Data compression and statistical modeling

Török Edwin

18 ianuarie 2007

1 Algoritmi de compresie

1.1 Range encoding

Ideea de "range encoding" a apărut prima dată în lucrarea lui G.N.N Martin[1] .

Spunând că lăţimea unui mediu de stocare este *s*, sau *d* cifre din baza *b*, înţelegem că poate lua una din cele s valori, sau una din cele b^d valori distincte.

Dacă stocăm o literă, şi restrângem mediul de stocare la una din t valori distincte, atunci lăţimea codării caracterului este s/t, şi lăţimea rămasă este t, în care putem stoca un REST de lăţime *t* . Setul de t valori ce pot reprezenta litera, se numeşte DOMENIUL literei în lăţimea spaţiului de stocare.

De exemplu dacă domeniul unei litere într-un spaţiu de stocare cu lăţimea 256 este [240, 250), atunci lăţimea lilterei este 25.6, şi lăţimea rămasă este 10.

Dacă un domeniu are forma $[B, T)$, atunci îl putem combina cu un rest prin aritmetică simplă. Dacă dorim să stocăm $i \in [0, T - B)$, ca rest pentru $[B, T)$, atunci valoarea stocată este $B + i$; sau dacă $[i, j) \subseteq [0, T - B)$ trebuie stocat ca rest parţial pentru $[B, T)$, atunci valoarea stocată este constrâns la $[B + i, B + j)$.

Fie $f(a)$ probabilitatea ca litera 'a' să apară în orice context dat. Presupunem că alfabetul este ordonat, şi definim $F(a)$ ca fiind probabilitatea unei literi precedente lui 'a' să apară în acelaşi context, adică:

$$F(a) = \sum_{x<a} f(x)$$

În continuare voi nota $f(a)$ cu fa, $F(a)$ cu Fa, $s.fa$ cu sfa.

Shannon a arătat, că pentru minimizarea cifrelor necesare pentru reprezentarea mesajului într-o bază *b*, ar trebui să codăm fiecare literă *'a'*, a.î. lăţimea să fie $-\log_b(fa)$ cifre, adică $1/fa$ în lăţime absolută.

Nu putem realiza acest lucru exact, dar dacă codăm *'a'* într-un spaţiu de stocare cu lăţimea *s*, ca şi $[\lfloor s.Fa \rfloor , \lfloor s(fa + F$ atunci lăţimea literei se aproprie de $1/fa$ pentru $s.fa \gg 1$. Dacă $s.fa \geq 1$, atunci fiecare literă se poate coda, şi decoda fără echivoc.

1.1.1 Decodificare

O literă *'a'*, împreună cu restul său este codificat (într-un spaţiu de stocare de lăţime *s*) ca $i \subseteq [\lfloor sFa \rfloor , \lfloor s(Fa + fa) \rfloor]$. Fie $L(j)$ ultima literă *e* din alfabet pentru care $Fe < j$. Putem folosi *L* pentru a deduce *'a'*, ştiind *i*:

$$\lfloor sFa \rfloor \;\leq\; i < \lfloor s(Fa + fa) \rfloor \quad \Rightarrow sFa < i + 1 \leq s(Fa + fa) \quad \Rightarrow Fa < \frac{i+1}{s} \leq Fa + fa$$

$\Rightarrow a = L\left(\frac{i+1}{s}\right)$

Trebuie ţinut cont şi de erorile de rotunjire la calcularea lui $\frac{i+1}{s}$. Putem verifica dacă litera este corectă prin confirmarea relaţiei $\lfloor sFa \rfloor \leq i < \lfloor s(Fa + fa) \rfloor$.

După ce am dedus *'a'*, restul este $i - \lfloor sFa \rfloor$, şi a fost codat cu o lăţime de $\lfloor s(Fa + fa) \rfloor - \lfloor sFa \rfloor$.

1.1.2 Algoritmul de codificare/decodificare

Dacă o literă *'a'* se codifică ca $[B, T)$, lăţimea rămasă este $T - B$. Dacă acesta e prea mic, îl putem extinde prin adăugarea unei cifre (în baza *b*), domeniul devenind: $[Bb, Tb)$, şi lăţimea rămasă devine $(T-B)b$. La decodificare ignorăm cifra în plus, pentru că codificarea lui *'a'* în lăţimea sb nu este neapărat $[Bb, Tb)$.

Fie $s = b^w$, unde w este numărul (întreg) maxim de cifre în baza *b* pe care îl putem utiliza în mod convenabil.

Codificăm prima literă a mesajului în lăţimea *s*, şi adăugăm atâtea cifre în coadă, cât putem fără să cauzăm ca restul să depăşească lăţimea *s*.

Fie lăţimea spaţiului de stocare după codarea a celei de a *i*-a literă: S_i, de valoare $[B_i, T_i)$; atunci putem coda următoarea literă $A(i + 1)$, in spaţiul de stocare de lăţime $R(i + 1)$, unde:

$$\begin{aligned} R_{i+1} &= (T_i - B_i)b^{k(i+1)} \\ k_{i+1} &= w - \lceil \log_b(T_i - B_i) \rceil \end{aligned}$$

Pentru $i > 0$:

$$\begin{aligned}
[B_i, T_i) &= \left[B_{i-1}b^{k_i} + \lfloor R_i FA_i \rfloor\, , B_{i-1}b^{k_i} + \lfloor R_i(FA_i + fa_i)\rfloor\right) \\
S_i &= \sum_{j=1}^{i} k_j \\
[B_0, T_0) &= [0, 1)
\end{aligned}$$

1.1.3 Exemplu de codificare

Codificarea mesajului: *"NMLNNNKKNML"*

Lăţime rămasă (ajustat)	litera următoare	domeniul literei următoare	Mesaj curent codificat	Domeniul curent mesajului	Lăţime rămasă
1000	N	$[580, 1000)$	N	$[580, 1000)$	420
420	M	$[130, 243)$	NM	$[710, 823)$	113
113	L	$[011, 035)$	NML	$[721, 745)$	24
240	N	$[139, 240)$	NMLN	$[7349, \ldots 450)$	101
101	N	$[058, 101)$	NMLNN	$[7407, \ldots 450)$	43
430	N	$[249, 430)$	NMLNNN	$[74319, \ldots 500)$	181
181	K	$[000, 018)$	NMLNNNK	$[74319, \ldots 337)$	18
180	K	$[000, 018)$	NMLNNNKK	$[743190, \ldots 208)$	18
180	N	$[104, 180)$	NMLNNNKKN	$[7432004, \ldots 080)$	76
760	M	$[235, 440)$	NMLNNNKKNM	$[73420275, \ldots 480)$	205
205	L	$[020, 063)$	NMLNNNKKNML	$[73420295, \ldots 338)$	43

Codul complet trebuie ales cu 7 cifre semnificative (din: $[73420295, 73420338)$), de ex: 7432031.

1.1.4 Implementare algoritm

Se observă că în cazul unui domeniu există 3 zone distincte:

$$\begin{array}{l} [\ \underbrace{13}\ \underbrace{19}\ \underbrace{314}\ \ , \\ \ \ \underbrace{13}_{z1}\ \underbrace{20}_{z2}\ \underbrace{105}_{z3}\ \ ] \end{array}$$

Zona z1 constă din cifre comune tuturor numerelor din domeniu, deci nu vor fi afectate de alegerea restului. Aceste cifre pot fi scrise la ieşire.

