Решения на задачите (zadachi_01.pdf и zadachi_02.pdf):

1.1

```
cat big_file.* > big_file
```

Обяснение: Командата **split** не прави нищо повече от това да изкопира и разреже байтовата последователност на оригиналния файл на съответния брой парченца (с указания размер) и да ги именува като файлове с имена *big_file.aa*, *big_file.ab*, *big_file.ac*, ..., *big_file.az*, *big_file.ba*,... в тази последователност. За да се възстанови оригиналният файл, тези парченца просто трябва да се наредят едно след друго в правилния ред – т.е. лексикографски. Командата **cat** ни позволява слепването, а правилният ред се осигурява от това, че и **split** генерира лексикографки подредени файлове, и filename expansion генерира лексикографки подредени имена на файлове.

1.2

```
#! /bin/sh
## music_ordering.sh
mkdir MixedMusic/MP3 MixedMusic/OGG MixedMusic/WMA
mv MixedMusic/*.mp3 MixedMusic/MP3/
mv MixedMusic/*.ogg MixedMusic/OGG/
mv MixedMusic/*.wma MixedMusic/WMA/
mv MixedMusic Music
ls Music/MP3/ > Music/MP3/mp3_playlist.m3u
ls Music/OGG/ > Music/OGG/ogg_playlist.m3u
ls Music/WMA/ > Music/WMA/wma_playlist.m3u
cd Music
```

```
ls */* > music_playlist.m3u
cd ..
```

Обяснение: Може да има вариации например с влизане в папката *MixedMusic* и изпълняване на местенията от нея. Също така, ако е възможно разширенията на файловете да не са изцяло с малки букви (а например *.Mp3*, *.WMA*, *.oGg*) filename expansion регулярните изрази трябва да бъдат **`*.[Mm][Pp]3`**, **`*.[Ww][Mm][Aa]`**, **`*.[Oo][Gg][Gg]`**.
Преименуването на файл става чрез командата **`mv`** с цел в същата директория.
Плейлистите във формат *m3u* (обикновения формат, без разширенията му) не представляват нищо повече от изредени пътища до музикалните файлове. Обърнете внимание на разликата в изхода на **`ls DIRNAME/`** и **`ls DIRNAME/*`**. Също така забележете какъв е изходът, ако на **`ls`** подадем име на директория и име на още някакъв файл (в частност друга директория).

1.3

```
#! /bin/sh
## folder_cleaning.sh
cd try
rm *.pyc *.class *.o *.in *.out 2> /dev/null
rm `ls --ignore="*?.?*"` 2> /dev/null
cd ..
```

Обяснение: Файловете, които имат определено разширение, изтриваме лесно. Пренесочваме изхода за грешки, защото filename expansion първо се опитва да се разшири до списък от имена на файлове по указания шаблон, но ако не успее, оставя като резултат указания шаблон – например **.pyc*. Т.е. **`rm`** ще получи аргумент, независимо от това, че няма файлове с имена, завършващи на *.pyc* . И тъй като тогава ще се опита да изтрие файл (**.pyc*), който не съществува, ще изведе съобщение за грешка (**`rm: cannot remove `*.pyc': No such`**

`file or directory`), което няма да означава някаква грешка в нашия скрипт.
За втората част на задачата – изтриване на всички файлове, които "нямат разширение" използваме опцията на **`ls --ignore`**, чрез която указваме филтър-шаблон за това кои файлове в текущата директория да *НЕ* се изведат от командата. Подобна опция е и **`--hide`**, но тя се припокрива от опциите **`-a`** и **`-A`** (т.е. комбинирана с тези опции се игнорира). В нашия случай също е приложима, но ако имахме условието да разглеждаме и скритите файлове, нямаше да върши работа.
Шаблонът, който ползваме е **`*?.?*`**, т.е файл, чието име съдържа **`.`** и поне по един символ около тази точка. Всъщност шаблон за име на файл с разширение е много относително понятие и може да има различни тълкувания. На практика например файловите мениджъри (Nautilus, Dolphin, PCManFM, Thunar и т.н.) считат за основно име на файла всичко преди последната точка, а за негово разширение всичко след нея (което може и да е нищо). Така че, шаблонът ни можеше да бъде и **`*.*`**.
Пренасочването на изхода за грешки правим в случай, че **`ls`** не ни върне нищо, т.е на **`rm`** не подаваме нито един аргумент, което би довело до грешката

```
rm: missing operand
Try `rm --help' for more information.,
```

която отново не е грешка в нашия скрипт.

