دانشکده‌ی علوم ریاضی

آنالیز الگوریتم‌ها ۲۰ اسفند ۹۱

جلسه‌ی ۱۰: الگوریتم حریصانه

مدرّس: دکتر شهرام خزائی *نگارنده: حمید پورربیع رودسری*

۱ مقدمه

برای موفقیت در بازی‌هایی مانند شطرنج باید تا حد ممکن آینده را پیش‌بینی کرد و برای هر حرکت عواقب ناشی از آن را در نظر گرفت. در این بازی شکست دادن فردی که فقط به برتری لحظه‌ای فکر می‌کند بسیار آسان است. اما در بازی‌هایی مثل «کلمات به هم ریخته»[۱] می توان بدون توجه به عواقب آینده و تنها با توجه به شرایط فعلی تصمیمی مناسب اتخاذ کرد.
این نوع تصمیم‌گیری که در آن عواقب آینده‌ی تصمیم در نظر گرفته نمی‌شود، بسیار ساده و مناسب به نظر می‌رسد و به همین دلیل تبدیل به یک روش جذاب برای طراحی الگوریتم‌ها شده است و با نام الگوریتم حریصانه[۲] شناخته می‌شود. الگوریتم‌های حریصانه سعی می‌کنند مرحله به مرحله به جواب برسند و در هر مرحله جوابی را انتخاب کنند که بهترین و واضح‌ترین نتیجه‌ی ممکن را ارائه می‌کند. اگرچه ممکن است این نوع عملکرد در بعضی مسائل بسیار زمان‌گیر و نامناسب باشد اما مسائل بسیاری وجود دارند که این روش الگوریتمی بهینه برای حل آن‌ها پیشنهاد می‌کند. در اینجا دو مثال از شرایطی را بررسی می‌کنیم که می‌توان با استفاده از این روش به الگوریتمی بهینه رسید.

۲ کد هافمن

برای معرفی کد هافمن[۳] ابتدا کد باینری[۴] و بعضی ویژگی‌های آن را بررسی می‌کنیم:

[۱] Scrabble
[۲] greedy algorithms
[۳] Huffman code
[۴] binary code

۱.۲ کد باینری و ویژگی‌های آن

تعریف ۱ *یک کد باینری برای الفبای Σ تابعی مانند* $c : \Sigma \longrightarrow \{۰, ۱\}^*$ *است که هر حرف a از الفبا را به رشته‌ی باینری $c(a)$ می‌نگارد.*

نکته ۱ *همانطور که از تعریف کد باینری معلوم است به ازای یک الفبای به خصوص می‌توان کدهای باینری متفاوتی داشت.*

مثال ۱

$\Sigma = \{$	$a,$	$b,$	$c,$	d	$\}$
کد۱:	۰۰	۰۱	۱۰	۱۱	*کد با طول ثابت*
کد۲:	۰	۰۱	۱۰	۱	*کد با طول متغیر*
کد۳:	۰۰	۰۱	۱۱۰	۱۱۱	

احتمالا تاکنون متوجه شده‌اید که کدهای باینری با طول مختلف می‌توانند ابهام داشته باشند. مثلا در کد ۲ که در مثال بالا معرفی شد رشته‌ی ۰۰۱ می‌تواند معرف ab و یا aad باشد.
با توجه به این موضوع ممکن است این سوال پیش بیاید که چرا باید از کدهایی با طول متغیر استفاده کرد. در پاسخ به این سوال باید گفت که انگیزه‌ی استفاده از کد با طول متغیر، استفاده از تعداد بیت‌های کمتر برای نمایش یک رشته از حرف‌های الفبا است. این اتفاق وقتی می‌افتد که احتمال حضور حرف‌های مختلف الفبا در یک رشته متفاوت باشند. احتمال حضور حرف a را با f_a نشان می‌دهیم و فرض می‌کنیم احتمال حضور حرف‌های مختلف در یک رشته مستقل از هم باشند.

