به نام خدا

پروژه‌ی مبانی برنامه‌سازی

دانشکده مهندسی کامپیوتر

گروه 1

موضوع: برنامه‌ی فشرده‌سازی

نوید اسلامی امیرآبادی

98100323

سجاد پاکسیما

98106286

دی 1398

سیر ساخت و توضیح الگوریتم‌ها

سیر ساخت پروژه‌ی ما به این صورت بود که بعد از خواندن و فهم الگوریتم‌ها، پروژه به صورت زیر تقسیم‌بندی شد:

- نوید اسلامی: Burrows-Wheeler Transform، Huffman و قابلیت فشرده‌سازی چند فایل(TARR) و مدل‌های ورودی گرفتن Interactive و Command Line.
- سجاد پاکسیما: Move to Front، PackBits و نوشتن داک و مقایسه‌ی الگوریتم‌ها.

مرحله‌ی برنامه‌نویسی به این شیوه شروع شد که با شروع از الگوریتم‌های Huffman و PackBits و Burrows-Wheeler Transform و Move to Front، برنامه‌ها را در فایل‌های cpp و h مجزا نوشته و در نهایت با هم Compile کردیم (مراجعه کنید به روش کامپایل کردن در آخر داک).

بعد از آن، مرحله‌ی ساخت روش‌های ورودی گرفتن فرا رسید که روش پیاده‌سازی آن‌ها همراه با الگوریتم‌ها را در ادامه توضیح خواهیم داد. همچنین دقت کنید که منظور ما از حرف در این داک همان بایت خوانده شده در فایل است.

دقت کنید که هر الگوریتم، در قالب یک Class نوشته شده است که دارای Constructor و Destructor و ... هستند و نیز دارای یک یا چند Driver Function هستند. این Driverها برای سهولت در به کارگیری الگوریتم‌ها نوشته شده‌اند و به این شکل عمل می‌کنند که فقط نام فایل یا فایل‌ها را ورودی می‌گیرند و Streamهای مناسب برای آن‌ها را ایجاد می‌کنند و اعمال مربوط به Error-Checking را نیز انجام می‌دهند.

- Huffman: این الگوریتم را با استفاده از یک Class به نام huffmanItem و یک Struct به نام node که به ترتیب برای به کارگیری در Priority Queue (Min Heap) و ساختن Huffman Tree به کار می‌روند. برای HuffmanItem اپراتورهای مقایسه‌ای تعریف شده‌اند که بتوان با استفاده از آن‌ها به راحتی از Min Heap استفاده کرد، همچنین هر کدام از این دو دارای خصیصه‌های مورد نیاز هستند که به شرح زیر هستند:

  huffmanItem: یک کاراکتر برای نمایش دادن مقدار کاراکتر در آن(c)، تعداد (count) و یک پوینتر به node مربوطه‌ی آن حرف در درخت هافمن.

  node: یک label برای نمایان کردن کاراکتر آن node و دو پوینتر به ترتیب برای بچه‌های راست و چپ آن در درخت آن.

  حال با استفاده از زیرساختی که در این‌جا توضیح داده شد، الگوریتم Huffman را روی فایل ورودی اجرا می‌کنیم. یعنی برای هر حرف تعداد بارهایی که در فایل آمده را محاسبه می‌کنیم و در هر مرحله دو huffmanItemای که تعداد درون آن‌ها از همه کمتر هستند را از Min Heap حذف کرده و به درخت

Huffman یک node دیگر اضافه می‌کنیم که بچه‌هایش این دو huffmanItem هستند. در ادامه نیز یک huffmanItem جدید در Min Heap خود اضافه می‌کنیم که مربوط به آن node جدید ما در درخت است، اما count آن برابر با مجموع countهای دو huffmanItem قبلی است. به همین روش ساختن درخت را ادامه می‌دهیم تا در Min Heap ما فقط یک huffmanItem باقی بماند.