2.1

```
rm [a-zA-Z]
```

Забележка: Разбира се, това работи само за файлове с име една латинска буква. Ако искахме да важи и за една кирилска буква от българската азбука, трябваше да бъде

```
rm [a-zA-Za-яA-Я]
```

например. Но на практика кирилските букви изобщо не са от **`а`** до **`я`**, латинските букви не са

само от **a** до **z** и файлът ни може да има име **Æ**, **ù**, **ђ**, **Ќ**, защото сме щастливи потребители на ОС, която не ограничава имената на файловете. А пък и азбуките по света са десетки – гръцка, арменска, арабска, индийска, еврейска и т.н., като прибавим и източните йероглифи, задачата става малко по-завъртяна. Все пак целта на *тази* задача беше лека мозъчна загрявка.

2.2

a)

```
rm .*.log
```

b)

```
rm .*.*
```

Обяснение: Както казахме filename expansion не се разширява до имена на файлове, започващи с `.` (скрити файлове). Едно единствено изключение: когато шаблонът ни започва с `..` В подточка b) може да дефинираме и по-подробен израз за казуса с разширението на файла, но както казахме, на практика никой не влиза в такива детайли. Все пак две вариации по тази тема са `.*.?*` и `.*.[^.]*`.

2.3

a)

```
ls *...*
```

Забележка: В bash filename expansion `.` не е специален символ.

b)

```
ls -a | grep '\.\.\.'
```

Обяснение: Разбира се тук може да си поиграем с filename expansion шаблони. Може даже и да стигнем до решението. Обаче не е нужно при наличието на **grep**.
С опцията **-a** на **ls** извеждаме всички файлове, включително и скритите. Може да ползваме и опцията **-A**, която има същото значение, с изключение на това, че не извежда файловете **.** и **..** (които описват съответно текущата и родителската директории). Изведеното от **ls** подаваме като стандартен вход на **grep.** Забележка: в някои случаи, когато извикваме **ls**, резултатът ни се изписва не като по едно име на ред, а например в колонки, което създава илюзия, че **grep**, който търси шаблона поредово, може да се счупи. Но на практика всичко работи правилно:

```
mara@OVNI:~/tests$ ls
a...  a........  a...b  abcde  bcdesf
mara@OVNI:~/tests$ ls | grep ^a
a...
a........
a...b
abcde
mara@OVNI:~/tests$
```

Обърнете внимание, обаче, че с **echo *** това не е така:

```
mara@OVNI:~/tests$ echo *
a... a........ a...b abcde bcdesf
mara@OVNI:~/tests$ echo * | grep ^a
a... a........ a...b abcde bcdesf
mara@OVNI:~/tests$
```

grep трябва да търси **...** на всеки прочетен ред. Само че в регулярните изрази, които той използва, **.** е специален символ, затова трябва да бъде екраниран с **** отпред. И тъй като **** се ползва за екраниране и от bash, който parse-ва командата и я анализира, ако напишем само

`grep \.\.\.`, bash ще бъде този, който ще прочете тези **`\`** и ще си каже, че следователно това е знак за екраниране и значението на последващият символ трябва да остане същото, а не да се тълкува по някакво специално значение, което може да има. Т.е. bash ще превърне **`\.\.\.`** в **`...`** и в този вид ще го подаде на **`grep`** като шаблон. За да получи **`grep`** от bash правилния шаблон трябва или да го оградим в единични или двойни кавички (и двата вида отменят специалното значение на **`\`** за *bash*), или командата да бъде **`grep \\.\\.\\.`**, като след минаването си през bash шаблонът да остане **`\.\.\.`**. Ние сме избрали варианта с кавичките.