مثال ۲

احتمال حضور:	۰٫۶	۰٫۲۵	۰٫۱	۰٫۰۵		
$\Sigma = \{$	$a,$	$b,$	$c,$	d	$\}$	
کد۱:	۰۰	۰۱	۱۰	۱۱	$\Rightarrow$	*تعداد بیت لازم به طور متوسط* = ۲
کد۲:	۰	۰۱	۱۰	۱	$\Rightarrow$	*تعداد بیت لازم به طور متوسط* = ۱٫۵۵

تعریف ۲ *کد بدون پیشوند*[۵] *کدی است که به ازای هر $a, b \in \Sigma$، $c(a)$ پیشوند $c(b)$ نیست.*

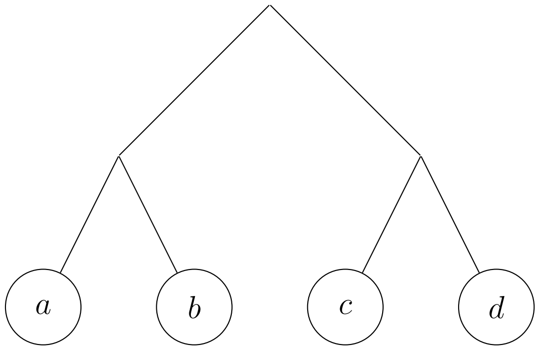
لم ۱ *کد بدون پیشوند، بدون ابهام است.*

تعریف ۳ *کد بهینه کدی است که مجموع احتمال حضور هر حرف الفبا ضرب در طول کد آن حرف، یعنی $\sum_{a\in\Sigma} f_a|c(a)|$، مینیمم باشد.*

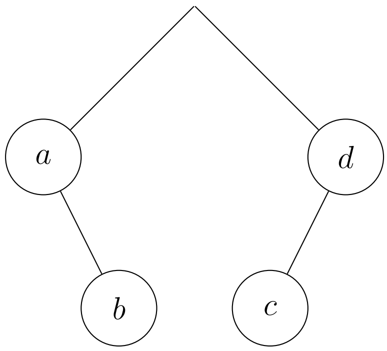
تعریف ۴ *درخت کد باینری*[۶] *درختی است که در آن حروف الفبا بر روی گره‌ها قرار گرفته‌اند و کد باینری متناظر با هر حرف توسط مسیر ریشه تا گره متناظر به دست می‌آید. به این صورت که به ازای هر حرکت به چپ ۰ و هر حرکت به راست ۱ در نظر می‌گیریم.*

[۵] prefix-free
[۶] binary tree

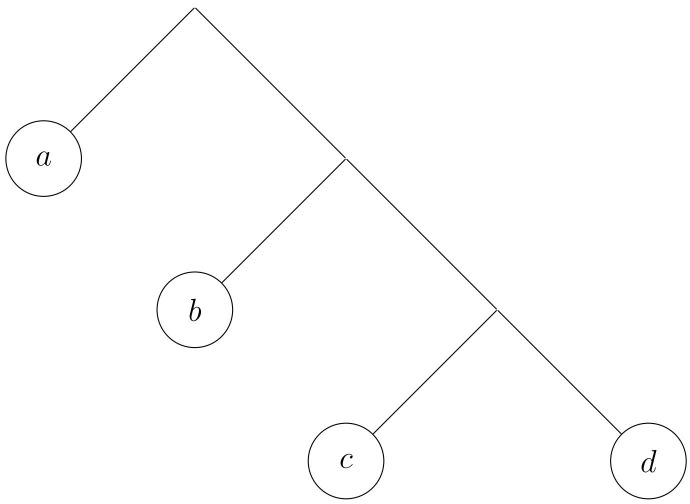
مثال ۳ *در ادامه درخت باینری متناسب با کدهای بیان شده در مثال ۱ آورده شده است:*
کد۱ :



:کد۲



:کد۳



در درخت یک کد باینری بدون ابهام، صرفا برگ‌ها دارای برچسب حرف‌ها هستند.

