است. این *task* ها را هم به عنوان *decision problem* هم درنظر می‌گیرند.

برای مثال در هر تابع مانند $F : \{0,1\}^* \rightarrow \{0,1\}$ ، اگر تمام ورودی‌هایی از تابع *F* را که به ازای آن‌ها خروجی تابع *F* برابر با یک است را در مجموعه *L* (*Language*) قرار دهیم (یعنی به طور ریاضی تعریف *L* برابر است با $L = \{x : F(x) = 1\}$)، پس تعریف *computation task* مربوط به تابع *F(x)* به طوری که برای هر *x* به طوریکه $x \in \{0,1\}^*$ ، برابر است با تصمیم‌گیری بر اینکه آیا *x* عضو مجموعه *L* هست یا خیر. (به این *deciding language* هم می‌گویند) به اصطلاح، به *L* ، زبان تابع *F* هم می‌گویند. پس *computational task* مطابق با *deciding language* است.

برای یک تابع *F* ، چندین الگوریتم ممکن است برای محاسبه آن وجود داشته باشد که تابع *F* را محاسبه کنند.

2.6.1 تفاوت *function* و *program* ها

Specification برابر است با *mathematical function* .

Implementation برابر است با *algorithm/program* .

برای مثال، روش‌های مختلف ضرب، برنامه‌هایی متفاوت هستند که یک *mathematical function* را محاسبه می‌کنند.

Program ، یک *function* را محاسبه می‌کند.

Function : یک *mapping* از *input* ها به *output* ها (*mapping* از ورودی‌ها به خروجی‌ها)

Program : یک سری از *instruction* ها (دستورالعمل‌ها) برای چگونگی (*how*) محاسبه و رسیدن به *output* از روی *Input* است.

computation beyond function – Remark 2.24 :

- تیکه اول: برای محاسبه *partial function* ها ما نیازی نیست که نگران جاهایی باشیم که ورودی در آن تعریف نشده است. ما می‌توانیم برای *partial function* ها در نظر بگیریم که حتما *function* در تمام ورودی تعریف شده است. (یک نفر به ما قول داده) و اگر ورودی‌ای بیاید که تابع در آن تعریف نشده باشد، خروجی آن برای ما مهم نیست.

  به اینکه یک نفر به ما قول داده است که تمام *input* ها در تابع تعریف شده اند، به این قول دادن *promise problem* می‌گویند.

- تیکه دوم: *Relation* را برابر با این در نظر می‌گیریم که برای یک ورودی تابع ممکن است بیشتر از یک خروجی قابل قبول داشته باشد. *Computational process* ای که به ازای هر ورودی *x* ، می‌آید و تمام خروجی‌های قابل قبول را پیدا می‌کند (یعنی تمام دوتایی‌های *(x, y)* را پیدا می‌کند به طوری که *y* خروجی‌های مختلف قابل قبول برای ورودی *x* باشد) یک *relation* را حل می‌کند.

خلاصه فصل:

- می‌توانیم *object* هایی را که می‌خواهیم رویشان محاسبه انجام دهیم، با *binary string* نمایش دهیم.
- *representation* برای یک مجموعه از *object* ها، یک تابع یک‌به‌یک از مجموعه *object* ها به $\{0,1\}^*$ است.
- می‌توانیم یک نمایش *prefix-free* برای مجموعه‌ای از *object* ها را ارتقا بدهیم تا بتوانیم لیستی از *object* ها را هم نمایش بدهیم.
- یک *computational task* ساده، *task* ای است برای *computing a function* به طوریکه $F : \{0,1\}^* \rightarrow \{0,1\}^*$ باشد. این برای خیلی از چیزها مثل *image processing* و ... به کار می‌رود و فراتر از کارهای محاسبات ریاضی ساده (arithmetic) است.