به نام خدا

دانشکده‌ی مهندسی برق و کامپیوتر دانشگاه تهران

طراحی و تحلیل الگوریتم‌ها، نیمسال دوم سال تحصیلی ۹۷-۹۶

پاسخ تمرین شماره ۳ (الگوریتم‌های حریصانه)

۱. به چپ‌ترین آدم چپ ترین صندلی را می‌دهیم. مساله‌ای که باقی می‌ماند مانند مساله اول است فقط یک آدم و یک صندلی حذف می‌شود. پس دوباره چپ‌ترین آدم را پیدا می‌کنیم و به او چپ‌ترین صندلی را می‌دهیم. (عملا داریم صندلی‌ها و آدم‌ها را بر اساس مکانشان مرتب می‌کنیم و به آدم iام در لیست مرتب شده، صندلی iام را می‌دهیم.)

اثبات انتخاب:
لیست آدم‌ها و صندلی‌ها را مرتب شده بر اساس مکان در نظر بگیرید.
فرض می‌کنیم راه حلی مانند S وجود دارد که به چپ‌ترین آدم، چپ‌ترین صندلی را نداده‌است. ثابت می‌کنیم با دادن چپ‌ترین صندلی به چپ‌ترین آدم، این راه حل بدتر نمی‌شود.
در S، به چپ‌ترین آدم صندلی kام داده‌شده‌است (k>1). پس برای چپ‌ترین صندلی، یک نفر مثل نفر iام انتخاب شده‌است.
ثابت می‌کنیم که اگر برای نفرِ اول، صندلی اول و برای نفر iام، صندلی kام انتخاب شود، این راه حل از نظر مسافت بدتر نمی‌شود.
این دو صندلی و دو آدم، با دانستن این که نفر اول چپ‌ترین آدم و صندلی اول چپ‌ترین صندلی است، به شش حالت زیر می‌توانند نسبت به هم قرار داشته باشند در همه‌ی حالت ها می‌بینیم که دادنِ صندلی اول به آدم اول و صندلی kام به آدم iام کل مسافت طی شده را یا تغییر نمی‌دهد یا کم می‌کند:

نحوه‌ی قرار گرفتن	مجموع فاصله‌ی طی شده در S	مجموع فاصله‌ی طی شده بعد از تغییر
P1 (x1) Pi (x2) C1 (x3) Ck	x1+x2+x3+ x2	x1+x2+ x2+x3
P1 (x1) C1 (x2) Pi (x3) Ck	x1+x2+x3+ x2	x1+ x3
P1 (x1) C1 (x2) Ck (x3) Pj	x1+x2+ x3+x2	x1+ x3
C1 (x1) Ck (x2) P1 (x3) Pi	متناظر حالت ۱ است	
C1 (x1) Pi (x2) C1 (x3) Ck	متناظر حالت ۲ است	
C1 (x1) Pi (x2) C1 (x3) Ck	متناظر حالت ۳ است	

اگر بخواهیم بیشترین فاصله‌ی طی شده کمترین مقدار ممکن باشد هم دقیقا می‌شود از استدلال قبل استفاده کرد. فقط باید نشان دهیم بیشترین مسافت طی شده بعد از تغییر، یا کمتر می‌شود یا همان قدر می‌ماند که این موضوع از روی جدول بالا قابل استدلال است.

۲. یک گراف وزن دار G' از روی گراف داده شده تشکیل می‌دهیم به این ترتیب که برای هر یال در G یال متناظر آن را با وزن صفر در G' قرار می‌دهیم و اگر یالی مانند (u, v) در G وجود داشت و یال (v, u) وجود نداشت، یال (v, u) را در G' با وزن یک اضافه می‌کنیم (این یال‌ها معادل یال‌هایی هستند که باید معکوس شوند). این کار اوردر $O(V+E)$ زمان می‌برد.

حالا کافی است وزن کوتاه‌ترین مسیر از s به t را در G' پیدا کنیم. چون در کوتاه‌ترین مسیر، از حداقل تعداد ممکن یال با وزن یک استفاده شده پس حداقل تعداد ممکن یالی که باید معکوس شوند استفاده شده است. (توجه کنید که ممکن نیست از یک یال و معکوسش همزمان در مسیر استفاده شده باشد).
برای پیدا کردن کوتاه‌ترین مسیر در گراف G'، از الگوریتم دایسترا استفاده می‌کنیم که اوردر زمانی آن $O(V^2)$است.
پس اوردر الگوریتم در کل $O(V^2)$ است.