نکته ۲ *در درخت باینری طول کد باینری برابر است با عمق درخت به ازای آن حرف. پس برای رسیدن به کد بهینه مقدار زیر باید مینیمم شود:*

$$L(T) = \sum_{a \in \Sigma} f_a \mathrm{depth}(a)$$

لم ۲ *درخت کد بهینه، درخت دودویی کامل*[۷] *است.*

برهان. برای اثبات این موضوع از برهان خلف استفاده می‌کنیم. فرض کنید رأسی باشد که یک بچه داشته باشد در این‌صورت با حذف این بچه درخت کدی به دست می‌آید که به ازای حداقل یک حرف عمق کمتری دارد لذا درخت کد اولیه بهینه نبوده است. ■

۲.۲ مسأله پیدا کردن کد بهینه

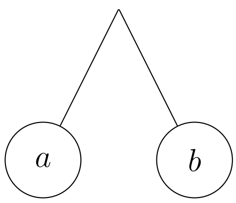
ورودی: الفبای Σ و مقادیر f_aها
خروجی: کد باینری بهینه

برای حل مساله‌ی مطرح شده دو ایده وجود دارد:

- حل از بالا به پایین[۸] که توسط فانو[۹] و شانون[۱۰] ارائه شده است. در این روش برای حل سوال از ایده تقسیم و حل استفاده شده است اما کد بهینه را همواره محاسبه نمی‌کند.
- حل از پایین به بالا[۱۱] که مبنای الگوریتم بهینه‌ای است که توسط هافمن ارائه شده است.

۱.۲.۲ الگوریتم هافمن

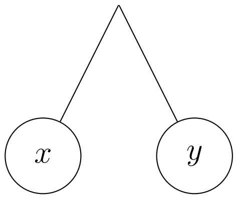
اساس الگوریتم هافمن به این صورت است که دو عضو x و y که دو عضو از بین حرف‌هایی هستند که کمترین تکرار را بین حروف الفبای Σ دارند، انتخاب می‌کنیم. حال الفبای Σ' را در نظر می‌گیریم که شامل تمام حروف الفبای Σ به جز x و y است و حرف جدید xy است. در این الفبای جدید احتمال حضور حرف‌ها، f'_aها، همان مقادیر قبلی است به ازای تمامی حروف به جز حرف xy که برابر با $f_{xy} = f_x + f_y$. حال دوباره این الگوریتم را برای Σ' و با مقادیر f'_aها تکرار می‌کنیم.
مراحل الگوریتم:
ـ اگر $|\Sigma| = ۲$ باشد، درخت کد بهینه به صورت زیر خواهد بود:



[۷] درخت دودویی کامل درختی است که در آن هر راس یا دو بچه دارد و یا برگ است.

[۸] top-down
[۹] Fano
[۱۰] Shannon
[۱۱] bottom-up

– اگر $|\Sigma| \geq 2$، آنگاه Σ' و f' را توسط روش ارائه شده در بالا به دست می‌آوریم و به صورت بازگشتی درخت T' را به دست می‌آوریم.
– در درخت T' گره‌ی xy را با درخت زیر جایگزین می‌کنیم.



۳.۲ اثبات صحت الگوریتم

قضیه ۳ *الگوریتم هافمن درخت کد بهینه را محاسبه می‌کند.*

برهان. برای اثبات قضیه دو لم و یک نتیجه مطرح می‌کنیم.

لم ۴ *در هر درخت دودویی کامل دو برگ همزاد در پایین‌ترین عمق وجود دارد.*

لم ۵ *اگر x و y حرف‌های با کمترین فرکانس (احتمال) در الفبا باشند در این صورت درخت کد بهینه‌ای وجود دارد که در آن برگ‌های x و y همزادند.*

برهان. فرض کنید این دو حرف همزاد نباشند. آنگاه طبق لم قبل، درخت باید دو برگ همزاد داشته باشد. آن‌دو را a و b بنامید. حال جای x و y را با a و b عوض می‌کنیم. از آنجایی که x و y دارای کمترین فرکانس هستند داریم:

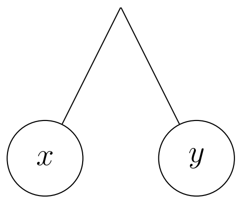
$$f_y \leq f_b, f_x \leq f_a$$

حال در صورتی که درخت اول را T و درخت دوم را T' بنامیم آنگاه در صورتی که عمق x به مقدار Δ_1 اضافه شده باشد، آنگاه عمق a نیز به همین مقدار کم شده است. این موضوع برای y و b نیز با مقدار Δ_2 روی می‌دهد. با توجه به موضوعات بیان شده داریم:

$$L(T) - L(T') = (f_a - f_x)\Delta_1 + (f_b - f_y)\Delta_2 \geq 0$$

در صورتی که مقدار عبارت فوق برابر صفر شود آنگاه درخت T' نیز درخت بهینه است و در غیر این صورت درخت اولیه بهینه نبوده است. زیرا ما توانستیم درخت بهتری پیدا کنیم. ■

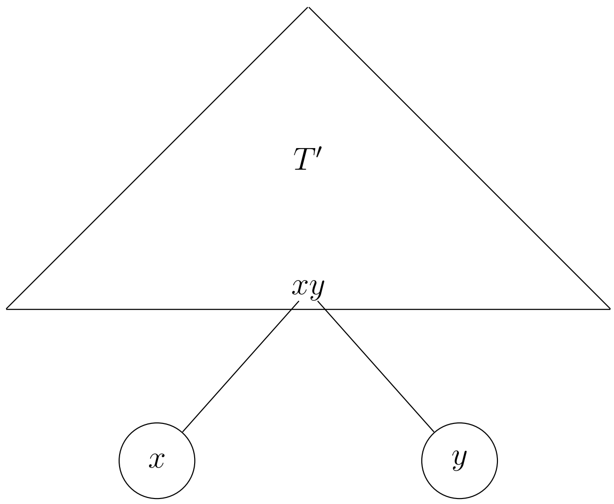
لم ۶ *فرض کنید T درخت کدی برای الفبای Σ با فرکانس f باشد که در آن حرف‌های با کمترین فرکانس x و y همزاد هستند. الفبای Σ' را که با جایگزینی حرف xy به جای حروف x و y با فرکانس $f'_{xy} = f_x + f_y$ بدست می‌آید (برای بقیه حروف $f'_a = f_a$) و درخت T' که با جایگزینی زیر درخت*



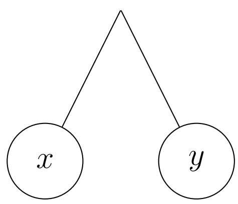
در درخت T با گره xy بدست می‌آید، در نظر بگیرید. آنگاه T' یک درخت کد برای الفبای Σ' با فرکانس f' است و داریم:

$$L(T) = L(T') + f(x) + f(y)$$

اثبات لم فوق با توجه به شکل زیر ساده است.



نتیجه ۱ *فرض کنید T' یک درخت کد بهینه برای الفبای Σ' با فرکانس f' باشد (که Σ' و f' همانند بالا از روی Σ و f تعریف می‌شوند) و T درختی باشد که با جایگزینی گره xy در درخت T' با زیر درخت*



حاصل می‌شود. آنگاه T یک درخت کد بهینه برای الفبای Σ با فرکانس f است.

برهان. واضح است که $L(T') + f(x) + f(y) = L(T)$. فرض کنید T یک درخت کد بهینه برای الفبای Σ با فرکانس f نباشد و درخت کد دیگری مانند $\mathcal{T}$ که در آن حرف‌های با کمترین فرکانس x و y همزاد هستند، درخت کد بهینه باشد. پس $L(\mathcal{T}) < L(T)$. درخت کد $\mathcal{T}'$ را برای الفبای Σ'

با فرکانس f' از روی $\mathcal{T}$ با توجه به لم ۶ می‌سازیم. چون T' بهینه است پس $L(\mathcal{T}') \geq L(T')$. اما داریم

$$\begin{aligned} L(\mathcal{T}) &= L(\mathcal{T}') + f(x) + f(y) \\ &\geq L(T') + f(x) + f(y) \\ &= L(T) \ , \end{aligned}$$

که تناقض است. ■

نهایتا اثبات قضیه با استفاده از نتیجه فوق به استقرا روی $|\Sigma|$ به سادگی تکمیل می‌شود.