حال با استفاده از این درخت، می‌توانیم فایل ورودی را فشرده کنیم. اول یک آرایه از vectorها می‌سازیم به نام alias که درواقع همان رشته از بیت‌هایی است که به یک کاراکتر خاص نسبت می‌دهیم. این کار را با الگوریتم بازگشتی DFS انجام می‌دهیم. حال با استفاده از alias به این شکل عمل شده که یک متغیر buffer داریم و مسیر از ریشه‌ی درخت Huffman را که به صورت یک دنباله از 0 (چپ) و 1 (راست) است و در alias ذخیره شده است را بیت به بیت به buffer اضافه می‌کنیم. به این صورت که هر دور buffer را در 2 ضرب نموده و مقدار آن بیت را به buffer اضافه می‌کنیم. هر موقع که هشت بیت buffer مقداردهی شد، buffer را در خروجی چاپ می‌کنیم. در نهایت هم اگر تعدادی بیت در buffer باقی‌مانده بود، buffer را آنقدر شیفت چپ می‌دهیم تا هشت بیت زا پر کنیم و سپس به فایل خروجی می‌ریزیم.

همچنین برای اینکه بتوانیم فایل خروجی را بازیابی کنیم، باید اصطلاحاً یک Lookup برای درخت Huffman خود داشته باشیم و آن را ذخیره کرده باشیم. برای انجام این کار، به این شیوه عمل می‌کنیم که به nodeهای اعداد از 0 تا (تعداد nodeها) نسبت می‌دهیم و آن‌ها را به همان ترتیب خروجی می‌دهیم. یال‌ها را نیز به این شیوه ذخیره می‌کنیم که به ازای هر یال دو عدد داریم که نمایان‌گر دو رأس پدر و پسر آن یال هستند، آن دو عدد را در فایل خود ذخیره می‌کنیم. توجه داشته باشید که اول یال‌های به سمت چپ و سپس یال‌های به سمت راست را ذخیره می‌کنیم.

حال برای خواندن آن درخت تنها کافی است که nodeها را به همان ترتیب ذخیره‌شده بخوانیم، و هر دو عدد متوالی را که می‌خوانیم، راس پدر را به بچه‌ی متناظر متصل می‌کنیم.

حال برای بازیابی کردن فایل اولیه ساز و کاری همانند فشرده کردن آن به کار رفته است. به این شکل که بایت به بایت از فایل خوانده می‌شود، و با توجه به بیت‌ها، از پر ارزش به کم ارزش، در درخت Huffman از ریشه شروع به حرکت می‌کنیم و تا وقتی که به یک برگ نرسیده‌ایم، به حرکت کردن ادامه می‌دهیم و نهایتاً که به یک برگ رسیدیم، کاراکتر یا بایت مد نظر را در خوروجی ذخیره می‌کنیم.

به این شیوه می‌توانیم یک فایل را هم فشرده هم بازیابی کنیم.

- PackBits: اول از فشرده‌سازی این الگوریتم شروع می‌کنیم. در نوشتن این بخش به این صورت عمل شده که دو State داریم که به ترتیب مربوط به «برابر بودن دو کاراکتر پشت سر هم» و «برابر نبودن دو

کاراکتر پشت سر هم» هستند. هر موقع که چندین کاراکتر برابر متوالی ورودی داده شود، حالت اول چند بار اجرا می‌شود. همین موضوع برای چند کاراکتر متمایز نیز برقرار است. همچنین یک متغیر counter داریم که نمایش‌دهنده‌ی تعداد کاراکترها در Stateای که داخلش هستیم، است. برای State نابرابری نیز یک استرینگ هم داریم که کاراکترهای ورودی را در آن ذخیره می‌کنیم. هر بار که counter ما به عدد 127 برسد، یا هر بارکه Stateمان عوض شود، اگر در State برابری بودیم، counter- و کاراکتر تکراری مربوطه که در harf ذخیره شده است را خروجی می‌دهیم. اگر هم در State نابرابری بودیم counter و استرینگ خود را که طولش همان counter است را خروجی می‌دهیم. تنها حالتی که باقی می‌ماند این است که برنامه به اتمام برسد اما ما در یکی از Stateها باشیم و خروجی نداده باشیم. این حالت را هم به صورت جداگانه به همان شکل بالا، مدیریت می‌کنیم.