۳. از برنامه‌نویسی پویا استفاده می‌کنیم.
برای راحت‌تر شدن توضیح، مساله را این شکلی تعریف می‌کنیم که دوتا مورچه داریم که می‌خواهند با طی کردن دو مسیر متفاوت از چپ‌ترین خوراکی به راست‌ترین خوراکی برسند و در مجموع همه‌ی خوراکی‌ها را خورده‌باشند و هر خوراکی را دقیقا یکی از آن‌ها خورده باشد. (این مساله معادل مساله‌ی اول است چون مثل این است که یکی از مورچه‌ها مسیر برگشت را طی می‌کند.) می‌خواهیم مجموع مسافت طی شده توسط دو مورچه کم‌ترین حالت ممکن شود.
$P(i,j)$ را این شکلی تعریف می‌کنیم:
کوتاه‌ترین مسافتی که برای خوردن خوراکی‌های مانده توسط دو مورچه طی می‌شود به شرطی که مورچه‌ی اول روی خوراکی iام باشد و مورچه‌ی دوم روی خوراکی jام باشد. و بین i و j و قبل از آن‌ها خوراکی خورده نشده‌ای وجود نداشته‌باشد*. بدون کم شدن از کل مساله فرض می‌کنیم همیشه i>j است (برای یک کسی که در لحظه‌ای که دو مورچه در خانه‌ی iو j اند به مساله نگاه کند موقعیت مورچه‌ها مهم است نه این که خوراکی‌هایی که خورده شده اند را کدامشان خورده).
خوراکی kام را اولین خوراکی بعد از هر دوی i و j در نظر بگیرید(k=j+1). داریم:
(طول خط راست بین خوراکی xام و خوراکی yام را با (x,y) نشان داده‌ایم)

$$P(i,j) = min\{P(i,k) + (j,k), P(j,k) + (i,k)\}$$

برای پیاده‌سازی این الگوریتم یک آرایه‌ی دوبعدی n در n را پر می‌کنیم (نصف آرایه را چون فرض کردیم همیشه i>j).
حالت‌های اولیه را وقتی می‌گیریم که k مساوی n باشد در این صورت $p(i,j) = (i,k) + (j,k)$. برای پر کردن هر خانه (i,j) باید $(i,j+1)$ و $(j,j+1)$ پر شده‌باشند. پس باید آرایه را از ردیف آخر به اول پر کنیم.
پیچیدگی این الگوریتم $O(n^2)$ است.

* شاید به نظر بیاید که این فرض باعث می‌شود که تعدادی حالت از دست برود. اما این طور نیست. فرض کنید به یک روشی همه‌ی خوراکی ها خورده شده باشد. راه ما حتما این روش را بررسی کرده چون می‌توانیم این روش را به این صورت اجرا کنیم که دو مورچه از خانه‌ی اول شروع کنند و روی خوراکی ها حرکت کنیم و در هر لحظه مورچه‌ای که خوراکی بعدی را خورده حرکت کند و به خوراکی بعدی برسد. پس بین دو مورچه در هیچ لحظه‌ای خوراکی خورده نشده‌ای وجود ندارد.