■

۱.۳.۲ پیچیدگی الگوریتم

الگوریتم هافمن اگر به صورت ساده فوق پیاده‌سازی شود با استفاده از رابطه بازگشتی $T(n) = T(n-1) + O(n)$ دارای پیچیدگی $O(n^2)$ است. اما اگر الگوریتم با استفاده از داده ساختار هرم[۱۲] پیاده‌سازی شود دارای پیچیدگی $O(n \log n)$ خواهد بود که الگوریتم آن در زیر آمده است. این الگوریتم برای ذخیره‌سازی درخت کد از سه آرایه استفاده می‌کند و برای هر گره i مقادیر فرزند سمت راست $R[i]$، چپ $L[i]$ و مادر $P[i]$ را در آن ذخیره می‌کند.

Algorithm 1 Algorithm: BuildHuffman

```
function BuildHuffman(f_1, ..., f_n)
    for i = 1 to n do
        L[i] ← 0; R[i] ← 0
        Insert(i, f_i)
    for i = n to 2n − 1 do
        x ← ExtractMin( )
        y ← ExtractMin( )
        f_i ← f_x + f_y
        L[i] ← x; R[i] ← y
        P[x] ← i; P[y] ← i
        Insert(i, f_i)
    P[2n − 1] ← 0
```

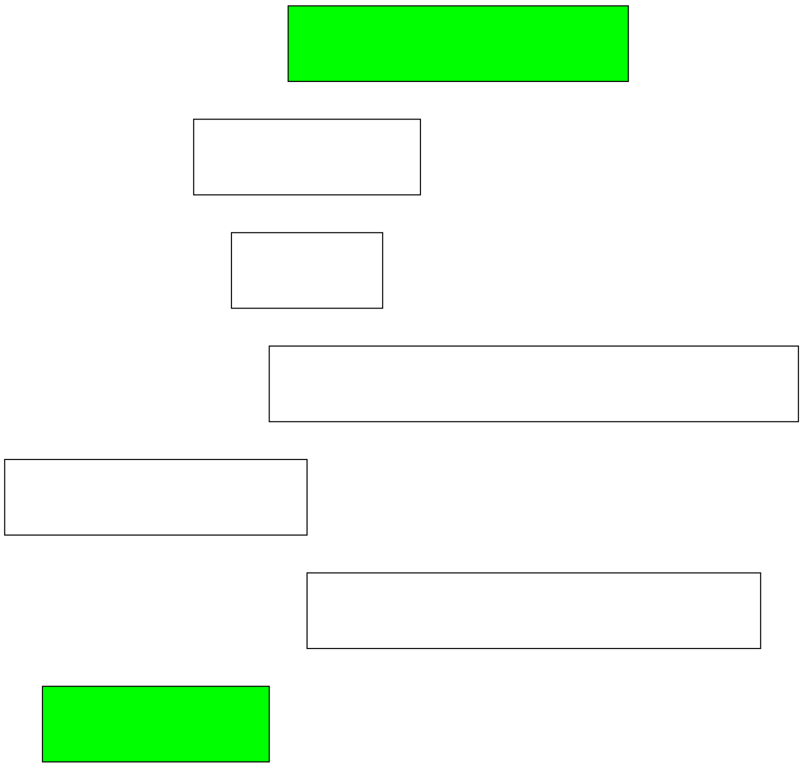
[۱۲] heap

۳ زمان‌بندی بازه‌ای

مسأله زمان بندی بازه‌ای[۱۳] به این صورت است که تعدادی کار با زمان‌های شروع و پایان معین موجود هستند. هدف این الگوریتم پیدا کردن بیشترین تعداد کاری است که با یکدیگر همپوشانی[۱۴] نداشته باشند. به بیان دیگر فرض کنید $s_{1}, \dots, s_{n}$ زمان شروع کارها و $f_{1}, \dots, f_{n}$ زمان پایان آن‌ها باشد. در این صورت هدف الگوریتم زمان‌بندی بازه‌ای پیدا کردن $\{i_{1}, i_{2}, \dots, i_{k}\} \subseteq \{1, 2, \dots, n\}$ است به‌گونه‌ای که

$$f_{i_j} \leq s_{i_{j+1}}$$

و k بیشینه مقدار ممکن باشد.

