1. ابتدا درخت را روی یک راس دلخواه مثل r ریشه دار میکنیم. حال به ازای هر راس مقدار white[v] را تعریف میکنیم تعداد حالاتی که می توان تعدادی یال از زیر درخت v حذف کرد طوری که مولفه ای که v در آن قرار میگیرد هیچ راس سیاهی نداشته باشد و بقیه مولفه ها دقیقا یک راس سیاه داشته باشند. به طور مشابه black[v] را تعریف می کنیم تعداد حالاتی که می توان تعدادی یال از زیردرخت v حذف کرد طوری که تمام مولفه ها دقیقا یک راس سیاه داشته باشند. برای محاسبه این دو مقدار یک پیمایش عمق-اول در درخت انجام میدهیم و به شکل زیر عمل میکنیم:

فرزندان v را $\{c_1, \dots, c_2\}$ مینامیم T_i را تعریف میکنیم درختی که از زیر درخت v با حذف تمام زیر درخت های با ریشه ها $\{c_i : j > i\}$ بدست می آید. به عبارت دیگر T_i درختی است که از زیر درخت v با در نظر گرفتن فقط i فرزند اول v بدست می آید. و T_k برابر با خود زیر درخت v است. حال مقدار whiteUpto[i] را تعریف میکنیم تعداد روش های حذف تعدادی یال از T_i به گونه ای که مولفه ای که v در آن قرار دارد راس سیاه نداشته باشدو بقیه ی مولفه ها دقیقا یک راس سیاه داشته باشند. به طور مشابه blackUptp[i] را تعریف میکنیم روش های حذف تعدادی یا از T_i به گونه ای که تمام مولفه ها دقیقا یک راس سیاه داشته باشند.

واضح است که white[v[= whiteUpto[k] و black[v] = blackUpto[k]. برای محاسبه این مقادیر به این شکل عمل میکنیم که برای یک راس سفید blackUpto[0] = 0 و whiteUpto[0] = 1 و برای یک راس سیاه blackUpto[0] = 1 و whiteUpto[0] = 0.

برای محاسبه whiteUpto[i] داریم:

برای اینکه راس v در یک مولفه سفید قرار بگیرد باید از T_{i-1} به گونه ای یال حذف شده باشد که v در یک مولفه ی سفید باشد، یعنی whiteUpto[i-1] حالت، سپس برای زیر درخت با ریشه c_i اگر یال بین v و c_i حذف نشود، c_i باید در یک مولفه سفید باشد یعنی white[c_i] حالت. اگر یال بین v و c_i باید در یک مولفه سیاه قرار داشته باشد یعنی black[c_i] حالت. سپس مقدار whiteUpto[i] از رابطه زیر بدست می آید:

whiteUpto[i] = whiteUpto[i-1] * (white[c_i] + black[c_i])

برای محاسبه blackUpto[i] داریم: برای اینکه راس v در یک مولفه سیاه قرار بگیرد دو حالت وجود دارد: با اینکه یال های T_i به گونه ای حذف شده اند که v در یک مولفه سفید قرار دارد یعنی whiteUpto[i-1] حالت که در این صورت باید بین v , c_i حذف نشود و c_i در یک مولفه ی سیاه باشد. یعنی black[c_i] حالت. یا اینکه یال های T_{i-1} به گونه ای حذف شده اند که v در یک مولفه سیاه قرار دارد یعنی blackUpto[i-1] حالت در این صورت اگر یال بین v و c_i حذف نشود، c_i باید در یک مولفه ی سفید باشد یعنی white[c_i] حالت. اگر یال بین v و c_i حذف نشود، c_i باید در یک مولفه سیاه باشد یعنی black[c_i] حالت. بنابراین مقدار blackUpto[i] از رابطه ی زیر بدست می آید:

blackUpto[i] = whiteUpto[i-1] * black[c_i] + blackUpto[i-1] * white[c_i] + blackUpto[i-1] * black[c_i]

واضح است که پاسخ مسئله در black[r] قرار میگیرد.

بدست آوردن مقادیر white[v] و black[v] در هر راس O(deg(v)) زمان میبرد پس زمان اجرای الگوریتم O(n) است.

