Assignment A0

Computer Systems 2025
Department of Computer Science
University of Copenhagen

Martin Oldhøj Lavrentiew <zhb423>
Kristian Gert Sørensen <jxk652>
Tobias Dübeck Kristensen <rlj430>

Sunday, September 21, 16:00

GitHub Link: `https://github.com/mullk/CompSys-MTK-`

Introduktion:

Fokus for denne opgave er, at vi dels kan åbne en fil samt afgøre, hvilken af fem kategorier filen kan inddeles under: Tom, ASCII, ISO-8859-1, UTF-8 eller data. Kategoriseringen udføres ud fra de udleverede kriterier i A0.
Vores kategorisering eftertjekkes af vores testscript, hvor vores output sammenholdes med det forventede output fra file kommando i Linux.[1]

Tilgang til design beslutninger:

Løsningen klassificerer files indhold. Derfor er først skridt at åbne filen og dens indhold samt indlæse som bytesekvenser ved *fread*. Det er hensigtsmæssigt af to grunde:
1) Ved UTF-8 tjek sammenstilltes bitesekvenserne med hexadecimal.
2) dataen fra filen indlæses kun én gang.
Efter indlæsningen tjekkes, om filen er tom. Hvis det ikke er tilfældet, så tjekkes filens indhold ud fra følgende model:

[[[[ASCII]ISO-8859-1]UTF-8]Data]

Hvis værdien af filens bytesekvenser placerer sig i intervallet for ASCII karakterer, så må filen være ASCII.
Hvis dette ikke er tilfældet, så tjekkes bytesekvenserne i henhold til ISO-8859-1's interval. Intervallet indholder ASCII samt værdier fra 160-255 jf. A0. Intervallet fra 128-159 er reserveret til kontroltegn, så de bliver ikke tildelt standartisrede tegn. Det kan sammenlignes med ASCII's interval fra 0-31.[2]
Hvis det ikke er tilfældet, så tjekkes bytesekvenserne ud fra UTF-8's mønster

[1] `https://www.geeksforgeeks.org/linux-unix/file-command-in-linux-with-examples/`
[2] `https://chatgpt.com/share/68c930f3-4a7c-800c-91f2-a1b6e2b2c04d`

for bytesekvenser, som uddybes i afsnittet 'Ikke trivielle dele af koden'.
Hvis det heller ikke er UTF-8, så kategoriseres filen jf. A0 som data.

Tvetydigheder i opgaven:

I testfasen er vi stødt på to tvetydigheder.

1) Om filen skulle kategoriseres som UTF-8 eller ISO-8859-1.
Problemet opstår ved karakterer såsom æ, ø og å. Alle tre karakterer er defineret i ISO-8859-1 som henholdsvis 0xE6, 0xF8 og 0xE5. De er ligeledes defineret i UTF-8 som 0xC3 0xA6, 0xC3 0xB8 og 0xC3 0xA5.[3]
I vores løsning burde æ, ø og å karakteriseres som værende ISO-8859-1, da ISO-8859-1's mønster tjekkes før UTF-8. Dette har vi ikke kunnet bekræfte gennem vores test. Vi har kun kunnet få æ, ø og å til at blive ISO-8859-1, hvis vi angav deres konkrete hexadecimal. Hvis vi blot anvendte tegnet fx. 'æ', så blev det automatisk tolket af file kommando i Linux som værende UTF-8. Dette har besværliggjort vores test af ISO-8859-1.
Problemet skyldes bytesekvenser i UTF-8, som er længere end 1 byte. Ved 2, 3 eller 4 byte vil startbyten være 1 samt de efterfølgende bytes vil også starte med 1 jf. A0. Idet ISO-8859-1 overlapper med UTF-8, og adskiller sig fra ASCII, da den 8 bit i bytesekvensen er 1, så vil bytesekvensen kunne fortolkes som både ISO-8859-1 og UTF-8. Afgørelsen af dette er op til file kommandoen. Vores test viser, at UTF-8 grundet dens udbredelse i praksis fungerer som standard.[4]
Derfor bliver æ, ø og å fortolket som UTF-8 af file kommandoen.