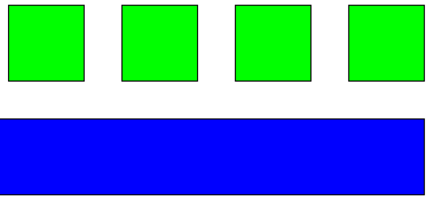


برای حل این مساله با استفاده از الگوریتم حریصانه می‌توان معیارهای مختلفی به عنوان بهتر بودن انتخاب در نظر گرفت از جمله معیارهای ممکن می‌توان موارد زیر را نام برد:

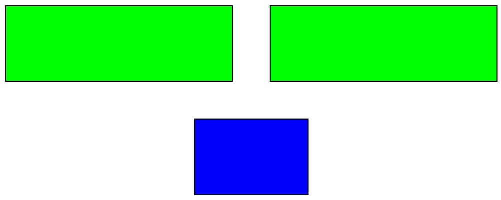
- زودترین زمان شروع
- زودترین زمان پایان
- کوتاهترین بازه‌ی انجام کار
- کمترین تعداد هم‌پوشانی

[۱۳] interval scheduling
[۱۴] overlap

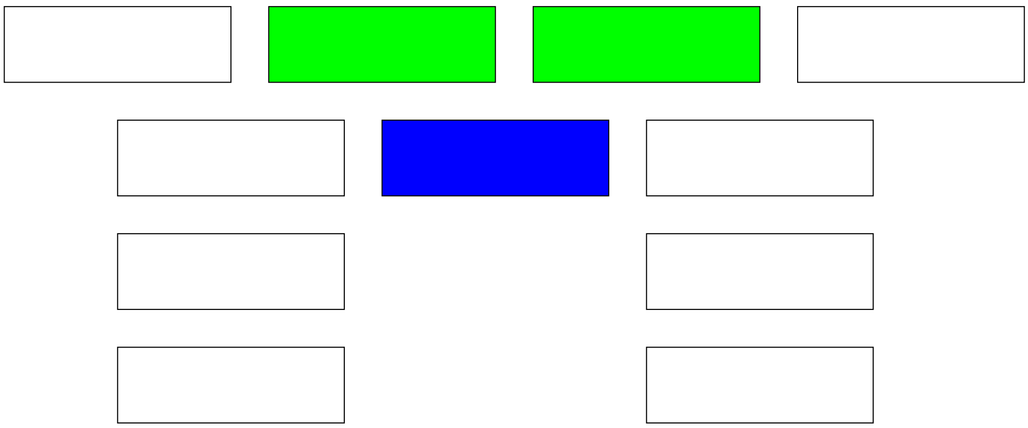
برای تمام معیارهای بیان شده به جز زودترین زمان پایان می‌توان مثال نقضی بیان کرد که در آن بیشترین تعداد کار انتخاب نمی‌شود.
(در هر مثال بلوک‌های سبز نشان‌دهنده‌ی کارهایی هستند که برای به دست آوردن ماکزیمم کارها باید آن‌ها را انتخاب کرد و بلوک‌های آبی کارهایی هستند که به اشتباه توسط معیار بیان‌شده انتخاب می‌شوند.)
زودترین زمان شروع:



کوتاهترین بازه‌ی انجام:



کمترین تعداد هم‌پوشانی:



پس با توجه به مسائل بیان شده به نظر می‌رسد زودترین زمان پایان بهترین معیار ممکن برای طراحی یک الگوریتم حریصانه باشد.