۴. پروژه‌ای که کمترین تعداد روز نیاز دارد را اول انجام می‌دهیم. و در روزی که پروژه تمام شود یک سری پروژه داریم که تعدادی روز زمان لازم دارند. پس مساله مثل مساله اول است.
اثبات انتخاب:
فرض کنید راه حل S وجود دارد که پروژه i را اول انجام داده. پس از روز 1 تا di این پروژه انجام شده و پروژه‌ی با کمترین روز لازم (پروژه‌ی 1)، در روز d تا d+d1 انجام شده. به راحتی قابل مشاهده است که اگر جای پروژه‌ی i و پروژه‌ی 1 را عوض کنیم. پروژه‌های بین این دو، زودتر تحویل داده می‌شوند و پروژه‌های بعد از پروژه‌ی 1 تغییری نمی‌کنند. پس راه حل بهتر می‌شود.
برای حالت بعد هم کافی است پروژه با کمترین روز مورد نیاز انتخاب شود. هر زمان که پروژه‌ی در حال انجام تمام شد یا پروژه‌ی جدیدی به ما داده شد، این انتخاب را دوباره انجام می‌دهیم. (فرض کنید که پروژه‌ای که در حال انجام آن بودیم d روز دیگر زمان لازم دارد، این پروژه مانند یک پروژه‌ای می‌شود که از حالا d روز زمان لازم دارد) پس در هر تصمیم یک سری پروژه داریم که هر کدام تعدادی روز زمان لازم دارند. پس مساله‌ی باقی‌مانده مانند مساله اول است.
اثبات درستی این انتخاب شباهت زیادی به حالت قبل دارد.

۵. با استفاده از برنامه‌نویسی پویا این سوال را حل می‌کنیم.
برای داشتن بیشترین تعداد Aی ممکن، بعد از اینکه تعدادی A چاپ کردیم، در هر لحظه یا با خرج کردن سه کلیک، select-all، copy و paste می‌کنیم یا همان چیزی که قبلا کپی شده را فقط با خرج کردن یک کلیک paste می‌کنیم.
(یعنی از اولین select-all و copy که انجام بدهیم، چون با استفاده از paste تعدادی A چاپ می‌شود دیگر استفاده از کلید A معنی ندارد)
برای nهای کمتر از ۷ بیشترین تعداد A ممکن (A(n)) همان n است. برای Nهای بزرگتر مساوی با ۷ داریم:

$$A(n) = MAX_{i=1...n-3}\{A(i) \times (n-i-2)\}$$

در واقع i+1 آخرین جایی است که select-all و copy کردیم و بعد از آن فقط paste کردیم.
پیچیدگی این الگوریتم $O(n^2)$ است.

۶.
دو اشاره گر p و t به اولین دزد و اولین پلیس تعریف می‌کنیم. اگر دزد در محدوده‌ی پلیس بود، دزد را به پلیس می‌دهیم و هر دو اشاره گر را آپدیت می‌کنیم تا به اولین دزد و پلیس بعدی اشاره کنند. وگرنه، اشاره‌گر کوچکتر را آپدیت می‌کنیم تا به دزد یا پلیس بعدی اشاره کند و دوباره همین کار را می‌کنیم.
اثبات انتخاب:
همه‌ی مچ‌های انجام شده را به ترتیب خانه‌ی پلیس آن مرتب می‌کنیم. ثابت می‌کنیم در هر راه حل بهینه‌ای می‌شود اولین مچ را با اولین مچ راه حل ما جایگزین کرد به طوری که کل تعداد مچ‌ها کمتر نشود.
اولین پلیسی که راه حل ما با یک دزد مچ کرده را p_1 و دزدی که با آن مچ شده را t_1 در نظر بگیرید.
فرض می‌کنیم یک راه حل بهینه‌ی دیگر مانند S وجود دارد. اولین پلیسی را که این راه حل مچ کرده‌است p'_1 و دزدی را که با آن مچ شده‌است t'_1 در نظر بگیرید.
چون الگوریتم ما اولین پلیس ممکن را با اولین دزد ممکن مچ می‌کند، پس پلیس‌ها و دزدهای قبل p_1 و t_1 هیچ مچ ممکنی برایشان وجود ندارد(وگرنه الگوریتم ما مچ می‌کرد) پس حتما $p'_1 \geq p_1$ و $t'_1 \geq t_1$ است.
اگر $p'_1 = p_1$ باشد،
اگر دزد t_1 در راه حل S با هیچ پلیسی مچ نشده باشد، t'_1 را از p'_1 می‌گیریم و t_1 را به او می‌دهیم.
اگرنه فرض کنید دزد t_1 در راه حل S با پلیس p' مچ شده باشد. پس حتما $p' > p'_1$ (چون p'_1 اولین مچ ممکن است).
در راه حل S، p' با t_1 و p'_1 با t'_1 مچ شده. این دو تا را جابه‌جا می‌کنیم. یعنی p'_1 را با t_1 و به p' را با t'_1 مچ می‌کنیم.
مچ شدن p'_1 و t_1 که واضح است با محدودیت مساله منافات ندارد (چون $p'_1 = p_1$ و در راه حل ما p_1 به t_1 مچ شده‌است پس اختلاف آن‌ها کمتر از k است). حالا باید ثابت کنیم که اختلاف p' و t'_1 هم کمتر از k است. اگر t'_1 از p' کمتر باشد از آنجایی که $t'_1 \geq t_1$ و اختلاف p' و t_1 کمتر از k است (چون در راه حل S با هم مچ شده‌اند)، اختلاف t'_1 از p' هم کمتر از k می‌شود. اگر اگر t'_1 از p' بزرگتر باشد، از آنجایی که $p' > p'_1$ و اختلاف t'_1 و p'_1 کمتر از k است (در راه حل S مچ شده‌اند)، اختلاف اگر t'_1 از p' هم کمتر از k می‌شود.
اگر $p'_1 > p_1$ باشد،
پس p_1 با کسی مچ نشده‌است. دزد t_1 را اگر با کسی مچ شده بود از او می‌گیریم و به p_1 می‌دهیم که این تعداد مچ‌ها را تغییر نمی‌دهد. اگر t_1 با کسی مچ نشده بود او را به p_1 می‌دهیم که در این صورت تعداد مچ‌ها یکی بیشتر می‌شود.