2) I vores test har vi erfaret, at startbyten for 4 bytesekvenser i UTF-8 har en øvre grænse. Den er 11110100. Dermed kan fx \xF5\x80\x80\x80\ ikke repræsenteres i UTF-8. Det skyldes, at startbyten overskrider 11110100 pga xF5, som er 11110101 i binær, er større. Dette er problematisk af to grunde.
Den første er, at der ikke er sat et loft på bytesekvensen i opgavebeskrivelse i A0. Vi har dog valg at implementere dette i vores løsning ved et simpelt tjek:
else if ((b0 & 0xF8) == 0xF0 && b0 <= 0xF4) // 4-byte sekvens (11110xxx 10xxxxxx 10xxxxxx 10xxxxxx)
Det næste problem er, at vi jf. kategoriseringen burde forvente, at output bliver data. Ikke desto mindre bliver output for file kommando: Non-ISO extended-ASCII text. Dette output falder uden for kategorierne i A0. Derfor har vi ikke oprettet en ny kategori til at håndtere denne.

Ikke trivielle dele af koden:

I dette afsnit vil vi kommentere bitwise-and tjek. Det er mekanismen, som tjekker for UTF-8's bytesekvenser. Vi vil forklare det ud fra følgende linje i vores kode:

[3] https://chatgpt.com/share/68c96bcc-1028-800c-89a8-e78be53c69f1

[4] https://chatgpt.com/c/68c9afb3-0a2c-832b-97e7-5323dfe05508, https://en.wikipedia.org/wiki/UTF-8, https://en.wikipedia.org/wiki/ISO/IEC_8859-1

((b0 & 0x80)==0x00)

hexa 0x80 er 128 i decimal og 10000 0000 i binær. Formålet med 0x80 er at tjekke, om sekvensen er ASCII. Alle ASCII tegns bytesekvenser starter med 0 fx. 0100 1000, 72 i decimal og 'H' i ASCII. I den konkrete linje (b0 & 0x80) udføres bitwise-and tjek: Hver bit vil nu evalueres ud fra sandhedstabellen. I det

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Tabel 1: Sandhedstabel for bitwise AND

konkrete eksempel med 'H':
0100 1000
1000 0000
I det kombinationen 1 og 1 ikke forekommer vil reslutatet blive 0000 0000. Herefter tjekkes resulatet af (b0 & 0x80) med == 0x00. Resultatet vil, hvis det er en ASCII, blive 0000 0000, da bite 8 er 0. Hvis bit nummer 8 er 1, så kan det ikke være ASCII, da denne bytesekvens vil være ud for ASCII's interval.
Denne tankegang er gennemgående i hele koden. Herudfra afgøres om given bytesekvens er inden for det tilladte interval. Ydermere inddrages bytesekvenslængde, da nogle UTF-8 har en bytesekvens som kan strække sig fra byte 1 - 4.

Beskriv hvordan man compiler koden, kører test og reproducerer test resultat:

Man kan udføre de ovenstående instruktioner ved at unzippe *src* filen og så skrive følgende i terminalen: *"bash test.sh"*. Derefter bliver en mappe med testfiler genereret og terminalen printer testresultaterne.

Test's

Formålet med vores tests er at sammenligne outputtet fra vores program med referenceprogrammet `file(1)`. På den måde kan vi verificere, at vores implementation klassificerer filer korrekt i forhold til de filtyper, som opgaven kræver:

`test.sh` genererer testfilerne, kører både vores program og referenceprogrammet `file(1)`, og sammenligner derefter output med `diff`. Dette giver en automatiseret måde at kontrollere korrektheden af vores implementation.