Zona_z2 constă din *n* cifre formând un număr db^{n-1}, sau $db^{n-1} - 1$, unde *d* este o singură cifră, şi *b* este baza codificării. În aceste exemplu $n = 2$, şi $d = 2$. Cifrele din această zonă pot fi afectate de alegerea restului, dar care nu sunt necesare pentru a distinge 2 numere din domeniu. Acestea le numim cifre AMÂNATE, şi (d, n) identifică posibilele valori ale cifrelor. Prin convenţie, dacă $n = 0 \quad \Rightarrow d = 0$.

Zona z3 constă din *w* cifre, şi sunt suficiente pentru a distinge între 2 numere din domeniu.

Considerăm domeniul $[B', T']$, cu cifrele transmise: *c*, şi cifrele amânate reprezentate prin (d, n). Fie *x* cifrele transmise după rezolvarea amânării superior:

$$x = cb^n + db^{n-1}$$

atunci putem exprima $[B', T']$, ca: $c, (d, n), [B, T]$, unde $B = B' - xs$, şi $T = T' - xs$. De exemplu $[1319314, 1320105)$ devine $13, (2, 2), [-686, 105)$.

Dacă lăţimea rămasă este $T - B$, şi dacă combinăm $c, (d, n), [B, T]$ cu restul parţial $[i, j) \subseteq [0, T - B]$, atunci creăm domeniul $c, (d, n), [B + i, B + j]$.

Dacă $B+j \leq 0$ atunci putem rezolva cifra amânată inferior, iar dacă $B+i \geq 0$atunci îl putem rezolva superior.

Acest algoritm se poate implementa simplu, fiindcă, dacă domeniul este $c, (d, n), [B, T]$, atunci: $-s < B < T \leq +s$, unde:

d este o singură cifră

n este un întreg mic

c nu trebuie reţinut în codificator/decodificator

Pentru a limita numărul de cifre amânate, putem impune o limită superioară. Putem forţa rezolvarea amânării prin modificarea capetelor domeniului.

Ex:

$$13, (2, 3), [-660, 140] \Rightarrow 13, (2, 3), [-660, 000] \Rightarrow 13199, (0, 0), [340, 1000)$$

$$13, (2, 3), [-140, 660) \Rightarrow 13, (2, 3), [000, 660) \Rightarrow 13200, (0, 0), [000, 660)$$

Prin acesta risipim cel mult 1 bit.

1.2 Modelare statistică

1.2.1 Modelare statistică statică

Una din metodele de determinare a probabilităţii de apariţie a unui simbol este modelarea contextuală finită [6]. Acesta se bazează pe ideea că se calculează probabilităţile pentr un simbol pe baza contextului în care apare. *Contextul* reprezintă simbolii deja întâlniţi. *Ordinul* modelului se referă la numărul de simboluri precedente care alcătuiesc contextul.

Cel mai simplu model cu context finit este un model de ordinul 0. În acest caz probabilităţile unui simbol sunt independente. Pentru implementare este necesar doar un tabel cu frecvenţa de apariţie a simbolurilor.

Pentru un model de ordinul 1 avem nevoie de 256 asemenea tabele, pentru că trebuie să avem contori separaţi pentru fiecare context posibil. Pentru un model de ordinul 2 avem nevoie de 65536 tabele, ş.a.m.d.

O metodă de implementare este de a face 2 parcurgeri asupra datelor: una pentru a determina frecvenţa de apariţie, şi încă una pentru codificarea simbolurilor (folosind un codificator aritmetic, sau un range-encoder).

1.2.2 Modelare statistică adaptivă

Pentru modele de ordinul > 1, spaţiul ocupat de modelul statistic devine foarte mare, în comparaţie cu datele de intrare (şi în multe situaţii le depăşeşte).

Pentru a înlătura acest dezavantaj se evită stocarea modelului. Dar şi decodificatorul trebuie să cunoască modelul, şi acesta nefiind stocat împreună cu datele comprimate, înseamnă că decodificatorul trebuie să-l construiască pas-cu-pas. Acesta se numeşte modelare adaptivă.

În acest caz algoritmii de compresie şi decompresie pornesc cu acelaşi model, codifică simbolul cu modelul curent, şi după aceea reîmprospătează modelul cu noul simbol. Astfel modelul curent se bazează pe caracterele întâlnite deja, cunoscute atât de către programul de compresie, cât şi de către cel de decompresie.

1.3 Compresie folosind coduri distanţă-lungime

În specificaţia formatului DEFLATE [5] (format folosit de Zip) se utilizează coduri distanţă-lungime. O pereche <lungime, distanţă> are următoarea semnificaţie: se copiază începând de la <*poziţia curentă*> - <*distanţă*> la ieşire <*lungime*> octeţi.

De observat, că <*lungime*> poate fi mai mare decât <*distanţă*>. Exemplu: *ABCDX<8,1>*, înseamnă: *ABCDXXXXXXXXX*.

Pentru stocarea acestor coduri, în cazul formatului *DEFLATE* se extinde alfabetul de 256 coduri, cu încă 30 coduri. Codul 256 marchează sfârşitul unui bloc.

Lungimile se reprezintă cu codurile 257-285, împreună cu eventualele biţi extra, vezi tabela 1.

Cod	Biţi	Lungime	Cod	Biţi	Lungime	Cod	Biţi	Lungime
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Tabela 1: coduri pentru lungime

•

Aceste coduri se generează de obicei prin aplicarea algoritmului LZ77 (Lempel-Ziv 1977[9]).

2 Structuri de date folosite

2.1 Arbori digitali “expanse-based”: arbori Judy

Arborii Judy au fost inventaţi de către Doug Baskins[3], şi implementaţi împreună cu Alan Silverstein[2] la Hewlett Packard. Ulterior algoritmul, şi programul au fost făcute publice.

Un arbore Judy este mai rapid, şi utilizează mai puţină memorie decât alte forme de arbori, cum ar fi: arbori binari, AVL, arbori B, skip-list. Când este folosit ca şi înlocuitor pentru algoritmi de dispersie, este în general mai rapid pentru toate populaţiile.

Judy fiind proiectat ca şi un vector nelimitat, dimensiunea unui vector Judy nu este prealocat, ci creşte, şi descreşte dinamic cu populaţia vectorului.

Judy combină scalabilitatea cu uşurinţa în utilizare. API-ul Judy este accesat prin operaţii simple de inserare, regăsire, şi ştergere. Configurare, şi tuning nu sunt necesare, şi de fapt nici posibile pentru Judy. În plus sortarea, căutarea, numărarea, şi accesul secvenţial sunt incluse în Judy.

Judy poate fi folosit când este nevoie de vectori de mărime dinamică, vectori asociativi. De asemenea Judy poate înlocui multe structuri de date comune, cum ar fi: vectori, vectori sparse, tabele de dispersie, arbori B, arbori binari, liste liniare, skiplists, algoritmit de căutare şi sortare, funcţii de numărare.

O umplere a liniei cache (CPU) înseamnă timp adiţional pentru citire din RAM, când un cuvânt nu este găsit în cache. În calculatoarele actuale acest timp este în zona 50..2000 instrucţiuni. Deci o umplere a liniei cache trebuie evitată când 50 sau mai puţine instrucţiuni pot face acelaşi lucru.