۷.
آ) مجموع تعداد خانه‌های بی‌استفاده برابر است با مجموع کل خانه‌ها منهای آن‌هایی که استفاده شده‌اند (مجموع خانه‌هایی که برای کلمه‌ها استفاده کردیم به‌اضافه‌ی خانه‌هایی که برای فاصله‌ی بین کلمه‌ها استفاده کردیم)
پس داریم:

$$\Sigma_{i=1}^{m} r_i = m \times k - [\Sigma_{i=1}^{n} l_i + (n-m)]$$

پس قشنگترین حروف‌چینی باید تعداد خطها(m) را کمترین حالت ممکن کند.
برای هر کلمه دو انتخاب داریم یکی این که به خط بعدی برویم و یکی این که در همان خط اگر جا بود بگذاریمش.
برای این کار هر کلمه را در خط کنونی قرار می‌دهیم. اگر این خط جا نداشت به خط بعدی می‌رویم.

اثبات بهینه بودن انتخاب:
در راه حلی مانند S فرض کنید اولین جایی که انتخاب با انتخاب ما متفاوت است کلمه‌ی iام باشد. ثابت می‌کنیم می‌شود انتخاب iام را با انتخاب راه حل ما عوض کرد طوری که راه حل بدتر نشود.
اگر انتخاب S این باشد که به خط بعدی برویم و انتخاب ما این باشد که در خط کنونی کلمه را بگذاریم، حتما خط کنونی به تعداد حروف این کلمه فضای خالی داشته(که راه حل کلمه را ما در این خط قرار داده) پس این کلمه می‌شود به خط کنونی منتقل شود و در آخر خط بعدی به ازای تعداد حروف این کلمه فضای خالی گذاشته‌شود. که این کار کل فضای خالی را تغییر نمی‌دهد.
اگر انتخاب S این باشد که در همین خط بمانیم حتما این خط به تعداد کافی جا دارد پس حتما انتخاب ما هم همین خواهد بود.

ب) برای حل این بخش از برنامه‌نویسی پویا استفاده می‌کنیم. $P(i)$ را تعریف می‌کنیم مقدار $\Sigma_{i=1}^{m} r_i^2$ برای بهترین حروف‌چینی کلمه‌های اول تا iام. فرض کنید در خط آخر حداکثر X کلمه می‌توانیم داشته‌باشیم (حداکثر X کلمه‌ی آخر متن (یعنی کلمه‌های i-X+1ام تا i ام) در یک خط جا می‌شوند) حالت‌های مختلف این است که ۱، ۲، ۳،... ،X کلمه در خط آخر باشند. پس داریم:

$$P(i) = min_{x=1...X}\{(k - \Sigma_{n=i-x+1}^{i}(l_n + 1))^2 + P(i - x)\}$$

هزینه‌ی زمانی این الگوریتم $O(n^2)$ است.