بازیابی این الگوریتم هم به این شکل است که در هر مرحله، یک کاراکتر را می‌خواند، اگر کاراکتر ما مقدار عددیش منفی بود، کاراکتر بعدی را می‌خواند و آن کاراکتر را به اندازه‌ی قدر مطلع آن کاراکتر اولی چاپ می‌کنیم. اگر هم کاراکتر اول یک مقدار مثبت بود، به آن تعداد کاراکتر بعدش را می‌خوانیم و همه‌ی آن‌ها را بدون عوض کردن در خروجی می‌آوریم.

- Burrows-Wheeler Transform: اول از تبدیل اولیه‌ی آن شروع می‌کنیم. توجه کنید که در فایل ذخیره شده، اندازه‌ی فایل را نیز ذخیره می‌کنیم.

الگوریتم به کار رفته در این‌جا الگوریتمی از $O(n \lg n)$ است که در محاسبه‌ی Suffix-Array از آن استفاده می‌شود. برای اجرای این الگوریتم اول فایل ورودی را خوانده و در یک استرینگ ذخیره می‌کنیم. حال باید شیفت‌های دوری آن فایل را Lexicographically مرتب‌سازی کنیم. به شکل Radix-Sort روی Classهای تعریفی خودمان مرتب می‌کنیم. اول روی استرینگ حرکت می‌کنیم، و از روی آن یک آرایه‌ی p می‌سازیم که در آن ترتیبی موقتی از شیفت‌های دوری استرینگ اصلی قرار دارد. به طوری که استرینگ اصلی با عدد 0، یک شیفت دوری با 1 و ... مشخص می‌شوند. حال، با حرکت کردن روی این استرینگ، به هرکدام از این اعداد، یک Class در آرایه‌ی c نسبت می‌دهیم که در اول برابر است با مقدار کاراکتری که اول آن شیفت دوری قرار دارد. درواقع این Classها را طوری مقدار دهی می‌کنیم که اگر مقدار c یک شیفت از دیگری کمتر بود، معنی‌اش این باشد که از نظر Lexicographically هم استرینگ دو به توان hتایی آن‌ها هم همین حالت را نسبت به هم داشته باشند.

توجه کنید که c_i به معنی Class شیفت iام است.

حال p را برحسب c با الگوریتم Counting-Sort مرتب می‌کنیم. از حالا به بعد، با یک حلقه پیش می‌رویم که تا $\lg n$ جلو می‌رود و در مرحله‌ی hام، می‌خواهیم شیفت‌های دوری را بر حسب دو به توان 1

+ h کاراکتر اول آن‌ها Lexicographically مرتب کنیم. به این شکل عمل می‌کنیم که دو آرایه‌ی جدید pn و cn را می‌سازیم و هر خانه از pn را برابر خانه‌ی متناظر در p اما دو به توان h تا به چپ. یعنی عملاً داریم یک دو به توان hتایی عقب‌تر را در نظر می‌گیریم. ما می‌دانیم که این pnها بر حسب دو به توان hتا کاراکتر جلویی خود مرتب هستند، طبق فرض استقرایی‌ای که داریم. حال باید pnها را به گونه‌ای مرتب کنیم که انگار زوج مرتب c این دو به توان h + 1تایی‌ها را مرتب کرده‌ایم. این کار را با Stable-Counting-Sort انجام می‌دهیم. درواقع می‌خواهیم طوری مرتب سازی کنیم که ترتیب آن pnهایی که عضو اول زوج مرتبشان با هم برابر هستند، ترتیب زوج دوم خود را حفظ کنند، که با استفاده از مجموع جزئی گرفتن از آرایه‌ی cnt انجام می‌دهیم.

حال باید cnها که همان Classهای جدید هستند را محاسبه کنیم. برای انجام این کار، روی pnها حرکت می‌کنیم و عملاً زوج مرتب‌های مذکور را می‌سازیم، و از آنجایی که pnها مرتب شده هستند، فقط کافی است که زوج مرتب‌های متوالی و کنار هم را با هم مقایسه کنیم و یک متغیر class داشته باشیم که اول صفر است و در صورت برابر نبودن زوج مرتب‌ها آن را به علاوه‌ی یک می‌کنیم. حال مقدار cn آن pn را برابر با class قرار می‌دهیم. نهایتاً نیز، c و cn را و نیز p و pn را با هم swap می‌کنیم.