Câteva motive pentru care Judy este mai bun decât arborii binary, arbori B, şi skiplists:

- Judy nu face compromisuri între simplitate şi performanţă/spaţiu (doar API-ul se păstrează simplu)
- Criteriul principal este: Judy e proiectat ca să evite umplerile de linii cache, când e posibil

- Un arbore B necesită o căutare a fiecărui nod, rezultând în mai multe umpleri de linii cache
- Un arbore binar are mai multe nivele (~8x), rezultând în mai multe umpleri de linii cache
- Un skip-list este aproximativ echivalent cu un arbore de grad 4, rezultând în mai multe umpleri de linii cache
- un arbore digital pe bază de întindere (a cărei variantă este Judy) nu necesită niciodată reechilibrări la creşterea arborelui
- o porţiune a cheii este utilizat pentru subdivizarea unei întinderi în sub-arbori. Doar restul cheii este trebuie să existe în sub-arbori, rezultând în compresia cheilor.

2.1.1 Arbori Judy de bază: JudyL, Judy1

JudyL reprezintă un arbore Judy care mapează întregi (pe 32/64 biţi) la întregi/pointeri (tot pe 32/64 biţi). Poate fi privit şi ca un vector "rar" (sparse).

Judy1 poate fi privit ca şi un vectori de biţi. Vectorul poate fi rar. Dacă un index este prezent înseamnă că bitul este setat, iar unul absent reprezintă un bit nesetat.

2.1.2 JudySL

JudySL este un vector asociativ, implementat folosing JudyL, în felul următor: se împarte un şir (terminat prin null), într-o secvenţă de cuvinte de 32/64 biţi lungime, şi se construieşte un arbore de vectori Judy, cu acele cuvinte ca şi indexi, reprezentând un prefix unic pentru fiecare şir. Fiecare nod terminal este un pointer la sufixul unic al şirului (un şir se poate termina şi fără nod terminal).

2.1.3 Diferenţiere JudyL, şi Judy1

Dacă într-o aplicaţie lucrăm atât cu pointeri către JudyL, cât şi către Judy1, şi nu vrem să irosim un flag care să facă distincţia, avem la dispoziţie o soluţie, care se bazează pe următoarea observaţie:

Un pointer (alocat cu *malloc(3), calloc(3), realloc(3)*) este aliniat la cel puţin 4 octeţi, acesta înseamnă că biţii cei mai puţin semnificativi sunt 00.

Astfel convenim ca un pointer care are biţii cei mai puţin semnificativi 00, este un pointer JudyL (lucru uşor de verificat cu o mască).

În cazul pointerilor Judy1 modificăm pointerul, setând biţii cei mai puţin semnificativi la stocare, respectivi îi resetăm ca să obţinem pointerul original.

Această abordare este folosită la implementarea JudySL, şi avem chiar şi un flag (JLAP_INVALID) pe care să-l putem folosi în mod portabil (curent JLAP_INVALID=0x1).

3 Contribuţii proprii la implementare de algoritmi de compresie

3.1 Coduri lungime-distanţă

În specificaţia *DEFLATE* se folosesc 28 de coduri pentru coduri de lungime, dar acesta se poate extinde în caz de nevoie.

Se observă următoarele la tabelul (1):

- primii 8 coduri au 0 extra biţi
- pentru următoarele 4 coduri avem *1* bit extra
- pentru următoarele 4 coduri avem *2* biţi extra
- ...
- pentru următoarele 4 coduri avem *i* biţi extra

Acesta conduce la următorul algoritm de generare a tabelului:

- primele 8 coduri, au 0 biţi extra
- $extra\_bits_i = \lfloor (i-8)/4 \rfloor$
- lungimile corespunzătoare unui cod $[start_i, end_i]$:
  - $start_i = end_{i-1} + 1$
  - $end_i = start + 2^{extra\_bits_i} - 1$

Acest algoritm l-am implementat în mk_codes.c ((A.1)).

3.2 Implementarea algoritmului LZ77 folosind arbori Judy

Pentru comprimarea datelor folosind coduri distanţă-lungime, elementul central este găsirea unei potriviri, de preferat a unei potriviri cât mai lungi.

Structura de date numită JudySL (secţiunea (2.1.1)) este foarte asemănătoare cu ce avem nevoie: asociază unui şir o valoare.

În cazul nostru valoarea este poziţia/distanţa, iar şirul sunt chiar datele din "sliding windowul" algoritmului LZ77. Singura problemă este că putem avea şi caracterul nul (0) în date, deci nu putem folosi JudySL ca atare.

Pe baza descrierii algoritmului JudySL din secţiunea (2.1.1) am realizat o structură de date, implementând următoarii operatori:

- judy_insert_bytearray(judyarray, data, length, position)
- judy_remove_bytearray(judyarray, data, length, position)
- lzbuff_search_longestmatch(judyarray, data, length, &distance, &position)

Implementarea se află în lz_coder.c(A.2).

Această implementare (realizată de mine), este unica care foloseşte arbori Judy pentru LZ77.

3.2.1 Structura de date creată de judy_insert_bytearray

Scopul este de a asocia unei secvenţe de caractere un întreg, reprezentând poziţia la care acesta apare în buffer-ul de intrare.

Datele de intrare se află într-un buffer circular. Dimensiunea bufferului s-a ales a fi putere a lui 2, pentru că în acest caz se poate determina simplu (şi eficient) offset-ul din buffer corespunzător unei anumite poziţii din fişierul original, şi anume: $offset \& len\_mask$, unde $len\_mask = 2^n - 1$, lungimea bufferului fiind 2^n.

Fiindcă JudyL lucrează cu întregi, procesăm din secvenţa de intrare câte 4 octeţi deodată. Ca să funcţioneze corect căutarea, trebuie să păstrăm ordinea octeţilor, deci avem nevoie de o reprezentare big-endian. Cum pe arhitectura intel, şi amd64 întregii se reprezintă folosind convenţia little-endian, trebuie să convertesc din big-endian în little-endian. Am folosit funcţia standard POSIX.1 *ntohl(3)* . Folosind compilatorul *gcc* (GNU C Compiler), sau *icc* (Intel C/C++ compiler), nici nu se mai face apelul de funcţie, ci se substituie cu instrucţiunea *bswap.*

Dar în acest fel putem lucra numai cu offseturi multiplii de 4. Pentru a evita acest neajuns, copiem primii 4 octeţi la sfârşitul bufferului, astfel suntem siguri că accesând orice offset din buffer, nu depăşim limitele lui.

În continuare se caută în arborele JudyL curent acest întreg determinat mai sus. Dacă nu se găseşte se inserează, împreună cu toţi subarborii necesari.

Dacă se găseşte, se citeşte valoarea asociată, acesta reprezentând un pointer către următorul arbore JudyL/Judy1.

Diferenţierea dintre JudyL/Judy1 se face ca şi la secţiunea **??**.

După procesarea unei anumite porţiuni din datele de intrare (o limită fixă), se stochează într-un arbore Judy1 poziţiile coresp.

De exemplu stocăm *"01234567ABCDEFGH"* cu limita de *2x4* octeţi, şi poziţia *0x200*; şi *"01234567ZBC-DEFH"* cu poziţia *0x300*:

"0123" => 0x30313233

"4567" => 0x34353637

judyarray1[0x30313233] => judyarray2 (inserat); judyarray2[0x34353637] => judy1array1 (inserat); judy1array1[0x
=> setat
judyarray1[0x30313233] => judyarray2 (găsit); judyarray2[0x34353637] => judy1array1 (găsit); judy1array[0x300]
=> setat

3.2.2 Căutarea folosing lzbuff_search_longestmatch

Se foloseşte procedeul de căutare de la secţiunea **??**. Diferenţa este la găsirea unei potriviri complete, respectivi a unei nepotriviri.