2. به چپ ترین کامیون، چپ ترین پارکینگ را میدهیم. مساله ای که باقی میماند مانند مساله ی اول است فقط یک کامیون و یک پارکینگ حذف میشود. پس دوباره چپ ترین کامیون را پیدا میکنیم و به او چپ ترین پارکینگ را میدهیم.
اثبات انتخاب:
لیست کامیون ها و پارکینگ ها را مرتب شده بر اسا مکان در نظر بگیرید.
فرض میکنیم راه حلی مانند S وجود دارد که به چپ ترین کامیون، چپ ترین پارکینگ را نسبت نداده اسن. ثابت میکنیم با دادن چپ ترین پارکینگ به چپ ترین کامیون، این راه حل بدتر نمی شود.
در S، به چپ ترین کامیون، پارکینگ kام داده شده است. پس برای چپ ترین پارکینگ، یک کامیون مثل کامیون iام انتخاب شده است. ثابت می کنیم که اگر برای کامیون اول، پارکینگ اول و برای کامیون iام، پارکینگ kام انتخاب شود، این راه حل از نظر مسافت بدتر نمیشود.
این دو صندلی و دو آدم، با دانستن این که کامیون اول چپ ترین کامیون و پارکینگ اول چپ ترین پارکینگ است، به شش حالت زیر می توانند نسبت به هم قرار داشته باشند در همه ی حالات میبینیم که دادن پارکینگ اول به کامیون اول و پارکینگ kام به کامیون iام کل مسافت طی شده را یا تغییر نمی دهد یا کم می کند:

نحوه ی قرار گرفتن	مجموع فاصله ی طی شده در S	مجموع فاصله ی طی شده بعد از تغییر
P1(x1) Pi(x2) C1(x3) Ck	x1 + x2 + x3 + x2	x1 + x2+ x2 + x3
P1(x1) C2(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 1 است	
C1(x1) Pi(x2) C1(x3) Ck	متناظر حالت 2 است	
C1(x1) Pi(x2) C1(x3) Ck	متناظر حالت 3 است	

اگر بخواهیم بیشترین فاصله ی طی شده کمترین مقدار ممکن باشد هم دقیقا می شود و میتوان استدلال قبل استفاده کرد. فقط باید نشان دهیم بیشترین مسافت طی شده بعد از تغییر، یا کمتر میشود یا همان قدر می ماند که این موضوع از روی جدول بالا قابل استدلال می باشد.

3. اگر طول دو رشته رتا به ترتیب a, b در نظر بگیریم، کاراکتر آخر دو String یکسان بودند، آنرا کنار گذاشته و به سراغ کاراکتر قبلی اش میرویم و عملیات را برای دو String با طول های a-1 و b-1 انجام میدهیم. اگر یکسان نبودند، ما هر سه عملیات را برای آخرین کاراکتر

string اول در نظر میگیریم و minimum آنهارا باز میگردانیم. dp[i][j] را min هزینه برای انجام این کار تا کاراکتر iام رشته ی اول و تا کاراکتر jام رشته ی دوم تعریف میکنیم.

```
dp[i][j] = 1 + min(dp[i][j-1],  // Insert
                   dp[i-1][j],  // Remove
                   dp[i-1][j-1]); // Replace
```

4. دو اشاره گر h وk را با اولین هویج و خرگوش تعریف میکنیم. اگر هویج در محدوده ی خرگوش بود، هویج را به خرگوش می دهیم و هر دو اشاره گر را آپدیت میکنیم تا به اولین هویج و خرگوش بعدی اشاره کنند. وگرنه، اشاره گر کوچکتر را آپدیت میکنیم تا به هویج یا خرگوش بعدی اشاره کنند و دوباره همین کار را میکنیم.
اثبات انتخاب:
همه ی مچ های انجام شده را به ترتیب خانه ی پلیس آن مرتب میکنیم. ثابت میکنیم در هر راه حل بهینه ای می شود. اولین مچ را با اولیت مچ راه حل ما جایگزین کرد به طوری که کل تعداد مچ ها کمتر نشود.
اولین خرگوشی که راه حل ما را با یک هویج مچ کرده را K_1 و هویجی که با آن مچ شده را H_1 در نظر میگیریم.
فرض میکنیم راه حل بهینه ای دیگر مانند S وجود دارد. اولین خرگوشی را که این راه حل را مچ کرده است ${K_1}'$ و هویجی که با آن مچ شده را ${H_1}'$ در نظر میگیریم.
چون الگوریتم ما اولین خرگوش ممکن را با اولین هویج مچ میکنئ، پس خرگوش ها و هویج های قبل K_1 و H_1 هیچ مچ ممکنی برایشان وجود ندارد(وگرنه الگوریتم ما مچ میکرد) پس حتما ${K_1}' > K_1$ و ${H_1}' > H_1$ است.
اگر ${K_1}' = K_1$ باشد، اگر هویج H_1 در راه حل S با هیچ هویجی مچ نشده باشد، ${H_1}'$ را از ${K_1}'$ میگیریم و H_1 را به او میدهیم.
اگر نه فرض کنید هویج H_1 در راه حل S با خرگوش K' مچ شده باشد. پس حتما $K' > {K_1}'$ (چون${K_1}'$ اولین مچ ممکن است).
در راه حل S، K' با H_1 و ${K_1}'$ با ${H_1}'$ مچ شده. این دو تا را جالجا میکنیم. یعنی ${K_1}'$ را با H_1 و به K' را با ${H_1}'$ مچ میکنیم.
مچ شدن ${K_1}'$ با H_1 که واضح است با محدودیت مسوله منافات ندارد (چون ${K_1}' = K_1$ و در راه حل ما K_1 به H_1 مچ شده است پس اختلاف آنها کمتر از k است). حال باید ثابت کنیم که اختلاف K' و ${H_1}'$ هم کمتر از k است. اگر ${H_1}'$ از K' کمتر باشد از آنجایی که ${K_1}' \geq K_1$ و اختلاف K' و H_1 کمتر از k است (چون در راه حل های S با هم مچ شده اند)، اختلاف ${H_1}'$ از K' هم کمتر از k می شود. اگر ${H_1}'$ از K' بزرگتر باشد، ار آنجایی که $K' > {K_1}'$ و اختلاف ${H_1}'$ و K' کمتر از k است (در راه حل S مچ شده اند)، اختلاف ${H_1}'$ از K' هم کمتر از k می شود.