با انجام این اعمال، توانستیم که شیفت‌های دوری این استرینگ را مرتب‌سازی کنیم. حال با استفاده از p نهایی، می‌توانیم به راحتی Burrows-Wheeler Transform را به دست آوریم، که برابر است را رشته‌ی t که در آن $t_i = s_{p[i]-1}$ (با در نظر گرفتن $n = -1$). این استرینگ را در فایل ذخیره می‌کنیم.

حال می‌رسیم به برعکس این تبدیل. برای انجام این‌کار، فایل را ورودی گرفته و استرینگ t را بازیابی می‌کنیم. حال استرینگ را در s کپی می‌کنیم و s را مرتب می‌کنیم تا به حروف اول شیفت‌های دوری مرب شده برسیم. حال باید با استفاده از t و s استرینگ اولیه را به دست آوریم. یک آرایه‌ی دو بعدی را با نام occ تشکیل می‌دهیم که در occ_i اندیس‌هایی از t آمده‌اند مثل x که $t_x = i$. دقت کنید که occ_i نزولی مرتب شده است. این کار را با O(n)، با حرکت برعکس روی t و استفاده از Stack در occ انجام می‌دهیم.

حال، می‌خواهیم آرایه‌ای به نام leftShift را محاسبه کنیم که $leftShift_i$ برابر است با اندیسی از t که اگر شیفت چپ iام را یک بار دیگر شیفت بدهیم به آن می‌رسیم. برای محاسبه‌ی این آرایه، شروع می‌کنیم از 0 تا n رو t حرکت کردن و به این شکل عمل می‌کنیم که $leftShift_i$ را برابر با کوچک‌ترین عنصر $occ_{s[i]}$ قرار می‌دهیم و سپس آن عنصر را حذف می‌کنیم.

این عمل درست است چون که t را Lexicographically مرتب کرده بودیم، پس اگر یک شیفت با اندیس کمتر را یک بار دیگر شیفت بدهیم، برابر می‌شود با کوچکترین اندیسی که تا آلان حذفش نکرده‌ایم

و t آن برابر با s_i است. چرا که اگر غیر از این بود، در شرط مرتب بودن t به تناقض می‌خوردیم. یعنی به این نتیجه می‌رسیم که این عنصر از t باید در اندیس‌های بزرگ‌تر می‌آمد. چون که حرف اولشان برابر ولی ادامه‌ی آن شیفت که اندیسش بیشتر است از ادامه‌ی اندیس فعلی بزرگ‌تر است. پس به تناقض می‌رسیم. درواقع این ادعا را با برهان خلف به راحتی می‌توان اثبات نمود، به همان شیوه‌ی مذکور.

پس آرایه‌ی leftShift را که محاسبه کردیم، می‌توانیم با استفاده از s و leftShift یکی یکی تمامی حروف استرینگ اولیه را از 0 تا n به دست آوریم. به صورتی که در کد پروژه مشاهده می‌شود. چون که حرف بعدی را با یک دور شیفت چپ دادن می‌توان به دس آورد.

بنابراین توانستیم استرینگ و فایل اولیه را به دست آوریم.

- Move to Front: تبدیل اولیه‌ی این برنامه از این قرار است که ما یک آرایه به نام list داریم که در آن تمامی الفبا مرتب آمده‌اند. حال تبدیل را این‌گونه انجام می‌دهیم که روی list حرکت می‌کنیم تا اینکه به عضوی مثل $list_i$ برسیم که $list_i = c$، که منظور از c همان کاراکتری است که اکنون به آن در فایل رسیده‌ایم. حال i را خروجی می‌دهیم و با عمل swap کردن $list_i$ را به اول لیست می‌آوریم. همین روند را برای تمامی کاراکترهای فایل انجام می‌دهیم. این عمل باعث می‌شود که تعداد حروف تکراری در فایل ما افزایش یابند.