Dacă nepotrivirea este la poziţia *i* , ştim că *i* octeţi s-au potrivit, însă nu ştim încă care este situaţia celor 4 octeţi tocmai comparaţi. Ştim că nu sunt egali toţi, dar totuşi ar putea fi o potrivire de 1,2,3 octeţi. Pentru acesta am implementat *compare_bytes(a, b)*, care procedează în felul următor:

- se efectuează *xor* între *a* şi *b*
- acum acei octeţi care sunt 0, semnifică că există o potrivire între octeţii respectivi între *a* şi *b*
- în continuare se verifică octet-cu-octet până unde există potrivire, această verificare se face cu măşti
- această metodă necesită 4 instrucţii de salt, dar compilatorul *gcc* a optimizat ultima instrucţiune de salt, şi l-a înlocuit cu următoare secvenţă:
  - *cmp al,1*
  - *sbb edx,edx*
  - *not edx*
  - *add edx,4*
  - *mov eax,edx*
- codul în C din care a fost generat codul de mai sus este:
  - *if(c&0x000000ff) return 3;*
  - *return 4*
- Funcţionarea acestei secvenţe de cod se explică astfel:
  - *cmp al,1* : se setează CF, dacă al < 1, adică dacă al == 0
  - *sbb edx,edx* : care înseamnă edx = (edx + CF)-edx, sau edx = CF, sau al==0 => edx = 1, altfel 0
  - *not edx* : dacă edx == 1 => edx = 0xffffffff (-1); adică dacă al==0 => edx = 0xffffffff, altfel edx = 0
  - *add edx,4* : al==0 => edx = 4-1 =3; altfel edx=4
  - *mov eax,edx* : se returnează ca şi rezultat edx calculat anterior

Mai există şi posibilitatea ca indexul următor sau anterior să aibă mai multe potriviri, aşa că se compară şi acestea. În final se caută primul arbore Judy1 din ierarhia de arbori JudyL. Din aceste Judy1 se alege poziţia care este cea mai apropriată de cea curentă.

Se ştie lungimea potrivirii, iar distanţa se calculează cu formula: $distance = (offset_{curent} - offset_{gasit}) \& len\_mask$

3.3 Programul singularity-compress

Folosind implementarea LZ77 descris în secţiunea **??**, şi o variantă modificată a rangecoderului lui Michael Schindler[**?**] am realizat un program numit singularity-compress.

Modificările faţă de rangecoderul lui Michael Schindler sunt:

- utilizare model adaptiv de ordinul 0, în loc de model static
- extindere alfabet pentru LZ77

Alte caracteristici:

- Program de self-extract de 4608 octeţi (faţă de 16k-100k la celelalte programe testate)
- posibilitate de îmbunătăţire a ratei de compresie, prin înlocuirea modelului, cu unul mai eficient

3.3.1 Rezultate

Compresia unui fişier text de 17962 octeţi => 11423 octeţi în 0.6 secunde. Programul *zip* reuşeste o compresie mult mai bună: 5221 octeţi.

Concluzia este că deşi implementarea LZ77 este eficientă, un model adaptiv de ordinul 0 este mai ineficient decât codificarea Huffman.

Acest lucru nefiind adevărat pentru modele adaptive de ordin superior, cum dovedeşte secţiunea următoare.

3.4 Optimizări PAQ8F

Unul din cele mai puternice programe de compresie este PAQ8F creat de Matt Mahoney [8]. Acesta se bazează pe un codificator aritmetic (similar cu rangecoderul descris la secţiunea **??**, şi foloseşte mai multe modele pentru predicţia probabilităţii unui simbol. Aceste modele sunt combinate cu ajutorul unei reţele neuronale. Cu opţiunile pentru compresie maximă programul poate necesita câţiva GB de memorie, atât la compresie, cât şi la decompresie. De asemenea timpul de decompresie este aproape identic cu cel de compresie (fiecare trebuie să recalculeze modelul, iar acesta e cel mai costisitor).

Am considerat că pentru a putea fi folosit practic programul, are nevoie de o variantă mai simplă, care:

- foloseşte mai puţină memorie
- este mai rapid
- are un dezarhivator mai mic < 32kb

Am modificat paq8f, şi l-am denumit paq8f-mini. Acesta are următoarele caracteristici:

- dezarhivator de 23Kb
- arhivator de 52Kb

4 Comparaţie între programe de arhivare

4.1 Date de test

Am folosit datele de test de la *maximumcompression* [**?**]. Fiindcă programele de arhivare se pot comporta mai bine/mai rău în funcţie de tipul datelor de compresat, pentru o evaluare "cinstită" trebuie folosite mai multe tipuri de fişiere.

În pachetul de fişiere oferit de *maximumcompression*, există următoarele tipuri de fişiere:

- text (în engleză) - 1995 CIA World Fact book (2.9Mb)
- fişier log - loguri de trafic de la fighter-planes.com (20Mb)

- listă sortată de cuvinte - listă alfabetică de cuvinte engleze (4Mb)
- executabil (EXE) - Acrobat Reader 5.0 (3.7Mb)
- executabil (DLL) - Microsoft Office 97 Dynamic Link Library (3.6Mb)
- imagine BMP - imagine 1356x1020 / 16.7 mil culori (4Mb)
- imagine JPEG - imagine 1152x864 / 16.7 mil culori (823Kb)
- fişier HLP - Delphi First Impression OCX Help (4Mb)
- fişiere DOC - Occupational Health and Safety; MS Word (4Mb)
- fişier PDF - Macromedia Flash MX Manual; Adobe Acrobat (4.4Mb)

4.2 Programele de compresie

program	versiune	autor	algoritm
paq-mini	paq8f-mini	Török Edwin (Matt Mahoney)	CM+ari
paq	paq8f	Matt Mahoney	f+CM+ari
zip	2.32	Info-Zip	lz77+huff
rar	3.60	Alexander Roshal	f+LZ77+PPMII+huff
7zip	4.43	Igor Pavlovq	f+LZMA+PPMII+lz77+BWT
gzip	1.3.9	Jean-Loup Gally, FSF	LZ77
bzip2	1.0.3	Julian Seward	BWT+huff

algoritm	detalii	observaţii
CM	Context Modeling	
ari	codificare aritmetică	
f	filtre (aplicate înainte de compresie)	
LZ77	Lempel-Ziv 77	vezi secţiunea **??**
PPMII	Prediction By Partial Match	modelare statistică
huff	Huffman Coding	
LZMA	Lempel-Ziv-Markov chain Algorithm	specific 7zip
BWT	Burrow-Wheeler Transform	

4.2.1 Observaţii

- Programele paq8f-mini, şi paq8f au fost compilate astfel: *g++ -O3 -mfpmath=sse -msse -msse2 -msse3 -mmmx -m3dnow -funsafe-math-optimizations -funit-at-a-time -fomit-frame-pointer paq8f-mini.cpp*
- Programele zip, 7zip, gzip, bzip2 sunt cele din pachetele oficiale Debian pentru arhitectura amd64
- Programul rar, este cel de pe rarlabs.com, care este pe 32-biţi
- Pentru rar am folosit setările de compresie recomandate pe *maximumcompression*, ca să obţin rata de compresie maximă
- Pentru celelalte programe am folosit rata de compresie maximă
- Memoria folosită de paq8f a fost de 1.8Gb
- Celelalte programe au folosit mult mai puţină memorie, (16-64Mb)
- Programul *singularity-compress* descris în secţiunea **??** nu a fost inclus în teste, acesta nefiind eficient pe fişiere de dimensiuni mari

4.3 Efectuare teste

Testele le-am efectuat în următoarele condiţii:

- Sistem test: Athlon64 3200+, 2Gb RAM DDR400
- Sistem de operare: Debian sid (amd64), kernel: Linux 2.6.18-3-amd64
- compilator: gcc (GCC) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
- măsurare timp: time (GNU time 1.7)

Compilatorul l-am folosit doar la paq8f-mini, şi paq8f ; pentru celelalte programe am folosit pachete binare gata compilate.