2.4

a)

```
cat porn | grep "xxl" | wc -l
```

или

```
cat porn | grep -c "xxl"
```

или

```
grep "xxl" porn | wc -l
```

или

```
grep -c "xxl" porn
```

Според това доколко сме прочели ***`man grep`***. Не забравяйте, че **`grep`** чете по *редове* и търси за шаблона в даден *ред*.

b)

```
n=0
for word in `cat porn`
do
```

```
    [ $word == "xxl" ] && n=$((n+1))
done
echo $n
```

Обяснение: Итерираме по всяка дума във файла и тестваме дали тя е "*xxl*". Ако това е така, увеличаваме **n** с **1**.
Изходът на командата **cat porn** използваме за списък от думи, по който да итерираме. Bash дели този списък на думи по стандартния начин – с разделителите, описани в Internal Field Separator (обикновено интервал, табулация и нов ред).
Използвали сме командата **test** във варианта **[EXPRESSION]**. Аритметичният израз може да бъде и **n=$[$n+1]** или **: $((n+=1))**. В изразът, който сме дали в решението, може да няма **$** пред вътрешния **n**, защото този синтаксис позволява изрази, по-приличащи на езика C.
Внимание: Ако напишем само **$((n+=1))**, резултатът ще се оцени като (име на) команда, която bash ще се опита да изпълни! Ако няма такава команда, ще изведе грешка. Но ако има, ще я изпълни и тази команда може да се окаже не съвсем безвредна. Затова оценяването като команда се избягва, като се напише преди това празната команда **:**, т.е **: $((n+=1))**. Празната команда не прави нищо, освен да expand-не аргументите си (в случая нашия аритметичен израз).

c)

```
n=0
for word in `cat porn`
do
    ( echo $word | grep "xxl" > /dev/null ) && : $((n+=1))
done
echo $n
```

или

```
n=0
for word in `cat porn`
do
    grep  "xxl" <<< $word > /dev/null && : $((n+=1))
done
echo $n
```

Обяснение: За да проверим дали дадена дума съдържа искания низ, използваме **grep**, на чийто стандартен вход подаваме един ред, съдържащ тази дума. Правим го или с pipe след **echo $word**, или с пренасочване на стандартния вход към низа с тази дума (не забравяйте, че **<<<** е опция само в bash, не и в други командни обвивки, т.е. универсалният вариант използва **<<** и някакъв ограничител). Пренасочваме стандартния изход от **grep**, за да не "загрозяваме" скрипта.

2.5

a)

```
ls try/a.out | wc -l
```

b)

```
ls try/*/a.out | wc -l
```

```
c)
cd try
ls * | grep -c '^a\.out$'
cd ..
```

Обяснение: разбира се може просто да съберем числата от a) и b). Но ние се опитахме да се правим на умни. Дори твърдяхме, че решението е **ls try/* | grep -c 'a.out'**. Къде

сбъркахме:

1. Огромна грешка в това, че забравихме, че точката е специален символ в **`grep`**. На един такъв шаблон отговаря както *a.out*, така и *about*, *mara.out*, *a.output*.Трябва да се екранира. Това води до **`ls try/* | grep 'a\.out'`**.
2. Обаче все още сме много далеч от истината. Трябва да филтрираме и *mara.out*, *a.output*. В същото време да не забравяме, че **`try/*`** ще се разшири евентуално и до *try/a.out*, който трябва да броим. Тръгваме да четем ***man grep*** за подходящ регулярен израз.
3. Блъскаме си главата и научаваме 1-2 неща за регулярните изрази на **`grep`**. Ако сме късметлии, може и да сме уцелили нещо. В нашия случай, обаче, след много неуспешни групирания, звездички, въпросителни и най-различни наклонени чертички, решаваме да почнем отначало.
4. И се сещаме, че ако идем в директорията *try* и направим **`ls *`**, то всички файлове, които търсим, ще излязат само с малкото си име. Това означава, че регулярен израз **`'^a\.out$'`** е нашето решение.
5. Ако все още имаме ентусиазъм, може да добавим поддръжка за търсене на файлове *a.out* и в скритите поддиректории. Само ако все още имаме ентусиазъм...

