گزارش پروژه سوم درس طراحی الگوریتم‌ها (James Bond)

امیرعلی صادقی فرشی (۹۹۱۲۸۳۴)

دیتاست مورد استفاده: برای این پروژه از لغت‌نامه‌ی انگلیسی آکسفورد که در این لینک موجود می‌باشد استفاده شده است. به‌طور دقیق‌تر، از فایل Oxford 3000 Word List.txt استفاده شده است. در این لغت‌نامه بعضی کلمات به‌صورت تکراری ظاهر شده‌اند؛ مثلاً close. در فایل ذکر شده مثلاً برای close، دو سطر close 1 و close 2 وجود دارد اما در فایل دیگر (Oxford 3000 Word List No Spaces.txt) به‌صورت close_1 و close_2 وجود دارد. شیوه‌ی اول برای کار پردازش راحت‌تر است چون با استفاده از تابع `split()` در پایتون می‌توان محتویات داخل فایل را چه با استفاده از کاراکتر New Line و چه با استفاده از فاصله جدا کرد و بنابراین به‌راحتی می‌توانیم close را نیز به دست آوریم. دو پیش‌پردازش بر روی این دیتاست انجام شد که به شرح زیر است:

۱. تمام حروف کلمات به حروف کوچک تبدیل شدند. هدف از این کار این است که اگر به‌عنوان مثال کلمه‌ی I به معنی «من» در جمله موجود باشد و چون جمله‌ی ورودی تماماً با حروف بزرگ است، ما ناچاریم همه‌ی حروف را ابتدا کوچک کنیم و بنابراین برای مطابقت با لغت‌نامه، تمام حروف کلمات در لغت‌نامه نیز باید کوچک باشند.
۲. بعضی افعال ساده و اساسی زبان انگلیسی در این دیتاست موجود نبودند که به آن اضافه شدند. این کلمات عبارتند از:
is - did - am - are - was - were - didnt - werent - wasnt - arent - isnt
لازم به ذکر است افعال منفی مخفف، بدون کاراکتر ' اضافه شده‌اند چون در جمله‌ی ورودی نیز همه‌ی علائم ورودی حذف شده‌اند.

الگوریتم مورد استفاده: قبل از توضیح الگوریتم، به توضیح تابع استفاده‌شده می‌پردازیم. تابع بازگشتی valid را به‌صورت زیر تعریف می‌کنیم:

ورودی‌ها: ۱- زیررشته‌ی باقی‌مانده از جمله ۲- آخرین کلمه‌ی جداشده

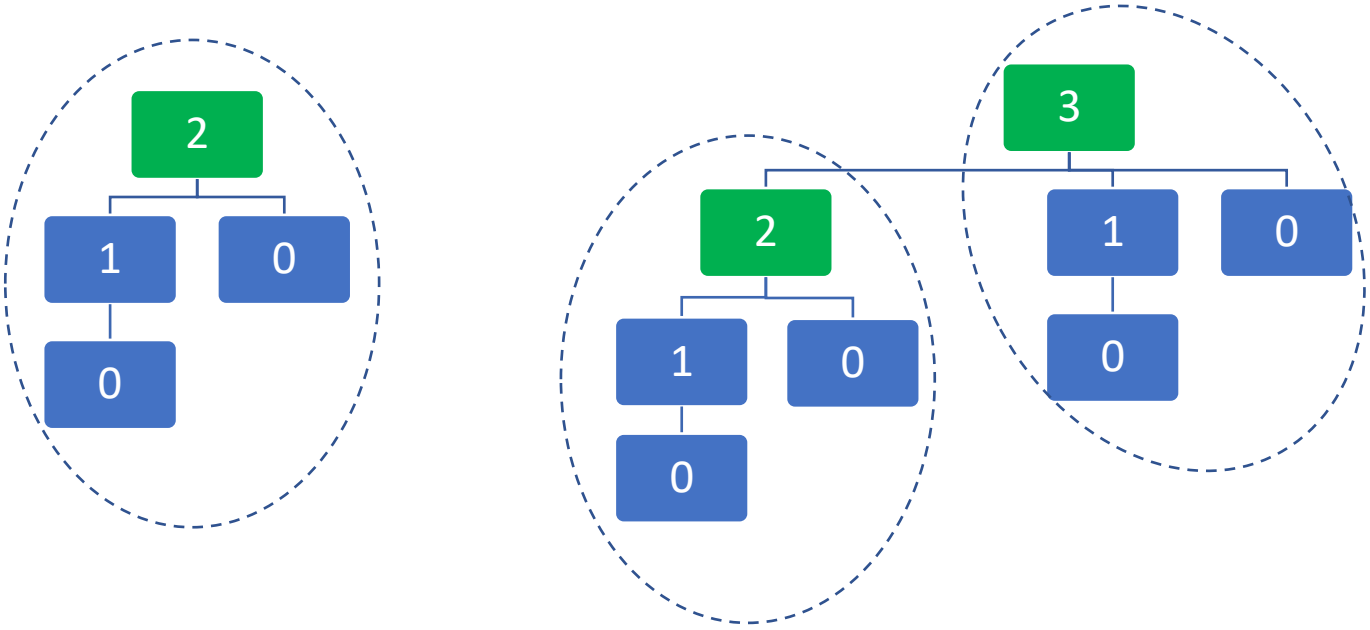
خروجی: اگر جمله قابل جداسازی باشد: جداشده‌ی جمله‌ی ورودی با فاصله. در غیر اینصورت: تهی (None)

حال به توضیح الگوریتم می‌پردازیم. جمله‌ی ورودی را پس از تبدیل تمام حروف آن به حروف کوچک، به تابع valid می‌دهیم:

۱. اگر جمله خالی شده باشد یعنی جمله parse شده است و هر فراخوانی، کلمه‌ی جداشده‌ی خود را به نتیجه‌ی returnشده از فراخوانی‌های قبلی می‌چسباند تا در نهایت کل جمله در فراخوانی اول return شود. اگر جمله هنوز خالی نباشد به مرحله‌ی ۲ می‌رویم.
۲. زیررشته‌ها با طول‌های ۱ تا طول جمله را از ابتدای جمله در نظر می‌گیریم. اگر این زیررشته در لغت‌نامه باشد، تابع valid را با زیرجمله و کلمه‌ی پیداشده فراخوانی می‌کنیم و به مرحله‌ی ۱ می‌رویم. لازم به ذکر است ترتیب در نظر گرفتن طول زیررشته‌ها به این ترتیب است: ۲، ۳، ...، n و ۱. دلیل این کار این است که کلماتی با طول بیشتر از یک در حالاتی که هم‌پوشانی وجود دارد ارجحیت داشته باشند. مثلاً کلمه‌ی away بدون در نظر گرفتن این ترتیب به کلمات a و way جدا خواهد شد اما با در نظر گرفتن ترتیب گفته‌شده به‌درستی away تشخیص داده می‌شود. اگر هیچ کدام از این زیررشته‌ها در لغت‌نامه موجود نباشند، عقب‌گرد می‌کنیم و زیررشته‌ی دیگری را در فراخوانی قبلی در نظر می‌گیریم.
۳. اگر تمام درخت جستجو پیمایش شد و به نتیجه‌ای نرسیدیم، بدین معنی است که جمله‌ی داده‌شده با لغت‌نامه‌ی مذکور قابل parse نیست و این موضوع را در terminal به اطلاع کاربر می‌رسانیم.

پیچیدگی زمانی: برای بررسی پیچیدگی زمانی این الگوریتم از دو منظر به آن نگاه می‌کنیم:

۱- ابتدا بدترین حالت را در نظر می‌گیریم که تمام درخت پیمایش شود. برای n=3 و n=2 درخت مورد پیمایش را رسم می‌کنیم. اعداد نوشته‌شده درون هر گره برابر طول جمله‌ی ورودی فراخوانی کنونی از تابع است.



همان‌طور که مشاهده می‌کنیم ساختار درخت مربوط به n = 2 دو بار در درخت n = 3 تکرار شده است. در واقع با هر افزایش n، تعداد گره‌های درخت دو برابر می‌شود. حال اگر basic operationها را ۱- «تولید گره (= پارتیشن‌بندی جمله)» و ۲- «بررسی وجود بخش اول در لغت‌نامه» در نظر بگیریم، و فرض کنیم عملیات شماره‌ی ۱ O(1) است و چون جستجو در Set نیز O(1) است پس هر گره در یک ضریب ثابت ضرب می‌شود. بنابراین چون $T(0) = 1$ است پس خواهیم داشت $T(n) = 2T(n-1) = 2^n$. پس **پیچیدگی زمانی این الگوریتم در بدترین حالت 2^n است.**

۲- روش دیگری نیز برای محاسبه‌ی پیچیدگی زمانی می‌توان به کار برد تا عبارت دقیق‌تری به‌طور احتمالاتی به دست آید. در این روش برای corpus مورد نظر که جملات از آن‌جا انتخاب می‌شوند (با فرض وجود همه‌ی کلمات آن در لغت‌نامه)، توزیع تعداد حروف کلمات را به دست می‌آوریم. در فضای واقعی، در هر سطح از درخت تمام فرزندان معتبر نیستند. بلکه تا جایی که یک کلمه‌ی معتبر پیدا کنیم در آن سطح پیش می‌رویم و سپس فرزندان آن را بررسی می‌کنیم. بنابراین در هر سطح، با توجه به توزیع به‌دست‌آمده یک عدد متناظر با تعداد حروف کلمه‌ی کنونی به‌صورت تصادفی انتخاب می‌کنیم که نشان‌دهنده‌ی تعداد گره‌هایی‌ست که قبل از رفتن به سطح بعدی باید مشاهده گردند. البته این تحلیل تخمینی پایین‌تر از درخت واقعی را مشخص می‌کند چون مثلاً اگر کلمه‌ی بعدی bedroom باشد، قبل از این که به bedroom برسیم، کلمه‌ی معتبر bed را مشاهده خواهیم کرد و زیردرخت متناظر با آن ابتدا پیمایش خواهد شد که در این تحلیل لحاظ نمی‌شود. اما به هر حال کران بالای 2^n برای هر درختی صادق است و ما با این روش می‌توانیم پیچیدگی زمانی میانگین را با دقت مناسبی به دست آوریم.

شبیه‌سازی: برای شبیه‌سازی تعدادی جمله‌ی ورودی و خروجی متناظر با آن را در زیر مشاهده می‌کنیم. اگر کلمه‌ای در لغت‌نامه موجود نباشد، همان‌طور که گفته شد، این موضوع به اطلاع کاربر می‌رسد:

```
26 if __name__ == '__main__':
27     sentence = "IHOPETHISISOURLASTPROJECTINTHISCOURSE"
28     print(parse(sentence))
29 
30     sentence = 'WEAREVERYTIREDRIGHTNOW'
31     print(parse(sentence))
32 
33     sentence = 'SLEEPISINTHEDICTIONARY'
34     print(parse(sentence))
35 
36     sentence = 'SLEEPYISNOTINTHEDICTIONARY' # Sleepy is not in the dictionary
37     print(parse(sentence))
38 
```

```
i hope this is our last project in this course
we are very tired right now
sleep is in the dictionary
This sentence can't be parsed :(
```