۱.۳ الگوریتم

Algorithm 2 Algorithm: IntervalScheduling

function IntervalScheduling($[s_1, f_1], ..., [s_n, f_n]$)
  [*assumes* $s_i < f_i$ *for every* i]
  Sort jobs by their finishing time
  $A \leftarrow \emptyset$
  for $j = 1$ **to** n **do**
    if job j is compatible with A **then**
      $A \leftarrow A \cup \{j\}$

در الگوریتم بالا دستور

Sort jobs by their finishing time

کارها را بر اساس زمان پایان مرتب می‌کند. یعنی یک لیست به صورت زیر برمی‌گرداند.

$$([s_۱, f_۱], ..., [s_n, f_n])$$

که

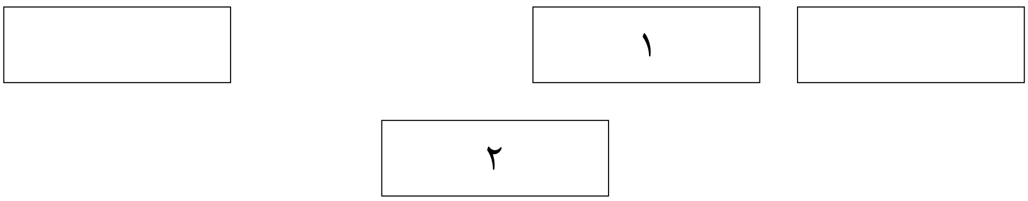
$$f_۱ \leq f_۲ \leq ... \leq f_n$$

A نیز لیست کارهای انتخاب‌شده است.
منظور از

job j is compatible with A

نیز این است که زمان شروع j بیشتر از زمان پایان همه کارهای موجود در A باشد.
پیچیدگی: پیچیدگی الگوریتم بیان شده به خاطر مرتب کردن کارها برابر است با $O(n \log n)$.

نکته ۳ *لازم به ذکر است که جواب بهینه الزاما یکتا نیست. برای مثال در مساله‌ی زیر تفاوتی بین انتخاب ۱ یا ۲ وجود ندارد و هر کدام می‌توانند در جواب بهینه حضور داشته باشند.*



۲.۳ بهینه بودن الگوریتم

قضیه ۷ *الگوریتم حریصانه‌ی بیان شده برای حل مساله‌ی زمان‌بندی بازه‌ای بهینه است.*

برهان. برای اثبات این قضیه از برهان خلف استفاده می‌کنیم. فرض کنید جواب الگوریتم حریصانه $i_1, ..., i_m$ باشد. فرض کنید جواب بهینه شامل k کار است که $k > m$. حال از بین تمام جواب‌های بهینه، جوابی را انتخاب می‌کنیم که در ابتدا شامل بیشترین اشتراک با جواب الگوریتم حریصانه است. فرض کنید تعداد این کارهای مشترک r باشد. پس جواب بهینه‌ای به صورت $i_1, ..., i_r, j_{r+1}, \ldots, j_m$ داریم که $r \leq m$. در این صورت داریم:

حریصانه:	i_1	i_2	$\cdots$	i_r	i_{r+1}	$\cdots$	i_m
بهینه:	i_1	i_2	$\cdots$	i_r	j_{r+1}	$\cdots$	

حال با توجه به مقدار r و k دو حالت به وجود می‌آید:

- $r = m$: در این صورت چون $k > m$، الگوریتم حریصانه می‌توانست کار j_{r+1} را نیز انتخاب کند زیرا هیچ همپوشانی‌ای با بقیه‌ی کارها ندارد. پس این حالت به تناقض می‌رسد.

- $r < m$: الگوریتم حریصانه‌ی بیان شده کمترین زمان پایان را انتخاب می‌کند لذا داریم:

$$f(i_{r+1}) \leq f(j_{r+1}) \ .$$

در این‌صورت با جایگزینی j_{r+1} با i_{r+1} در جواب بهینه فوق، یک جواب بهینه دیگر خواهیم داشت. دقت کنید که این تعویض مجاز است و مشکلی پیش نخواهد آمد زیرا زمان پایان i_{r+1} زودتر از j_{r+1} است. اما این موضوع با فرض بیشینه بودن r در تناقض است.

■