  حال برای بازیابی فایل اولیه، کافی است که الگوریتمی مثل الگوریتم بالا را اجرا کنیم. درواقع همان list را داریم، که اول مرتب است. حال به جای آنکه در list به دنبال $list_i = c$ باشیم، صرفاً $list_c$ را خروجی می‌دهیم و $list_c$ را با swap کردن به اول list می‌آوریم. این عملیات دقیقاً برعکس عملیات بالا است و به استرینگ اولیه می‌رسیم.
- فشرده کردن چند فایل: برای انجام این عمل سعی شده که مثل روش این کار در Linux پیش برویم. در Linux برنامه‌ی tar یک برنامه است که چند فایل را ورودی می‌گیرد و به یک فایل تبدیل می‌کند. در اینجا ما تلاش کردیم که یک برنامه همانند tar را تحت عنوان tarr یا Tar Replica بسازیم. یعنی عملاً چند فایل را ورودی می‌گیرد، و در یک فایل بزرگتر، نام آن فایل، اندازه‌اش و خود فایل را ذخیره می‌کند. نام و اندازه را برای این ذخیره می‌کند که در مرحله‌ی بازیابی هم نام فایل را داشته باشیم، و هم اینکه فایل مورد نظر را از بقیه‌ی فایل‌های داخل فایل بزرگ تمیز دهیم.

  الگوریتم بازیابی آن هم طبق همین منطق عمل می‌کند. درواقع نام فایل و اندازه‌ی آن را می‌خواند و به آن تعداد بایت بعدی در یم فایل جدید با نام خوانده شده ذخیره می‌کند.
- سیستم ورودی گرفتن: این سیستم هم به صورت کاملاً Straight-Forward طراحی شده است. یعنی تمامی ورودی‌ها را خوانده، Error-Check کرده و در صورت نبودن مشکلی، ورودی‌های مذکور را به

توابع Driver مربوطه می‌دهد تا که با استفاده از آن‌ها خروجی مد نظر را چاپ کند. توجه کنید که حالت Command Line با استفاده از ورودی‌های argc و argv گرفته شده‌اند، که argc تعداد ورودی‌ها و argv آرایه‌ای از استرینگ‌ها است که خود ورودی‌ها هستند.
برای نحوه‌ی استفاده از این دو حالت ورودی، به آخر فایل مراجعه کنید.

همچنین یک نکته‌ی شایان توجه این است که در نمودارهای مقایسه‌ی الگوریتم‌ها، Move to Front فقط به همراه Huffman و Burrows-Wheeler Transform فقط با PackBits تست شده‌اند.
دلیل این امر این است که اجرا کردن Burrows-Wheeler Transform روی الگوریتم Huffman تأثیر مثبتی ندارد، چرا که خروجی الگوریتم هافمن تنها به مجموعه‌ی حروف فایل بستگی دارد، نه به ترتیب آن‌ها. چون که با عوض کردن ترتیب حروف، اندازه‌ی فایل خروجی عوض نمی‌شود.
همچنین برای Move to Front و PackBits این موضوع نیز صادق است. چون که در Move to Front، دو حرف که کنار هم باشند و برابر نباشند، در خروجی هم همین خاصیت را دارند، و چند حرف کنار هم که مساوی هستند هم تقریباً به همان حالت باقی می‌مانند. حتی کمی بد تر می‌شوند چون که حرف اول آن عوض می‌شود و با بقیه‌ی حروف نامساوی می‌شوند. مثلاً AAA در Move to Front به A00 تبدیل می‌شود، که ورودی بدتری برای PackBits است.
پس به نظر می‌آید که این زوج الگوریتم‌ها با هم خوب کار نمی‌کنند و تصمیم گرفته شد که این‌ها را با هم تست نکنیم. چرا که یک نتیجه‌ی بدتر یا مساوی حالت بدون وجود آن‌ها و Huffman و PackBits صرف را به ارمغان می‌آورند.