- empty (tomme filer)

- ASCII text (kun de tilladte bytes jf. A0)
- ISO-8859-1 text (latin1, inkl. æøå og andre 8-bit tegn)
- UTF-8 text (gyldige multibyte sekvenser)
- data (alt der ikke falder i de andre kategorier)

Vi har selv genereret testfiler ved hjælp af `printf` i shell med escape-sekvenser for at indsætte særlige bytes. For eksempel:

- `printf "\xE6\xF8\xE5\n"` for at lave en ISO-8859-1 fil med de danske bogstaver æ, ø og å.
- `printf "\x00"` for at indsætte null-bytes og dermed skabe en data-fil.
- `printf "Hej 😀 \n"` for at teste gyldig UTF-8 med en 4-byte emoji.

Alle testfiler oprettes automatisk af vores `test.sh` script, så de altid kan genskabes fra bunden.
Opgaven kræver mindst 12 tests pr. kategori. Vi har derfor lavet:

- Over 12 ASCII-tests med forskellige kontroltegn, mellemrum, tab, carriage return og blandede grænsetilfælde.
- Over 12 ISO-8859-tests, som dækker æøå, accenter, tyske umlauter, islandske/færøske bogstaver og grænsetegn som NBSP (0xA0) og ß (0xFF).
- Over 12 UTF-8-tests med emoji, kinesiske tegn, arabisk, græsk, hebraisk, Hindi, kombinerende accenter og overflod af blandede sekvenser.
- Over 12 data-tests, hvor vi bevidst indsætter null-bytes, DEL (0x7F), ugyldige continuation-bytes, overlong UTF-8-sekvenser og tilfældige binære mønstre.
- Empty dækkes af tre forskellige tomme filer (0 bytes).

Vores tests dækker hele det krævede input-rum bredt: simple eksempler, forskellige sprog, grænsetilfælde og ugyldige byte-mønstre.
ASCII og empty er meget ligetil og altid klassificeret korrekt. ISO vs. UTF-8 er sværere, fordi mange tegn kan repræsenteres på begge måder. Her kan referenceprogrammet `file(1)` nogle gange være uenig med vores program. Vi har dokumenteret disse forskelle som "false fails" dvs. forskelle der ikke er fejl i forhold til A0, men kun skyldes at referenceprogrammet genkender flere varianter. Data er den mest udfordrende kategori, da den samler alt ugyldigt. Her har vi testet mange kombinationer af ugyldige bytes for at sikre robusthed.

Når `diff` viser forskelle, har vi brugt det til at identificere reelle fejl i vores kode. Eksempelvis afslørede en test med en overlong UTF-8 sekvens, at vores validering skulle strammes op. Andre forskelle skyldes blot, at referenceprogrammet bruger mere detaljerede beskrivelser end vi skal, hvilket er acceptabelt ifølge opgaven. Vi har igennem tests demonstreret både normaltilfælde og

grænsetilfælde. ASCII og empty er testet godt og simpelt. De mest udfordrende områder er at håndtere ugyldige data-sekvenser, hvor referenceprogrammet `file(1)` kan give andre beskrivelser.

Samlet viser testene, at vores program overholder A0-kravene og klassificerer filer korrekt i alle de definerede kategorier.

Konklusion:

Vi har udviklet et *C* script, som kan bruges til at tjekke, om det forventede input i filer matcher det faktiske input. Vores $file.c$ kan genkende de angivne filkodninger og datatyper, herunder tomme filer, ASCII, ISO 8858-1, UTF-8 samt data. Vi har også udvidet vores test-dækning, således at flere tekst datatyper og tegn bliver testet. Generelt havde vi ikke problemer med at verificere det forventede input og faktisk input. En vigtig undtagelse var dog overlappet mellem karakterer, som både eksisterer i ISO og UTF. De giver anledning til en fortolkning af tegnene. I disse tilfælde vinder UTF konflikten og angives som indholdet, med mindre hexadecimaler anvendes.

Teori:

Boolsk logik:

I denne opgave viser vi, at følgende boolske udtryk er korrekt:

$$(A \oplus B)\&A = (A\&\neg B)$$

Vi viser det ved at tegne en sandhedtabel ud fra det angivne udtryk[5]:

A	B	$A \oplus B$	$(A \oplus B)\ \&A$	$\neg B$	$(A\&\neg B)$
0	0	0	0	1	0
0	1	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	0

Tabellen illustrerer, at det boolske udtryk er korrekt, fordi værdierne på venstre side af lighedstegnet $(A \oplus B)\&A$ er identiske med værdierne på højre side: $A\&\neg B$.