Paq8f-mini este varianta mea de paq8f descrisă în secţiunea **??**, iar paq8f este varianta originală.

4.3.1 Alegere parametrii de măsurat

- dimensiunea comprimată, fiindcă scopul principal al programelor de compresie este reducerea acestuia
- timp de compresie: acesta este mai puţin important, dar totuşi trebuie să fie o valoare rezonabilă
  - de exemplu: se foloseşte compresie pentru trimiterea datelor prin reţea. În acest caz scopul este de a transfera fişierul cât mai repede, acesta însemnând:$timp = timp_{compresie} + dimensiune_{comprimat}/viteza_{trans}$
  - dacă timpul de compresie este foarte mare, poate depăşi chiar şi timpul necesar transmiterii datelor originale, adică pentru a fi util trebuie îndeplinită condiţia:
  - $timp_{compresie} + \frac{dimensiune_{comprimata}}{viteza_{transfer}} < \frac{dimensiune_{originala}}{viteza_{transfer}}$

4.3.2 Parametrii derivaţi

- rată de compresie = $\frac{(dimensiune_{originala} - dimensiune_{comprimat})}{dimensiune_{originala}}$, interpretare: mai mare => mai bun
- viteză = $\frac{dimensiune_{originala}[kB]}{timp_{compresie}}[kB/s]$, interpretare: mai mare => mai bun

4.4 Rezultate compresie

program	timp arhivare	timp dezarhivare	dim comprimată	rată	viteză(kB/s)
paq8f-mini	1362	1364	10709453	79.84	38
paq8f	21000	-	9079591	82.1	2.4
zip	17.8	1.8	39743794	25.2	2915
rar	30	-	11486327	78.3	1687
7zip	40	2.45	11688445	78	1297
gzip	8.8	1.2	14025629	73.6	5891
bzip2	138	138	13379118	74.8	376

4.5 Concluzii

- Cea mai bună rată de compresie o are *paq8f.* Preţul este viteza foarte scăzută, şi memoria necesară (1.8Gb).
- Următorul este *paq8f-mini* are o rată de compresie apropriată, o viteză mult mai bună, şi memorie folosită de 20Mb
- Foarte apropiaţi ca şi rată de compresie sunt *rar* şi *7zip*
- În schimb cel mai rapid este *gzip.* Rata de compresie fiind mai scăzută, dar totuşi acceptabilă.
- Următoarele programe cu viteză destul de bună, şi rată de compresie rezonabilă sunt *rar* şi *7zip*

Părerea mea este că atingerea unei rate de compresie cât mai mari este importantă, dar trebuie avut în vedere şi timpul de compresie, şi necesarul de memorie. În acest sens merită studiat cum se poate îmbunătăţii în continuare paq8f-mini.

De asemenea pentru comprimarea de fişiere mici este importantă dimensiunea programului de dezarhivare. Pentru reducerea acestuia pot fi folosite şi următoarele metode:

- scoaterea mesajelor de eroare/progres afişate de dezarhivator
- comprimarea dezarhivatorului cu UPX
- separarea codului dezarhivatorului de cel al arhivatorului
- folosirea optimizării pentru dimensiune a compilatorului
- ştergere simboluri de debug din executabil
- ştergere comentarii adăugata de compilator (secţiunea .comment în cazul gcc)

Toate acestea le-am aplicat la singularity-compress, obţinând un executabil de 3585 octeţi.

De asemenea merită studiat cu ce algoritm de codificare se poate obţine o viteză, şi rată de compresie rezonabilă, folosind algoritmul LZ77 cu arbori Judy **??**. Este clar că alegerea unui codificator aritmetic de ordinul 0 nu este bună.

Bibliografie

[1] G.N.N Martin - “Range encoding: an algorithm for removing redundancy from a digitised message.”, Video & Data Recording Conference, Southampton, 1979, http://www.compressconsult.com/rangecoder/rngcod.pdf.gz

[2] Alan Silverstein - “Judy IV Shop Manual”, Hewlett-Packard, August 2002, http://judy.sourceforge.net/doc/shop_interm.pdf

[3] Doug Baskins - “A 10-minute description of how Judy arrays work and why they are so fast”, http://judy.sourceforge.net/doc/10minutes.htm, July 2002

[4] * - http://judy.sourceforge.net/examples/judysl.pdf

[5] Peter Deutsch - “DEFLATE Compressed Data Format Specification version 1.3 (RFC 1951)”, May 1996 - ftp://ftp.nic.it/rfc/rfc1951.pdf

[6] Mark Nelson - “Arithmetic Coding + Statistical Modeling = Data Compression”, Dr. Dobb’s Journal, February 1991, http://www.dogma.net/markn/articles/arith/part2.htm

[7] Moffat, Neal, and Witten - "Arithmetic Coding Revisited", ACM Transactions on Information Systems, July 1998, 16(3):256-294

[8] Matthew V. Mahoney - “Adaptive Weighing of Context Models for Lossless Data Compression”, Florida Institute of Technology CS Dept, Technical Report CS-2005-16, https://www.cs.fit.edu/Projects/tech_reports/cs-2005-16.pdf

[9] Michael Schindler - http://www.compressconsult.com/rangecoder/

[10] Ziv J., Lempel A. - “A Universal Algorithm for Sequential Data Compression”, IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343. http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

[11] * - http://www.maximumcompression.com

A Programul singularity-compress

Se găseşte şi la http://code.google.com/p/singularity-compress/ , http://singularity-compress.googlecode.com/svn/trunk/

```
/*
 * Singularity-compress: LZ77 encoder: length code tables
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110-1301, USA.
 */
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

static void make_length_code_tables(uint8_t extra_symbols,const ssize_t start)
{
    ssize_t code;
    unsigned short extra_bits = 0;
    unsigned short extra_bits_count = 0;
    ssize_t      i;

    printf("#include \"stdlib.h\"\n\n");
    printf("static const struct {\n");
    printf("\tssize_t start;\n");
    printf("\tuint8_t  extra_bits;\n");
    printf("} code_to_length[] = {\n");

    printf("\t");
    for(code = start; code < start + 8 && code < start + extra_symbols; code++)
        printf("{%lu, 0}, ",code);

    extra_bits = 1;
    printf("\n\t");
    for(i = 8; i < extra_symbols; i++ ) {
        printf("{%lu, %u}, ", code, extra_bits);

        code += (1 << extra_bits);
        extra_bits_count++;

        if ( extra_bits_count == 4) {
            printf("\n\t");
            extra_bits++;
            extra_bits_count=0;
        }
```

```
    }
    printf("{%ld, 0xff}\n",code);

    printf("};\n");
}


int main(int argc,char* argv[])
{
    printf("#ifndef _CODES_TABLE_H\n");
    printf("#define _CODES_TABLE_H\n\n");
    make_length_code_tables(28,3);
    printf("#endif\n");
    return 0;
}
```