راهنمای استفاده از zipper

این برنامه دو روش استفاده دارد، که به شرح زیر می‌باشند:

1. روش Command Line: در این روش، باید در یک محیط Shell به روش زیر از برنامه استفاده نمود:

zipper [options] [args]

که اینجا options یک رشته از حروف به طول بیشتر از 1 و کتر از 6 است. که به صورت mabcd می‌باشد.

اینجا m نمایان‌گر عملیات یا Commandای است که ما می‌خواهیم استفاده کنیم. این مقدار حتماً باید یکی از حروف c یا x باشند، که به ترتیب مخفف Create و Extract هستند.

حروف a و b برای تعیین کردن الگوریتم‌های بهبوددهی هستند و می‌توانند مقادیر m و b را داشته باشند، که به ترتیب به معنای Move to Front و Burrows-Wheeler Transform هستند. این دو option اجباری نیستند و می‌توان هر کدام از آن‌ها را به کار نبرد. حتی می‌توان هر دو الگوریتم را به کار برد، اما ترتیب اجرا کردن این الگوریتم‌ها همان تریتیبی است که به آن ورودی می‌دهیم. همچنین نمی‌توان یک الگوریتم را دو بار ورودی داد.

حرف c نمایان‌گر الگوریتم فشرده‌سازی ما است، که می‌تواند مقادیر p یا h را به خود بگیرد. این مقادیز به ترتیب به معنای PackBits و Huffman هستند. وجود این حرف اجباری است.

حرف d یک option اختیاری است و می‌تواند به کار نرود. این حرف تنها می‌تواند مقدار o را به خود بگیرد، که به معنای Output Name است. با به کار گیری این option، نام آخری که در لیست نام‌های [args] وارد شده است، به عنوان نام فایل فشرده‌ی نهایی ما در نظر گرفته می‌شود.

برای Decompress کردن یک فایل فشرده نیز، کافی است آن را با همان تنظیماتی که آن را به وجود آورده‌ایم به zipper ورودی دهیم تا فایل‌های اولیه به ما داده شوند. لازم به ذکر است که به‌جای c باید x گذاشت، و اگر هم option مربوط به o داشتیم، آن را به کار نخواهیم برد.

این برنامه توانایی ورودی گرفتن چند فایل را نیز دارد، و فقط کافی است که آن‌ها را در [args] ذکر کنیم. همچنین لازم است که حداقل یک فایل به برنامه ورودی بدهیم تا آن را فشرده کند. اگر هم فشرده کردن چند فایل مد نظر است، استفاده از o لازم و ضروری است.

2. روش Interatctive Menu: در این روش، صرفاً اجرا کردن برنامه بدون ورودی دادن به آن، یک Menu با توضیحات مربوطه برای شما باز می‌شود.

این Menu اول از شما عملیات مد نظرتان را می‌پرسد، سپس ترتیب و اوناع الگوریتم‌های بهبودی مد نظر شما را می‌پرسد، و بعد از آن الگوریتم فشرده‌سازی مدنظر و در نهایت در صورت تمایل به فشرده‌سازی چند فایل یا نام‌گذاری خروجی، نام فایل خرورجی را ورودی می‌گیرد.

سپس نیز شروع به ورودی گرفتن فایل یا فایل‌های مربوطه برای فشرده‌سازی می‌کند.

در صورت ورودی دادن چند فایل، باید در صورت اتمام همه‌ی ورودی‌ها، یک رشته‌ی 1- به برنامه ورودی داده شود.

بعد از اتمام عملیات از شما پرسیده خواهد شد که آیا تمایل به اجرا کردن برنامه برای بار دوم دارید یا خیر، که ورودی آن y یا n است، که به ترتیب به معنای Yes و No هستند. ورودی بخش تمایل به فشرده‌سازی چند فایل نیز به همین شیوه هستند.

کامپایل کردن از Source

برای کامپایل کردن Source این برنامه، درون فولدر پروژه رفته و از دستور زیر استفاده کنید:

```
g++ source/*.cpp -fpermissive -o bin/zipper
```

این دستور تمامی فایل‌ها را با هم کامپایل کرده و خروجی را در فولدر bin با نام zipper دخیره می‌کند.