$A \oplus B$ betyder eklusiv eller, og den returnerer (1) *true*, hvis én af værdierne er *true*. [6]

Bitwise logiske operatører og repræsentation:

- Spørgsmål: Given an integer value int x, give a C expression that returns the value multiplied by 8.
  Svar: Bitvis venstreskifte-operator: $x \ll 3$ Operation er valgt for at gange med 8. Hvis man vil gange med 2^n, så svarer det til at rykke bits n positioner mod venstre. I dette tilfæde vil vi gerne gange med 8, hvilket er 3: $2^3 = 8$. Dermed anvender vi $x \ll 3$ til at gange med 8.[7]

- Spørgsmål: Given an integer value int x, give a C expression that returns true if the value is equal to 6.
  Svar: $!(x \oplus 6)$
  Den returnerer *true* når $x = 6$. Det sker gennem to trin.
  Det første er, at bitesekvensen for x og bitesekvensen for 6 sammenholdes ud fra sandhedstabellen med x $\oplus$6. Dermed bliver resultatet 0, da bitesekvenserne er identiske.
  Det næste trin er NOT-operatør. I C betrages 0 som værende falsk, mens 1 er sand.[8] Idet bitesekvensen vil retunere 0 (falsk), så anvendes ! for at konvertere falsk til sandt. Udtrykket er korrekt, da de to værdier i parantesen returnere sand, hvis x og 6 er ens. [9]

[5] `https://www.geeksforgeeks.org/electronics-engineering/truth-table/`
[6] COD, s. 95-97
[7] `https://chatgpt.com/share/68c0106c-2cf0-800c-acee-02a41f4b1277`
[8] https://chatgpt.com/share/68cd50e9-fcc8-800c-9c9a-cb6581a40795
[9] `https://chatgpt.com/share/68c0106c-2cf0-800c-acee-02a41f4b1277`

- Spørgsmål: Given an integer value int x, give a C expression that returns true if the value is less than or equal to 0.
  Svar: $(x \gg (\text{sizeof(int)} \times 8 - 1)) \mid !x$
  Her kombineres to tjeks og returnerer *true*, hvis værdien er ≤ 0.
  Her udnyttes, at man også kan lave højreskifte med 2^n, hvilket svarer til at rykke bits n positioner mod højre. I udtrykket
  $(x \gg (\text{sizeof(int)} \times 8 - 1))$ rykkes x det antal bits, som er i en int -1. Det skyldes, at vi er interesseret i at tjekke, om den mest signifikante bit er 1 eller 0. Hvis den er 1, så er tallet negativ. Udtrykket forudsætter, at *int* er signed, så x kan være negativ og MSB (most significant bit) [10] kan være 1. Når vi foretager højre skift på en signed int, så bliver udtrykket 11111111 11111111 11111111 11111111, hvilket er -1 i decimal. Dermed tjekker udtrykket, om x er negativ.
  I næste tjek undersøges, om x er 0. Hvis det er sandt, at x er 0, så returnere negationen af x: 1. Det skyldes, at 0 i *C* betragtes som falsk, så negationen returnerer sand, fordi x er nul. Dermed returnerer vi sandhedsværdien af, om x < 0 eller om x = 0.[11]

- Spørgsmål: Given integer values int x and int y, give a C expression that returns true if the value of x and y are different. Svar: $!!(x \oplus y)$
  Her anvendes to dele. Det første er, om x og y er ens. $\oplus$ returnerer 0, hvis de er ens og 1 eller derover, hvis de er forskellige.
  Næste del er at anvende to negationer. Hvis de er forskellige, så anvendes den første negation til at konvertere !(værdien af x $\oplus$y) til 0. Altså falsk. Den næste negation bruges til at konvertere falsk til sand. "!!"er blevet skrevet for at gå fra bit tjek til sand/falsk værdi, så vi kan returnere en bool. !!(x $\oplus$y) evaluere om x og y er forskellige. Hvis det er tilfældet så evalueres alle ikke 0-værdier til sande, da de er forskellige. [12]

Flydende-komma repræsentation:

Forklar fordelene ved at have denormaliserede tal i IEEE 754 flydende-komma formatet.