A.2 lz_coder.c

```
/*
 * Singularity-compress: LZ77 encoder
 * Copyright (C) 2006-2007     Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110-1301, USA.
 */
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef WIN32
#include <netinet/in.h>
#else

#define ntohl(x) bswap_32(x)
#endif

#include <stdarg.h>
#include "lz_coder.h"

/* ntohl uses bswap_32, as appropriate for current endianness,
 * also bswap_32 on libc/gcc uses asm instructions to swap bytes with 1 instruction */
#define CHAR4_TO_UINT32(data, i) ntohl((*(const uint32_t*)(&data[i])))
```

```
/* Use JLAP_INVALID to mark pointers to "not JudyL arrays", in this case pointers to Judy1 array
 * JLAP_INVALID is defined in Judy.h, and is currently 0x1. Since malloc returns a pointer at least
 * 4–byte
 * aligned, marking it with 0x1, allows us to differentiate Judy1, and JudyL pointers */

#define J1P_PUT( PJ1Array )    ( (Pvoid_t) ( (Word_t)(PJ1Array) | JLAP_INVALID) )
#define J1P_GET( PJ1Array )    ( (Pvoid_t) ( (Word_t)(PJ1Array) & ~JLAP_INVALID) )
#define IS_J1P(    PJ1Array )    ( (Word_t)(PJ1Array) & JLAP_INVALID )


/* offset –> WRAP(offset+buffer_len_half) : new buffer
 * WRAP(offset+buffer_len/2–1) –> offset–1: old buffer (search buffer)
 */

#define WRAP_BUFFER_INDEX(lz_buff, index) ((index) & (lz_buff–>buffer_len_mask))

/* error values */
#define EMEM –2



/* debug logging */
/*#define LOG_DEBUG 1*/


#if !defined(NDEBUG) && defined(LOG_DEBUG)
static void log_debug(const size_t line, const char* fmt,...)
{
    va_list ap;
    va_start(ap,fmt);
    fprintf(stderr,"DEBUG %s: %ld ",__FILE__,line);
    vfprintf(stderr,fmt,ap);
    va_end(ap);
}

#else
static inline void log_debug(const size_t line, const char* fmt,...) {}
#endif

/*  ********** */
int setup_lz_buffer(struct lz_buffer* lz_buffer,const size_t buffer_len_power)
{
    lz_buffer–>buffer_len_power = buffer_len_power;
    lz_buffer–>buffer_len       = 1 << buffer_len_power;
    lz_buffer–>buffer_len_mask  = lz_buffer–>buffer_len – 1;
    lz_buffer–>offset           = 0;
    lz_buffer–>jarray           = (Pvoid_t) NULL;
    lz_buffer–>buffer = calloc(lz_buffer–>buffer_len+4,1);
    if(!lz_buffer–>buffer) {
        log_debug(__LINE__,"Out of memory while trying to allocate %ld bytes",
          lz_buffer–>buffer_len);
        return EMEM;
    }
```

```
    return 0;
}


static void judy_free_tree(Pvoid_t jarray)
{
    Word_t Index = 0;

    if(!jarray)
        return;

    if(IS_J1P(jarray)) {
        int rc;
        Pvoid_t judy1_node = J1P_GET(jarray);

        log_debug(__LINE__,"Retrieved original Judy1 array pointer: %p from %p, \
freeing it.\n",judy1_node, jarray);

        J1FA(rc, judy1_node);
    }
    else {
        Word_t *PValue;

        log_debug(__LINE__,"Retrieving first entry in JudyL array: %p. ",jarray);

        JLF(PValue, jarray, Index);

        log_debug(__LINE__,"Retrieved first entry: index: %lx; value: %p -> %p\n\
", Index, PValue, *PValue);

        while(PValue != NULL) {
            log_debug(__LINE__,"At entry: index: %lx, value: %p -> %p\n", Index, PValue,
              *PValue);

            judy_free_tree((Pvoid_t)*PValue);
            JLN(PValue,jarray, Index);
        }
        {
            int rc;
            log_debug(__LINE__,"Freeing JudyL array: %p\n",jarray);
            JLFA(rc,jarray);
        }
    }
}

void cleanup_lz_buffer(struct lz_buffer* lz_buffer)
{
    if(lz_buffer->buffer) {
        free(lz_buffer->buffer);
        lz_buffer->buffer = NULL;
    }
    if(lz_buffer->jarray)
        judy_free_tree(lz_buffer->jarray);
}
```

```
static Pvoid_t create_judy_tree(const struct lz_buffer* lz_buff,const size_t offset, const ssize_t
 length,const size_t position)
{
    Pvoid_t    PJLArray = (Pvoid_t) NULL;
    Pvoid_t* const first_node = &PJLArray;
    Pvoid_t* node = first_node;
    ssize_t i;

    for(i=0; i < length-4 ; i += 4) {
        Pvoid_t* next_node;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset+i) );

        log_debug(__LINE__,"create_judy_tree: Inserting %x into JudyL array: %p \
-> %p\n",val, node, *node);

        JLI(next_node, *node, val );

        log_debug(__LINE__,"create_judy_tree: Inserted into %p->%p, value: %p \
-> %p\n", node, *node, next_node, *next_node);

        if (next_node == PJERR)
            return PJERR;
              node = (Pvoid_t*) next_node;
    }
    {
        int rc;

        log_debug(__LINE__,"create_judy_tree: Inserting into judy1 array \
%p->%p, value:%lx\n",node,*node,position);

        J1S(rc, *node, position );

        log_debug(__LINE__,"create_judy_tree: Inserted into %p->%p\n",node,*node);

        if(rc == JERR)
            return PJERR;

        *node = J1P_PUT(*node);
        if(i==0)
            return *node;
    }
    return *first_node;
}


static int judy_remove_bytearray(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 length,Pvoid_t* node,size_t position)
{

    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);
```

```
    if( length>0 && !IS_J1P(*node) ) {
        int rc;
        Pvoid_t* next;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset) );

        log_debug(__LINE__,"judy_remove_bytearray: Querying %x, in JudyL array \
%p->%p.  ",val,node,*node);

        JLG(next,*node, val);

        log_debug(__LINE__,"Query result: %p->%p\n", next, next==NULL ? NULL : *next);

        if(next == NULL) {
            log_debug(__LINE__,"judy_remove_bytearray: not found:%x!\n",val);
            return -1;
        }
        rc = judy_remove_bytearray(lz_buff, offset + 4, length-4, next, position);
        if(!*next) {
            JLD(rc, *node,val);
        }
        return rc;
    } else if( IS_J1P(*node)) {
        int rc;
        Pvoid_t judy1_node    = J1P_GET(*node);

        log_debug(__LINE__,"Retrieved Judy1 pointer %p from %p->%p.Removing \
value: %lx\n",judy1_node,node,*node, position);

        J1U(rc,judy1_node, position );
        if(!rc) {
            log_debug(__LINE__,"judy_remove_bytearay: not Found pos:%ld!\n",position);
        }

        log_debug(__LINE__,"Storing Judy1 array into %p->%p\n",node,*node);

        if(judy1_node) {
            *node = J1P_PUT(judy1_node);
            if(rc == JERR)
                return JERR;
        } else {
            *node = NULL;
        }
    }
    return 0;
}


int lz_remove(struct lz_buffer* lz_buff,const size_t offset)
{
    return judy_remove_bytearray(lz_buff,offset,lz_buff->buffer_len/2,&lz_buff->jarray,
     WRAP_BUFFER_INDEX(lz_buff, offset));
}
```

```
/* length must be multiple of 4 */
static int judy_insert_bytearray(const struct lz_buffer* lz_buff,const size_t offset, const size_t length,
 Pvoid_t* node,size_t position)
{
    size_t i;


    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);

    for(i=0;i < length && !IS_J1P(*node); i += 4) {
        Pvoid_t* next;
        const uint32_t val = CHAR4_TO_UINT32(lz_buff->buffer, WRAP_BUFFER_INDEX(lz_buff,