2.6

a)

```
who | grep '^student\>' && write student <<< “hello”
```

b)

```
who | grep '^student\>' && write student < messagefile
```

c)

```
who | grep '^student\>' && write student <<END
```

```
Your message
here
END
```

Забележка: Използваме шаблон **'^student\>'**, за да укажем изрично, че това трябва да бъде първата дума в реда. **^** означава, че редът започва там. Специалният символ **\>** пък означава край на дума, т.е ако в момента в системата има логнат *student1*, но *student* не е логнат, без този символ **grep** ще мине успешно и чак на **write student** ще получим грешка, която няма да знаем откъде идва. Заради **** слагаме целия шаблон в кавички.

2.7

а)

```
cd junk
for filename in * .[^.]*
do
    [ ! -L $filename -a -f $filename ] && [ `cat $filename | wc -c` -eq 0 ]
&& echo $file >> ../junk_empty
done
cd ..
```

Обяснение: ***** и **.[^.]*** в общия случай отговаря на изведеното от **ls -A**.
По условие говорим само за обикновени файлове, затова това е и първата проверка. Не искаме файлът да бъде символна връзка, защото **-f** тестът ще я последва, а ако файлът, към който тя сочи, е празен, ще изтрием непразната символна връзка. **[[! -L $filename && -f $filename]]** също е възможен синтаксис.
Размерът в байтове на файла вземаме като **`cat $filename | wc -c`**, но може и като **`wc**

-c < "$filename"`**. Във втория случай има " " около **$filename, защото иначе bash дава грешка **ambigious redirect** (двусмислено пренасочване).

b)

```
while read line
do
    rm "try/$line"
done < junk_empty
```

или

```
cat junk_empty | while read line
do
    rm "try/$line"
done
```

Обяснение: Трябва да четем файла ред по ред, защото в имената на файловете може да има интервали (а дори и табулатори и нови редове) и думите, на които ще се раздели **`cat junk_empty`**, няма да бъдат имената на файловете. Четем един ред от стандартния вход чрез **read**, а самия стандартен вход на целия цикъл пренасочваме към файла или правим pipe след **cat**. Напомняме, че при pipe процесите се изпълняват едновременно, а не един след друг, и чрез програмния канал си комуникират.

Една голяма ЗАБЕЛЕЖКА – относно генерирането на списък с имена на файлове:

Вече показахме два начина да направим това – чрез filename expansion шаблон и чрез вземане на изхода от командата **`ls`**. И двата начина, обаче, имат големи недостатъци!!!

- Чрез filename expansion списъкът с файлове никога няма да е празен – ако не намери съвпадения, списъкът няма да се разшири и ще остане само шаблона. Тоест

  ```
  for file in *.pyc
  do
    echo $file
  done
  ```

  изпълнен в папка, в която няма файлове с разширение *.pyc*, ще има изход:

  ```
  *.pyc
  ```

  Когато правим някакви операции над този списък с имена на файлове, това може да доведе до грешки. Трябва да предвиждаме такива случаи. При възможност може да пренасочваме стандартния изход за грешки към */dev/null*, но в случай, че на него могат да се изпишат грешки от друго естество, е по-добре да не го правим.
  В задача 2.7 а) правим тест за съществуването и вида на файла, така че използването на този подход е безопасно.
- Чрез списък, получен като изход от командата **`ls`**, губим файловете, които имат интервал, табулатор или знак за нов ред в името си, тъй като техните имена ще бъдат разделени от bash на отделните думи. Filename expansion ще екранира IFS символите и ще подаде правилните имена на файловете в списъка.
  В задача 1.3 говорим за файлове с програмен код, чиито имена по конвенция са “една” дума (т.е. не съдържат IFS символи). Използването на този подход за изтриването на файловете без разширение е уместно.

Универсално решение на този проблем няма, но човек винаги трябва да се замисля кой начин е по-добър за конкретната нужда. И никога да не изпада в крайността да ползва само единия или само другия начин.