1. Introduction. This is a routine to search through a directory hierarchy, find all the files matching the input specifications, and list them out in order according to size. It is based on **Program 4.7** in W. Richard Stevens wonderful book *Advanced Programming in the UNIX Environment.*

Updated for huge file systems in 2013. Essentially made all integers into long integers to accomodate files of size greater than 2 Gigabytes, and huge directories.

2. This program is written in `WEB`, a preprocessor for C or Pascal. This style of programming is called "Literate Programming." For Further information see the paper *Literate Programming*, by Donald Knuth in *The Computer Journal*, Vol 27, No. 2, 1984; or the book *Weaving a Program: Literate Programming in* `WEB` by Wayne Sewell, Van Nostrand Reinhold, 1989.

3. The following is the top-down structure of all my `CWEB` programs. `CWEB` is just a variant of `WEB` that handles Standard C.

⟨ Global # **include**s 4 ⟩
⟨ Global structures 5 ⟩
⟨ Global variables 6 ⟩
⟨ Functions 9 ⟩
⟨ The main calling routine 7 ⟩

4.

⟨ Global # **include**s 4 ⟩ ≡
```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include "ourhdr.h"
```
See also section 16.

This code is used in section 3.

5. Function type that's called for each filename.

```
#define MAXSYSFILES 2000000
```

⟨ Global structures 5 ⟩ ≡
```
typedef int (Myfunc)(const char *, const struct stat *, int);
static long filesize[MAXSYSFILES], indices[MAXSYSFILES];
static char *filenames[MAXSYSFILES];
```
This code is used in section 3.

6.

⟨ Global variables 6 ⟩ ≡
```
static Myfunc myfunc; static int myftw (char *, Myfunc * ) ; static int dopath ( Myfunc * ) ;
static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;
```
See also sections 13, 15, 18, and 20.

This code is used in section 3.

7. Here is the top calling routine. I recursively descend the directory saving the file names which match the parameters.

```
⟨ The main calling routine 7 ⟩ ≡
  int main(int argc, char *argv[]){ int ret;
      extern int optind; ⟨ Parse input parameters 14 ⟩
      for ( ; optind < argc; optind++) ret = myftw(argv[optind], myfunc);      /* does it all */
      ⟨ Print statistics 8 ⟩
      ⟨ Sort names 17 ⟩
      ⟨ Output data 19 ⟩
      exit(ret); }
```

This code is used in section 3.

8. Here I print out the statistics.

```
⟨ Print statistics 8 ⟩ ≡
  if ((ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) ≡ 0) ntot = 1;
  printf("regular␣files␣␣=␣%7ld,␣%5.2f␣%%\n", nreg, nreg * 100.0/ntot);
  printf("directories␣␣␣␣=␣%7ld,␣%5.2f␣%%\n", ndir, ndir * 100.0/ntot);
  printf("block␣special␣␣=␣%7ld,␣%5.2f␣%%\n", nblk, nblk * 100.0/ntot);
  printf("char␣special␣␣␣=␣%7ld,␣%5.2f␣%%\n", nchr, nchr * 100.0/ntot);
  printf("FIFOs␣␣␣␣␣␣␣␣␣␣=␣%7ld,␣%5.2f␣%%\n", nfifo, nfifo * 100.0/ntot);
  printf("symbolic␣links␣=␣%7ld,␣%5.2f␣%%\n", nslink, nslink * 100.0/ntot);
  printf("sockets␣␣␣␣␣␣␣␣=␣%7ld,␣%5.2f␣%%\n", nsock, nsock * 100.0/ntot);
  if (do_uid) printf("\n%s␣files␣=␣%7ld,␣%5.2f␣%%\n", pwd→pw_name, ureg, ureg * 100.0/ntot);
```

This code is used in section 7.

9. The routine *myftw*( ) is lifted whole from *APUE*. Descend through the hierarchy, starting at "pathname". The caller's func() is called for every file.

```
#define FTW_F  1      /* file other than directory */
#define FTW_D  2      /* directory */
#define FTW_DNR  3    /* directory that can't be read */
#define FTW_NS  4     /* file that we can't stat */
```

```
⟨ Functions 9 ⟩ ≡
  static char *fullpath;      /* contains full pathname for every file */
  static int      /* we return whatever func() returns */
  myftw(char *pathname, Myfunc * func)
  {
    fullpath = path_alloc(Λ);      /* malloc's for PATH_MAX + 1 bytes */      /* (Prog pathalloc) */
    strcpy(fullpath, pathname);      /* initialize fullpath */
    return (dopath(func));
  }
```

See also sections 10, 11, and 21.

This code is used in section 3.

10. Function *dopath*( ). Descend through the hierarchy, starting at "fullpath". If "fullpath" is anything other than a directory, we lstat() it, call func(), and return. For a directory, we call ourself recursively for each name in the directory.

```
⟨ Functions 9 ⟩ +≡
  static int      /* we return whatever func() returns */
  dopath(Myfunc * func)
  {
    struct stat statbuf;
    struct dirent *dirp;
    DIR * dp;
    int ret;
    char *ptr;
    if (lstat(fullpath, &statbuf) < 0) return (func(fullpath, &statbuf, FTW_NS));      /* stat error */
    if (S_ISDIR(statbuf.st_mode) ≡ 0) {
      if (debug) {
        printf("debug>fullpath␣=␣%s.\n", fullpath);
      }
      return (func(fullpath, &statbuf, FTW_F));      /* not a directory */
    }      /* * It's a directory. First call func() for the directory, * then process each filename in the
          directory. */
    if (debug) {
      printf("debug>fullpath␣=␣%s.\n", fullpath);
    }
    if ((ret = func(fullpath, &statbuf, FTW_D)) ≠ 0) return (ret);
    ptr = fullpath + strlen(fullpath);      /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;
    if ((dp = opendir(fullpath)) ≡ Λ) return (func(fullpath, &statbuf, FTW_DNR));
        /* can't read directory */
    while ((dirp = readdir(dp)) ≠ Λ) {
      if (strcmp(dirp→d_name, ".") ≡ 0 ∨ strcmp(dirp→d_name, "..") ≡ 0) continue;
          /* ignore dot and dot-dot */
      strcpy(ptr, dirp→d_name);      /* append name after slash */
      if ((ret = dopath(func)) ≠ 0)      /* recursive */
        break;      /* time to leave */
    }
    ptr[−1] = 0;      /* erase everything from slash onwards */
    if (closedir(dp) < 0) err_ret("can't␣close␣directory␣%s", fullpath);
    return (ret);
  }
```

11. This is my function.

```
⟨ Functions 9 ⟩ +≡
  static int myfunc(const char *pathname, const struct stat *statptr, int type){ switch (type) { case
          FTW_F: switch (statptr→st_mode & S_IFMT) {
    case S_IFREG: nreg++; ⟨ Process regular file 12 ⟩
      break;
    case S_IFBLK: nblk++;
      break;
    case S_IFCHR: nchr++;
      break;
    case S_IFIFO: nfifo++;
      break;
    case S_IFLNK: nslink++;
      break;
    case S_IFSOCK: nsock++;
      break;
    case S_IFDIR: err_dump("for␣S_IFDIR␣for␣%s", pathname);
        /* directories should have type = FTW_D */
      } break;
    case FTW_D: ndir++;
      break;
    case FTW_DNR: err_ret("can't␣read␣directory␣%s", pathname);
      break;
    case FTW_NS: err_ret("stat␣error␣for␣%s", pathname);
      break;
    default: err_dump("unknown␣type␣%d␣for␣pathname␣%s", type, pathname); } return (0); }
```

12.

```
⟨ Process regular file 12 ⟩ ≡
  if ((uid ≡ statptr→st_uid) ∨ (¬do_uid)) {
    ureg++;
    filenames[ureg] = malloc(strlen(pathname) + 1);
    strcpy(filenames[ureg], pathname);
    filesize[ureg] = statptr→st_size;
    if (debug) {
      printf("Debug>␣Process␣regular␣file.\n");
      printf("Debug>␣Count␣is␣%ld.\n", ureg);
      printf("Debug>␣Filename␣is␣%s.\n", filenames[ureg]);
      printf("Debug>␣Filesize␣is␣%ld.\n", filesize[ureg]);
    }
  }
```

This code is used in section 11.

13. Let me declare the new variables used above.

```
⟨ Global variables 6 ⟩ +≡
  long ureg;
  int debug = 0;
```

14.

```
⟨ Parse input parameters 14 ⟩ ≡
  yrcnt = 50;
  while ((c = getopt(argc, argv, "n:u:")) ≠ EOF) {
    switch (c) {
    case 'n': yrcnt = atoi(optarg);
      break;
    case 'u': pwd = getpwnam(optarg);
      if (pwd ≡ Λ) {
        fprintf(stderr, "Error: username %s not found.\n", optarg);
        exit(2);
      }
      else {
        uid = pwd→pw_uid;
        do_uid = TRUE;
      }
      break;
    case '?': errflg++;
    }
  }
  if (optind ≥ argc) errflg++;
  if (errflg) {
    err_quit("usage: dirsort [-n <#>] [-u <username>] path1...");
  }
```

This code is used in section 7.

15. Now let me declare all the new variables used above.

```
#define TRUE 1
#define FALSE 0
⟨ Global variables 6 ⟩ +≡
  extern char *optarg;
  int errflg, yrcnt, c;
  int do_uid = FALSE;
  uid_t uid;
  struct passwd *pwd;
```

16.

```
⟨ Global # includes 4 ⟩ +≡
#include <pwd.h>
```

17.

```
⟨ Sort names 17 ⟩ ≡
#if defined (DEBUG)
  printf ("Sorting␣file␣sizes.\n");
#endif
  if (ureg > 1) indexx(ureg, filesize, indices);
#if defined (DEBUG)
  for (j = 1; j ≤ ureg; j++) {
    printf ("%s,␣indices[%ld]␣=␣%ld\n", filenames[j], j, indices[j]);
  }
#endif
```

This code is used in section 7.

18.

```
⟨ Global variables 6 ⟩ +≡
  long j;
```

19. Modified to output GB, MB, or KB, depending on file size.

```
⟨ Output data 19 ⟩ ≡
#if defined (DEBUG)
  printf ("Outputting␣data.\n");
#endif
  if (yrcnt > ureg) yrcnt = ureg;
  for (j = ureg; j > ureg − yrcnt; j−−) {
    tsize = filesize[indices[j]];
    if (tsize > 1000000000) {
      fsize = tsize/1000000000;
      fprintf (stdout, "%3ld.␣(%4g␣GB)␣\t␣%s␣\n", (ureg − j + 1), fsize, filenames[indices[j]]);
    }
    else if (tsize > 1000000) {
      fsize = tsize/1000000;
      fprintf (stdout, "%3ld.␣(%4g␣MB)␣\t␣%s␣\n", (ureg − j + 1), fsize, filenames[indices[j]]);
    }
    else if (tsize > 1000) {
      fsize = tsize/1000;
      fprintf (stdout, "%3ld.␣(%4g␣KB)␣\t␣%s␣\n", (ureg − j + 1), fsize, filenames[indices[j]]);
    }
    else {
      fprintf (stdout, "%3ld.␣(%4ld␣bytes)␣\t␣%s␣\n", (ureg − j + 1), filesize[indices[j]],
          filenames[indices[j]]);
    }
  }
```

This code is used in section 7.

20.

```
⟨ Global variables 6 ⟩ +≡
  long tsize;
  double fsize;
```

21. Sort the indices of an array. Lifted from *Numerical Recipes*.

```
⟨ Functions 9 ⟩ +≡
  void indexx(n, arrin, indx)
      long n, indx[];
      long arrin[];
  {
    long l, j, ir, indxt, i;
    long q;
    for (j = 1; j ≤ n; j++) indx[j] = j;
    l = (n ≫ 1) + 1;
    ir = n;
    for ( ; ; ) {
      if (l > 1) q = arrin[(indxt = indx[−−l])];
      else {
        q = arrin[(indxt = indx[ir])];
        indx[ir] = indx[1];
        if (−−ir ≡ 1) {
          indx[1] = indxt;
          return;
        }
      }
      i = l;
      j = l ≪ 1;
      while (j ≤ ir) {
        if (j < ir ∧ arrin[indx[j]] < arrin[indx[j + 1]]) j++;
        if (q < arrin[indx[j]]) {
          indx[i] = indx[j];
          j += (i = j);
        }
        else j = ir + 1;
      }
      indx[i] = indxt;
    }
  }
```

22. Index.

⟨ Functions 9, 10, 11, 21 ⟩ Used in section 3.
⟨ Global structures 5 ⟩ Used in section 3.
⟨ Global variables 6, 13, 15, 18, 20 ⟩ Used in section 3.
⟨ Global # **include**s 4, 16 ⟩ Used in section 3.
⟨ Output data 19 ⟩ Used in section 7.
⟨ Parse input parameters 14 ⟩ Used in section 7.
⟨ Print statistics 8 ⟩ Used in section 7.
⟨ Process regular file 12 ⟩ Used in section 11.
⟨ Sort names 17 ⟩ Used in section 7.
⟨ The main calling routine 7 ⟩ Used in section 3.

Directory Sort

(Version 1.2, March 2014
(Ansi C Version)

This documentation last produced June 28, 2017.