         offset + i) );

        log_debug(__LINE__,"judy_insert_bytearray: Querying %x, in JudyL array \
%p->%p.  ",val,node,*node);

        JLG(next,*node, val);

        log_debug(__LINE__,"Query result: %p->%p\n", next, next==NULL ? NULL : *next);

        if(next == NULL) {
            log_debug(__LINE__,"judy_insert_bytearray: Inserting %x, into \
%p->%p.\n",val,node,*node);

            JLI(next, *node,val);

            log_debug(__LINE__,"judy_insert_bytearray: Inserted into %p->%p, \
value: %p->%p\n",node,*node,next,*next);

            *next = create_judy_tree(lz_buff, offset+i+4, length-i-4, position);

            log_debug(__LINE__,"judy_insert_bytearray: Stored into \
:%p->%p;value: %p->%p\n",node,*node,next,*next);

            if(*next == PJERR)
                return JERR;
            else
                return 0;
        }
        node = next;
    }

    if(IS_J1P(*node)) {
        int rc;
        Pvoid_t judy1_node    = J1P_GET(*node);

        log_debug(__LINE__,"Retrieved Judy1 pointer %p from %p->%p. Setting \
value: %lx\n",judy1_node,node,*node, position);

        J1S(rc,judy1_node, position );

        log_debug(__LINE__,"Storing Judy1 array into %p->%p\n",node,*node);
```

```
        *node = J1P_PUT(judy1_node);
        if(rc == JERR)
            return JERR;
    }
    return 0;
}

int lzbuff_insert(struct lz_buffer* lz_buff, const char c)
{
    const int rc = judy_insert_bytearray(lz_buff, lz_buff->offset, lz_buff->buffer_len/2, &lz_buff
     ->jarray, lz_buff->offset);
    lz_buff->offset = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + 1 );
    /*lz_buff->buffer[ lz_buff->offset ] = c;*/
    return rc;
}

static int get_longest_match(const struct lz_buffer * lz_buff, const uint32_t prev,const uint32_t next,
 size_t pos,ssize_t* distance, ssize_t* length)
{
    size_t i;
    const size_t buffer_half_len = lz_buff->buffer_len/2;
    const ssize_t data_search_idx = lz_buff->offset - buffer_half_len/2 - 1;
    ssize_t data_buff_idx        = prev;

    size_t match_len = pos;

    for(i = pos; i < buffer_half_len; i++) {
        if(lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, data_search_idx + i)] != lz_buff
         ->buffer[WRAP_BUFFER_INDEX(lz_buff, data_buff_idx + i)]) {
            match_len = i - 1;
            break;
        }
    }

    if(i == buffer_half_len) {
        /* entire buffer matches */
        *length = buffer_half_len;
        *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
        return 0;
    }

    data_buff_idx = next;

    for(i = pos; i < buffer_half_len; i++) {
        if(lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, data_search_idx + i)] != lz_buff
         ->buffer[WRAP_BUFFER_INDEX(lz_buff, data_buff_idx + i)]) {
            if(i - 1 < match_len) {
                /* data_buff_idx = prev has a longer match */
                *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
            }
            else {
                match_len = i-1;
                *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - next);
```

```
            }
            *length = match_len;
            return 0;
        }
    }

    /* entire buffer matches */
    *length = buffer_half_len;
    *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - prev);
    return 0;
}


/* return number of bytes that match,
 * we always assume big endian, since
 * 0x30313233, comes from the string "0123" */
static inline uint8_t compare_bytes(const uint32_t a,const uint32_t b)
{
    const uint32_t c = a ^ b;/*will be 0 where there is a match*/
    if(c&0xff000000) /* if there are any 1's there, we got a mismatch in the very first byte */
        return 0;
    if(c&0x00ff0000)
        return 1;
    if(c&0x0000ff00)
        return 2;
    if(c&0x000000ff)
        return 3;
    return 4;
}


static inline const Pvoid_t* get_best_match(const uint32_t orig_val, const Word_t prev_val, const
 Word_t next_val, const Pvoid_t* prev, const Pvoid_t* next,uint8_t* match)
{
    /* determine how many bytes we have matched */
    const uint8_t match_next = compare_bytes(next_val, orig_val);
    const uint8_t match_prev = compare_bytes(prev_val, orig_val);
    if( match_prev < match_next) {
        *match = match_next;
        return next;
    }

    *match = match_prev;
    return prev;
}


/* gets index that is closest */

#define J1_CLOSEST(rc, J1Array, search_index,       prev_index, next_index) \
{\
    (next_index) = (search_index);\
    J1F((rc), (J1Array), (next_index));\
    if(!(rc)) {\
```

```
        (prev_index) = (search_index);\
        J1P((rc), (J1Array), (prev_index));\
        if((rc)) {\
            (next_index) = (prev_index);\
            J1N((rc), (J1Array), (next_index));\
        }\
    }\
    else {\
        if((next_index) != (search_index)) {\
            (prev_index) = (next_index);\
            J1P((rc), (J1Array), (prev_index));\
        }\
    }\
}

#define J1_NEIGHBOUR(rc, J1Array, search_index, neighbour_index) \
{\
    (neighbour_index) = (search_index);\
    J1F((rc), (J1Array), (neighbour_index));\
    if(!(rc)) {\
        (neighbour_index) = (search_index);\
        J1P((rc), (J1Array), (neighbour_index));\
    }\
}

static int get_closest(const struct lz_buffer* lz_buff,const Pvoid_t* node,ssize_t* distance)
{
    /* now walk the array of JudyL arrays, till we reach judy1 pointers, and then select the
     * position stored there that is closest to current offset*/
    while(node && !IS_J1P(*node)) {
        Pvoid_t *next;
        Word_t val = 0;
        JLF(next,*node,val);
        node = next;
    }

    if(node == NULL) {
        fprintf(stderr,"Warning: encountered unexpected empty JudyL array");
        *distance = 0;
        return -2;
    }
    else {
        int rc;
        Pvoid_t judy1_node = J1P_GET(*node);
        Word_t val ;

        J1_NEIGHBOUR(rc, judy1_node, lz_buff->offset, val);

        if(!rc) {
            fprintf(stderr,"Emtpy judy1 array?\n");
            *distance = 0;
            return -1;
        }
```

```
        *distance = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - val);
        return 0;
    }
}


int lzbuff_search_longest_match(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 data_len, ssize_t* distance, ssize_t* length)
{
    size_t i;
    /* start searching from root array */
    const Pvoid_t* node = &lz_buff->jarray;

    memcpy(&lz_buff->buffer[lz_buff->buffer_len],&lz_buff->buffer[0],4);
    /* when we reach end-of-buffer during a search, we would need to wrap the search,
     * but we can only wrap on multiples of 4, so to prevent accessing uninitialized memory,
     * we copy first 4 bytes, to end of buffer */

    for(i=0;i < data_len && !IS_J1P(*node); i+=4) {
        const Pvoid_t* next;
        const Pvoid_t* prev;

        const uint32_t orig_val = CHAR4_TO_UINT32(lz_buff->buffer,