Denormaliserede tal i IEEE 754 flydende-komma formatet giver afrundingsmetoder, som gør tilnærmelser mere præcise. Hvis et tal bliver tilstrækkeligt lille og tæt på 0, skal man bruge denormaliserede tal frem for normaliserede tal. En fordel er derfor, at sådanne tal ikke automatisk bliver sat til 0. En anden fordel er, at man kan få mere præcise resultater ud fra floating-point operationer. Denormaliserede tal udfylder hullet mellem det mindste normaliserede tal og 0 [13]. Det gør det muligt at repræsentere tal, som er mindre end 1×2^{-126}, som netop er grænsen for normaliserede tal. Dermed får vi et yderligere spænd, som kan eksemplificeres i decimaltal: Normalisert mindste værdi:

[10] https://en.wikipedia.org/wiki/Bit_numbering
[11] https://chatgpt.com/share/68c0106c-2cf0-800c-acee-02a41f4b1277
[12] https://chatgpt.com/share/68c0106c-2cf0-800c-acee-02a41f4b1277
[13] COD, s. 233

$1.17549435 \times 10^{-38}$ og denormaliseret mindste værdi: 1.4013×10^{-45}. Dette er yderst relevant, når man skal repræsentere små størrelser såsom Plancks længde som er 1.616×10^{-35}. [1415]. Hvis et denormaliseret tal bliver mindre end 1.4013×10^{-45}, så sker der underflow.[16] Bevægelsen mod underflow er gradvis modsat normalisrede tal. Det skyldes, at mantissen i denormalisrede tal starter med 0.fraction frem for normaliseret, hvor den starter med 1.fraction. Det gør, at mantissen kan blive mindre end 1 og tilnærmelsen kan derfor ske gradvis frem for i ryk ved de normaliserede tal. Derudover har IEEE 754 fire afrundingsmetoder: afrunding opad mod $+\infty$, afrunding nedad mod $-\infty$, afkortning (truncate), og afrunding til det nærmeste lige tal [17].

Sprog niveauer:

- **Hvis alt alligevel kører i maskinkode, hvorfor programmerer vi så i højere niveau-sprog som C?**
  Vi har brug for højere niveau-sprog som C og Python, fordi maskinkode er mindre læsbar, da det enten består af 0 eller 1. Man kan også lettere debugge og vedligeholde programmer samt udføre instruktioner vha. sådanne programmeringssprog. Det højere abstraktionsniveau i sådanne sprog gør koden både mere forståelig og lettere at arbejde med, da detaljerne (i maskinkoden) bliver gemt væk [18]. Det ville sandsynligvis også være mere tidskrævende og sværere, hvis programmører var tvunget til at anvende maskinkode.

- **Hvis højere niveau-sprog som C er så gode, hvorfor opfinder vi så ikke en CPU, der bruger det som sit instruktionssæt i stedet?**
  Fordi CPUer kun kan udføre maskinkode med binære instruktioner (0 eller 1). Man kan netop anvende høj-niveau sprog som *Python* og *C* til at skjule disse detaljer og i stedet fokusere på konkrete instruktioner og programmer. Det er altså mere effektivt, at lade CPUen styre beregninger og håndtere samt udføre instruktioner [19]. Endelig ville det sikkert også være meget besværligt at implementere sådan en CPU.

Kildeliste:

- Computer Organization and Design [COD], RISC-V Edition: The Hardware Software Interface, David A. Hennessy and John L. Patterson, 2nd edition, Morgan Kaufmann, ISBN: 978-0128203316

- GeeksForGeeks. `https://www.geeksforgeeks.org/linux-unix/file-command-in-linux-with-examples/`

[14] `https://chatgpt.com/share/68cc5ef4-e8ec-800c-ba5f-270efd4b4339`
[15] `https://dk.facts.net/natur/univers/27-fakta-om-planck-skala/`
[16] `https://chatgpt.com/share/68cd6b2c-d238-800c-a578-91869802807d`
[17] COD, s. 230
[18] COD, s. 171-173
[19] `https://www.geeksforgeeks.org/computer-science-fundamentals/central-processing-unit-cpu/`

- GeeksForGeeks. https://www.geeksforgeeks.org/computer-science-fundamentals/central-processing-unit-cpu/
- GeeksForGeeks. https://www.geeksforgeeks.org/electronics-engineering/truth-table/
- da.101-help. https://da.101-help.com/sadan-aendres-standard-tegnkodning-i-notesblok-pa-windows-11-10-15e98bec45/
- Wikipedia. https://en.wikipedia.org/wiki/UTF-8
- Wikipedia. https://en.wikipedia.org/wiki/ISO/IEC_8859-1
- GeeksForGeeks. https://www.geeksforgeeks.org/c/error-handling-in-c/
- FreeCodeCamp. https://www.freecodecamp.org/news/bash-scripting-tutorial-linux-shell-script-and-command-line-for-beginners/
- StackOverflow. https://stackoverflow.com/questions/6280055/how-do-i-check-if-a-variable-is-of-a-certain-type-compare-two-types-in-c?utm_source=chatgpt.com
- cplusplus.com. https://cplusplus.com/reference/cstdio/
- GeeksForGeeks. https://www.geeksforgeeks.org/c/bool-in-c/
- cplusplus.com. https://cplusplus.com/reference/cstring/strerror/
- ASCII Table. https://www.asciitable.com/
- cplusplus.com. https://cplusplus.com/reference/cstdio/ungetc/
- cplusplus.com. https://cplusplus.com/reference/cstdio/fgetc/
- cplusplus.com. https://cplusplus.com/reference/cstdio/ferror/?kw=ferror
- cplusplus.com. https://cplusplus.com/reference/cstdio/fread/
- cplusplus.com. https://cplusplus.com/reference/cstdio/size_t/
- AdamTheAutomator. https://adamtheautomator.com/bash-file-test/?utm_source=chatgpt.com
- StackOverflow. https://stackoverflow.com/questions/5519315/a-test-data-set-for-auto-testing-utf-8-string-validator?utm_source=chatgpt.com
- ASCII Table. https://www.asciitable.com/
- UTF-8 Chartable. https://www.utf8-chartable.de/

- Unicode Charts. `https://www.unicode.org/charts/`
- Wikipedia. `https://en.wikipedia.org/wiki/ISO/IEC_8859`
- Wikipedia. `https://en.wikipedia.org/wiki/ISO/IEC_8859-1`
- MGK25 UTF-8 Test Examples. `https://www.cl.cam.ac.uk/~mgk25/ucs/examples/UTF-8-test.txt`
- Wikipedia. `https://en.wikipedia.org/wiki/Bit_numbering`
- Planck-skala. `https://dk.facts.net/natur/univers/27-fakta-om-planck-skala/`

Declaration of using generative AI tools (for students)

☒ I/we have used generative AI as an aid/tool (please tick)

☐ I/we have **NOT** used generative AI as an aid/tool (please tick)

If generative AI is permitted in the exam, but you havent used it in your exam paper, you just need to tick the box stating that you have not used GAI. You dont have to fill in the rest.

List which GAI tools you have used and include the link to the platform (if possible):

Example: [Copilot with enterprise data protection (UCPH license), `https://copilot.microsoft.com`] [1em]

- Chat-GPT:
- Bit-wise-and-tjek: `https://chatgpt.com/share/68c91e1f-08f0-800c-88af-540044f49de7`
- Udarbejde af koden: `https://chatgpt.com/share/68bf1b3e-4e8c-800c-ab71-9cb42d62db49`
- Læsning af fil: `https://chatgpt.com/share/68c92229-d71c-800c-951d-f3291f6de704`
- Bitwise logiske operationer: `https://chatgpt.com/share/68c0106c-2cf0-800c-acee-02a41f4b1277`
- Mindste normaliserede tal (IEEE 754): `https://chatgpt.com/share/68cc5ef4-e8ec-800c-ba5f-270efd4b4339`
- UTF-8 som standart: `https://chatgpt.com/c/68c9afb3-0a2c-832b-97e7-5323dfe05508`
- 0 I C: `https://chatgpt.com/share/68cd50e9-fcc8-800c-9c9a-cb6581a40795`
- ISO 8859-1 kontroltegn: `https://chatgpt.com/share/68c930f3-4a7c-800c-91f2-a1b6e2b2c04d`
- UTF-8 vs. ISO 8859-1: `https://chatgpt.com/share/68c96bcc-1028-800c-89a8-e78be53c69f1`
- Underflow `https://chatgpt.com/share/68cd6b2c-d238-800c-a578-91869802807d`

Describe how generative AI has been used in the exam paper:

1. Purpose (what did you use the tool for?)
   Vi har anvendt chatGPT som en sparringspartner gennem udarbejdelse af koden og test. Ydermere har den bistået os i besvarelsen af teorispørgsmålene.
   Vi har anvendt AI som støtteværktøj i forbindelse med vores testafsnit. Formålet var at få inspiration til, hvordan man kunne udforme testfiler (f.eks. med printf og escape-sekvenser), hvordan man kunne dække grænsetilfælde, og hvordan man bedst kunne beskrive testdækning i rapporten. Vi har brugt andre kilder til at finde selve tegn og sekvenser som er blevet brugt. Men AI har været god til at finde kombinationer som kunne teste edge cases.

2. Work phase (when in the process did you use GAI?) Vi har særligt anvendt chatGPT til tre ting.
   Den første er, at vi anvendte den til at finde en løsning på, hvordan vi kunne tjekke for UTF-8. Det skyldes, at vi var usikre på, hvordan man kunne udføre et bit-wise tjek.
   Den anden er i forbindelse med forståelse af testscriptet samt udarbejdelse af test, så fik testet vores løsning.
   Den sidste til at besvarelse af teoriafsnittet. Her har vi særligt brugt den til at forstå de forskellige led i repræsentationen af fx $!(x \oplus 6)$
   samt denormaliserede tal.
   AI blev brugt under udviklingen af vores tests og i den senere dokumentationsfase, hvor vi skrev rapporten. Når vi var i tvivl om, hvordan vi kunne variere vores testfiler, eller hvilke typer grænsetilfælde der kunne være relevante, blev AI brugt spurgt til råds.

3. What did you do with the output? (including any editing of or continued work on the output) I første del anvendte vi koden direkte i vores løsning og lavede kun enkelte ændringer.
   I anden del
   I den tredje del anvendte vi chatGPT's løsninger på opgaverne. Dernæst anvendte vi den ligeledes i vores videre arbejdsproces med at beskrive, hvad de enkelte led i operationerne resulterede i. Ud fra denne sparring opnåede vi en stor forståelse af, hvorfor det virkede. Det gjorde, at vi kunne give en detaljeret forklaring af, hvorfor løsningen fungerede.
   I forbindelse med denormaliserede tal fik vi en større forståelse for, hvad er denormaliserede tal samt deres snitflade med normaliserede tal. Herudfra kunne vi udarbejde et afsnit, som kunne besvare spørgsmålet.
   Under skrivning af test har vi brugte outputtet som inspiration og baggrundsviden. De konkrete tests er lavet af os selv, og vi har selv skrevet test.sh og testfilerne. AI-output blev tilpasset, redigeret og omskrevet, så det passede til vores egen implementering og opgavens krav.

Please note: Content generated by GAI that is used as a source in the paper requires correct use of quotation marks and source referencing. Read the guidelines from Copenhagen University Library at KUnet.
`https://copilot.microsoft.com/`
`https://kunet.ku.dk/faculty-and-department/copenhagen-university-library/library-access/Pages/AI.aspx`