اگر $K_1{}' > K_1$ باشد، پس K_1 با کسی مچ نشده است. هویج H_1 را اگز با کسی مچ شده بود از او میگیریم و به K_1 میدهیم که این تعداد مچ ها را تغییر نمیدهد. اگر H_1 با کسی مچ نشده بود او را به K_1 میدهیم که در این صورت تعداد مچ ها یکی بیشتر می شود.

5. ابتدا همه ی بازه ها را بر اساس زمان مرتب میکنیم و در یک آرایه میریزیم. برای انتخاب اولین مراقب (بازه) باید بازه ای را انتخاب کنیم که شروع آن a باشد. پس از اول آرایه حرکت میکنیم و از بین تمام بازه هایی که ابتدای آن ها a است، آن بازه ای بیشترین اندازه را دارد انتخاب میکنیم. (مثلا $[a, F_0)$) و باقی را حذف میکنیم. برای بازه ی دوم دوباره به حرکت ادامه میدهیم و بزرگترین بازه ای که شروع آن ها از F_0 بزرگتر یا مساوی است را انتخاب میکنیم. این روند را تا جایی ادامه می دهیم که بازه ای با پایان بزرگتر مساوی b انتخاب میشود.
برای اثبات درستی الگوریتم از برهان خلف استفاده میکنیم:
فرض میکنیم که پاسخ تولید شده توسط الگوریتم بهینه نیست. هر پاسخ به مسئله را توسط یک لیسا مرتب صعودی طبق زمان شروع از بازه های موجود در آن پاسخ نشان میدهیم، به عنوان نثال پاسخ تولید شده توسط الگوریتم را توسط $S = < I_1, \ldots, I_k >$ نشان میدهیم. دقت کنید که با توجه به نحوه ی ساخته شدن S لیست باز ها مرتب صعودی طبق زمان پایان هم هست. حال شبیه ترین پاسخ به S را $S' = < I_1{}', \ldots, I_k{}' >$ تعریف میکنیم به گونه ای که کوچکترین i که برای آن $I_i{}' \neq I_i$ است بیشینه باشد. با توجه به اینکه S یک پاسخ بهینه است میدانیم که $< I_1{}', \ldots, I_m{}' >$مرتب صعودی طبق زمان پایان هم هست چرا که اگر برای یک j داشته باشیم $I_{I_{j-1}{}'} \geq I_{I_j{}'}$ آنگاه با حذف بازه I_j به یک پاسخ معتبر با تعداد کمتری بازه میرسیم. حال پاسخ S'' را با جایگزین کردن $I_i{}'$ توسط I_i در S' میسازیم. میدانیم که S'' یک پاسخ معتبر است زیرا $S_{I_i{}'} \leq S_{I_{i-1}{}'}$ و همچنین با توجه به نحوه ی انتخاب I_i حتما داریم $F_{I_i} \geq F_{I_i{}'} \geq S_{I_{i+1}{}'}$، از طرف دیگر S'' یک پاسخ بهینه است چرا که تعداد بازه هایش با S' مساویست اما با این فرض S' شبیه ترین پاسخ بهینه به S است در تناقض است.
در این الگوریتم ابتدا *O(n log n)* هزینه برای مرتب سازی و یک پیمایش آرایه *(n)* نیاز است. پس هزینه کل برابر است با *O(n log n)*

6. مسئله ی پرانتز ها را که این گونه تعریف شده بود :
C_n تعداد عبارات حاوی *n* جفت پرانتزهایی را که به طور صحیح مطابقت دارند شمارش می کند : مثلا برای *n = 3* داریم:
((())) ()(()) ()()() (())() (()())
حال این مسئله معادل است با اینکه *Cn* تعداد درختهای باینری کامل با *n + 1* برگ است: