• **How to find me**
  - bwk@cs, www.cs.princeton.edu/~bwk
  - 311 CS Building
  - 609-258-2089 (but email is always better)

• **TA's:**
  - Matvey Arye (arye), Tom Jablin (tjablin), Nick Johnson (npjohnso)

• **Today**
  - course overview
  - administrative stuff
  - regular expressions and grep

• **Check out the course web page (CS, not Blackboard!)**
  - notes, readings and assignments posted (only) there
    monitor the web page every day
  - Assignment 1 is posted
  - initial project information is posted

• **Do the survey if you haven't already**

---

**Themes**

• **languages and tools**
  - mainstream: C, C++, Java, C#, (Objective-C?), ...
  - scripting: AWK, (Perl?), Python, (PHP?), Javascript, ...
  - programmable tools, application-specific languages
  - frameworks, toolkits, development environments, interface builders
  - debuggers (gdb), source code control (SVN), ...

• **programming**
  - design, prototyping, reuse, components, interfaces, patterns
  - debugging, testing, performance, mechanization
  - portability, standards, style
  - tricks of the trade

• **reality**
  - tradeoffs, compromises, engineering

• **history and culture of programming**

• **etc.**
**Very Tentative Outline**

Feb  2    regular expressions; grep, shell
Feb  9    AWK; testing; project stuff
Feb 16    Perl, Python, PHP
Feb 23    Javascript, Ajax, CGI
Mar  2    frameworks, databases
Mar  9    networks, user interfaces

Mar 15    (spring break)

Mar 23    C++, STL, objects
Mar 30    Java, collections
Apr  6    components: COM, .NET, C#
Apr 13    XML, REST/Atom, JSON
Apr 20    ?
Apr 27    ?

May  5-7   demo days: project presentations

---

**Some Mechanics**

- **prerequisites**
  - C, Unix (COS 217); Java (COS 126, 226)
- **5 programming assignments in first half**
  - posted on course web page Tuesday, due Friday evening 10 days later
  - deadlines matter
- **project in second half (starts earlier!)**
  - groups of 3-5; start identifying potential teammates
  - start thinking about topic
  - deadlines matter
- **monitor the web page**
  - readings for most weeks
  - notes generally posted ahead of time
  - newsgroup for discussion, finding partners, ...
- **class attendance and participation <=> no midterm or final**
  - sporadic unannounced short quizzes are possible
Regular expressions and grep

- **regular expressions**
  - notation
  - mechanization
  - pervasive in Unix tools
  - not in most general-purpose languages
    though common in scripting languages and (some) text editors
  - basic implementation is remarkably simple
  - efficient implementation requires good theory and good practice

- **grep is the prototypical tool**
  - people used to write programs for searching
    (or did it by hand)
  - tools became important
  - tools are not as much in fashion today

Grepl regular expressions

c
- any character matches itself, except for
- **metacharacters** . [ ] ^ $ * \r1r2
- matches r1 followed by r2
. 
- matches any single character
- [. . .] matches one of the characters in set ...
  a set like a-z or 0-9 includes any character in the range
- [^ . . .] matches one of the characters not in set
  a set like a-z or 0-9 includes any char in the range
^ 
- matches beginning of line when ^ begins pattern
  no special meaning elsewhere in pattern
$ 
- matches end of line when $ ends pattern
  no special meaning elsewhere in pattern
* 
- any regular expression followed by * matches 0 or more
\c 
- matches c unless c is ( ) or digit
\( . . . \) 
- tagged regular expression that matches ...
  the matched strings are available as \1, \2, etc.
Examples of matching

```
thing             thing anywhere in string
^thing            thing at beginning of string
thing$            thing at end of string
^thing$           string that contains only thing
^                matches any string, even empty
^$                empty string
.                 non-empty, i.e., at least 1 char
thing.$           thing plus any char at end of string
thing\.$          thing. at end of string
\\thing\\        \thing\ anywhere in string
[tT]hing          thing or Thing anywhere in string
thing[0-9]        thing followed by one digit
thing[^0-9]       thing followed by a non-digit
thing[0-9][^0-9]  thing followed by digit, then non-digit
thing1.*thing2    thing1 then any text then thing2
^thing1.*thing2$  thing1 at beginning and thing2 at end
```

egrep: fancier regular expressions

```
r+                 one or more occurrences of r
r?                 zero or one occurrences of r
r1|r2              r1 or r2
(r)                r (grouping)
grahmart:         r: c . ^ $ [ccc] [^ccc]
r*                 r+   r?
r1 r2              r1 or r2
r1|r2              r1 or r2
(r)                r (grouping)
precedence:        * + ? higher than concatenation, which is higher than |

([0-9]+\.?[0-9]*|\.[0-9]+)([Ee][-+]?[0-9]+)?
```
The grep family

- grep
- egrep
  - fancier regular expressions, trades compile time and space for run time
- fgrep
  - parallel search for many fixed strings
- agrep
  - "approximate" grep: search with errors permitted
- relatives that use similar regular expressions
  - ed original Unix editor
  - sed stream editor
  - vi, emacs, sam, ... editors
  - lex lexical analyzer generator
  - awk, perl, python, ... all scripting languages
  - Java, C# ... libraries in mainstream languages
- simpler variants
  - filename "wild cards" in Unix and other shells
  - "LIKE" operator in Visual Basic, SQL, etc.

Basic grep algorithm

while (get a line)
  if match(regexpr, line)
    print line

- (perhaps) compile regexpr into an internal representation suitable for efficient matching
- match() slides the regexpr along the input line,
  looking for a match at each point
Grep (TPOP, p 226)

/* grep: search for regexp in file */
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}

Match anywhere on a line

* look for match at each position of text in turn

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
Match starting at current position

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

• follow the easy case first: no metacharacters
• note that this is recursive
  – maximum depth: one level for each regexpr character that matches

Matching * (repetitions)

• matchstar() called to match c*...
• matches if rest of regexpr matches rest of input
  – null matches require test at the bottom

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}

• finds the leftmost shortest match
  – just right for pattern matching in grep
  – NOT usually what we want in a text editor
    null matches are surprising and rarely desired
Profiling: where does the time go

- measure how long each function takes:

  gcc -pg x.c
  a.out
  gprof

  - display is very flaky

- count number of times each line is executed:

  gcc -fprofile-arcs -ftest-coverage x.c
  a.out
  gcov x.c
  cat x.c.gcov

Statement frequency counts

$ gcc -fprofile-arcs -ftest-coverage grep.c; a.out ../bib >foo;
$ gcov grep.c; cat grep.c.gcov

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
4360969 {  
4360969   if (regexp[0] == '\0')  
1326     return 1;
4359643   if (regexp[1] == '*')  
####     return matchstar(regexp[0], regexp+2, text);  
4359643   if (regexp[0] == '$' && regexp[1] == '\0')  
####     return *text == '\0';  
4359643   if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))  
1326     return matchhere(regexp+1, text+1);
4358317   return 0;
}

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
31102 {  
31102   if (regexp[0] == '^')  
4359643     return matchhere(regexp+1, text);
4359643   do {    /* must look even if string is empty */  
4359643     if (matchhere(regexp, text))  
1326       return 1;
4358317   } while (*text++ != '\0');
29776   return 0;
}
How to make grep faster

- use optimization (cc -O)
- change compilers
- code tuning
  - e.g., match calls matchhere many times
  - even though most of them must necessarily fail
  - because the target string doesn't contain the first character of the pattern
- algorithm changes

Code tuning variant

- checks whether target contains first character of pattern before calling matchhere, unless it is x*

```c
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    char *p;
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    if (regexp[0] != '\'0' && regexp[0] != '.'
        && regexp[1] != '*')
        if ((p=strchr(text, regexp[0])) == NULL)
            return 0;
    do {    /* must look even if string is empty */
        if (matchhere(regexp, p))
            return 1;
    } while (*p++ != '\'0');
    return 0;
}
```

- is this faster?
Statement frequencies after change

```c
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

```c
int match(char *regexp, char *text)
{
    char *p = text;
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    if (regexp[0] != '\0' && regexp[0] != '.' && regexp[1]!='*')
        if ((p=strchr(text, regexp[0])) == NULL)
            return 0;
    do {    /* must look even if string is empty */
        if (matchhere(regexp, p))
            return 1;
    } while (*p++ != '\0');
    return 0;
}
```

Simple grep algorithm

• best for short simple patterns
  - e.g., grep foo *.[ch]
  - most use is like this
  - reflects use in text editor for a small machine

• limitations
  - tries the pattern at each possible starting point
    e.g., look for aaaaab in aaaa...aaaab
    potentially O(mn) for pattern of length m
  - complicated patterns (.* .* .*) require backup
    potentially exponential
  - can't do some things, like alternation (OR)

• this leads to extensions and new algorithms
  - egrep complicated patterns, alternation
  - fgrep lots of simple patterns in parallel
  - boyer-moore long simple patterns
  - agrep approximate matches
Finite state machines/finite automata

- finite state machine
  - a set of states
  - an alphabet (e.g., ascii)
  - transition rules: current state & input char -> new state
  - a start state
  - a set of final "accepting" states

- regular expressions are equivalent to finite state machines
  - can go from one to the other mechanically

- \( ab^*c \)

- \( a^n b^n \), if \( n < 4 \)
  - can't count: can't handle arbitrary \( n \) in a fixed number of states
  - can't do palindromes: no memory

Non-deterministic finite automata (NDFA)

RE: \( .*ab.*abab \)

FSM: 0 1 2 3 4 5 6

input: x x a b a b a a b a b

state after: 0 0 1 2 3 4 5 ?

diff seq: 0 0 1 2 2 2 2 3 4 5 6

- if the machine could guess right every time, it would match properly
  - avoids "backing up", decides about each character the first time it's seen
- a NDFA matches an input if there is any possible path from start state to a final state.
- it rejects/does not match if there is no path from the start state to a final state.
- how do we make a machine that's always lucky?
  - make a deterministic finite automaton that simulates the NDFA
Egrep: regexpr -> NDFA -> DFA

- Example: \((a|aa|aaa)b\)

- NDFA:

- Convert to DFA by inventing states that represent *sets of states* of the NDFA:

- Recognition time is \(O(n)\)
- Construction time could be \(O(2^m)\)
  - because there are \(2^m\) subsets of the states
  - newer versions construct states as needed:
    - lazy evaluation

Important ideas from regexprs & grep

- **tools**: let the machine do the work
  - good packaging matters
- **notation**: makes it easy to say what to do
  - may organize or define implementation
- **hacking** can make a program faster, sometimes, usually at the price of more complexity
- a better algorithm can make a program go a lot faster
- don’t worry about performance if it doesn’t matter (and it often doesn’t)
- when it does,
  - use the right algorithm
  - use the compiler’s optimization
  - code tune, as a last resort