         WRAP_BUFFER_INDEX(lz_buff, offset + i) );
        /* indexes in the array are groups of 4 characters converted into uint32 */
        Word_t prev_val;
        Word_t next_val = orig_val;

        /* search the first index >= than the value we search for,
         * or if it doesn't exist, then the index lower than it
         * next is the value contained in the JudyL array at index val
         * It is really a pointer to another JudyL array (if IS_J1P(*next) == 0) ;
         * Or it is a pointer to a Judy1 array if IS_J1P(*next) == 1
         */

        JLF( next, *node, next_val);
        if(!next) {
            prev_val = orig_val;
            JLP(prev, *node, prev_val);
            if(prev) {
                /* previous found, search for next neighbour now */
                next_val = prev_val;
                JLN(next, *node, next_val);
                if(!next) {
                    /* array has only 1 element */
                    next = prev;
                    next_val = prev_val;
                }
            }
            else {
                log_debug(__LINE__,"Empty JudyL array?\n");
                /* no value found in JudyL array, buffer is empty */
                *length = -1;
                *distance = 0;
```

```
            return −1;
        }
    }
    else {
        if(next_val != orig_val) {
            prev_val = next_val;
            JLP(prev, ∗node, prev_val);
            if(!prev) {
                /∗ array has only 1 element ∗/
                prev = next;
                prev_val = next_val;
            }
        }
    }

    if(next_val != orig_val) {
        uint8_t match;
        /∗ we encountered a mismatch ∗/
        node = get_best_match(orig_val, prev_val, next_val, prev, next ,&match);
        ∗length = match+i;
        return get_closest(lz_buff, node, distance);
    }

    node = next;
}

if(IS_J1P(∗node)) {
    int rc;
    Pvoid_t judy1_node = J1P_GET(∗node);
    const uint32_t orig_val = CHAR4_TO_UINT32(lz_buff−>buffer,
     WRAP_BUFFER_INDEX(lz_buff, offset + i) );
    Word_t val = orig_val;
    Word_t val_prev, val_next;

    log_debug(__LINE__,"Retrieved original Judy1 array pointer: %p from \
%p−>%p\n",judy1_node, node,∗node);

    J1_CLOSEST(rc, judy1_node, val, val_prev, val_next);

    if(val_next != orig_val) {
        return get_longest_match(lz_buff,val_prev, val_next,i, distance, length);
    }
    else {
        ∗distance = −data_len;
        ∗length   = data_len;
        return 0;
    }
}
else {
    return −1;
    /∗ assertion failure ∗/
}
}
```

```
static void judy_show_tree(Pvoid_t jarray,int level)
{
    char* spaces = malloc(level+1);
    Word_t Index = 0;

    memset(spaces,0x20,level);
    spaces[level] = 0;

    if(IS_J1P(jarray)) {
        int rc;
        Pvoid_t judy1_node = J1P_GET(jarray);
        J1F(rc,judy1_node,Index);
        while(rc) {
            fprintf(stderr,":%s%lx\n",spaces,Index);
            J1N(rc,judy1_node,Index);
        }
    }
    else {
        Word_t * PValue;
        log_debug(__LINE__,"querying first:%p\n",jarray);
        JLF(PValue, jarray, Index);
        while(PValue != NULL) {
            fprintf(stderr,"%s%lx\n",spaces,Index);
            judy_show_tree((Pvoid_t)*PValue, level+1);
            JLN(PValue, jarray, Index);
        }
    }
    free(spaces);
}

void show_lz_buff(const struct lz_buffer* lz_buff)
{
    size_t i;
    for(i=0;i < lz_buff->buffer_len;i++) {
        if(i == lz_buff->offset)
            fprintf(stderr,"|");
        fprintf(stderr,"%02x ",lz_buff->buffer[i]);
    }
    fprintf(stderr,"\n");
}

void show_match(const struct lz_buffer* lz_buff,ssize_t distance,ssize_t length)
{
    size_t start = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - distance);

    ssize_t match = -1;
    size_t i;

    for(i=0;i < lz_buff->buffer_len;i++) {
        if(i == start) {
            fprintf(stderr,"<");
            match = 0;
        }
```

```
        if(match >= 0)
            match++;
        fprintf(stderr,"%02x ",lz_buff->buffer[i]);
        if(match == length)
            printf(">");
    }
    fprintf(stderr,"\n");
}

/*
int main(int argc,char* argv[])
{
    const unsigned char test[] = "0123456789012345";
    const size_t len = sizeof(test)-1;
    size_t i;

    struct lz_buffer        lz_buff;
    FILE* out = fopen("/tmp/testout","w");

    setup_lz_buffer(&lz_buff,5);


     this will be replaced by a read()
    for(i=0;i < len;i++) {
        lz_buff.buffer[lz_buff.offset + i] = test[i];
    }


    for(i=0; i < len;) {
        ssize_t distance;
        ssize_t     length;

        show_lz_buff(&lz_buff);

        lzbuff_search_longest_match(&lz_buff, &lz_buff.buffer[lz_buff.offset], lz_buff.buffer_len/2,
        &distance, &length);

        show_match(&lz_buff,distance,length);

        lzbuff_insert(&lz_buff, lz_buff.buffer[lz_buff.offset + i]);

        show_lz_buff(&lz_buff);
        judy_show_tree(lz_buff.jarray,0);
        printf("match:%ld,%ld\n",distance,length);
        if(length < 3) {
            length = 1;
        }
        else {
        }
        i += length;
    }
    cleanup_lz_buffer(&lz_buff);

    return 0;
```

```
}
 */
/*
int main(int argc,char* argv[])
{
    Pvoid_t jarray = (Pvoid_t) NULL;
    const size_t len = 16*2;
    const size_t maxIdx = 8;
    unsigned char* test = malloc(len+1);

    const unsigned char test2[] = "0123456789012346";
    const size_t len2 = sizeof(test)-1;
    size_t i;
    ssize_t distance=0;
    size_t length=0;

    srand(2);
    for(i=0;i<len/2;i++) {
        test[i] = test2[i%len2];
    }
    test[len]=0;
    for(i=0;i<len/2;i++) {
        judy_insert_bytearray(test+i,maxIdx,&jarray,i);
    }
    judy_insert_bytearray(test,maxIdx,&jarray,0);

    judy_show_tree(jarray,0);
    struct lz_buffer buff;
    lzbuff_search_longest_match(&buff,test2+10,len2-10,&distance,&length);
    printf("%ld:%ld\n",distance,length);
    judy_free_tree(jarray);
    jarray = NULL;
    return 0;
}
 */
```

A.3 lz_coder.h

```
/*
 * Singularity-compress: LZ77 encoder
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
```

```
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */
#ifndef _LZ_CODER_H
#define _LZ_CODER_H

#include <Judy.h>

struct lz_buffer {
    unsigned char*      buffer;
    size_t      buffer_len;
    size_t      buffer_len_mask;
    size_t      buffer_len_power;/* buffer_len = 2^buffer_len_base2 */
    ssize_t     offset;
    Pvoid_t  jarray;
};
int setup_lz_buffer(struct lz_buffer* lz_buffer,const size_t buffer_len_power);
void cleanup_lz_buffer(struct lz_buffer* lz_buffer);

int lzbuff_insert(struct lz_buffer* lz_buff, const char c);
int lzbuff_search_longest_match(const struct lz_buffer* lz_buff,const size_t offset,const size_t
 data_len, ssize_t* distance, ssize_t* length);

int lz_remove(struct lz_buffer* lz_buff,const size_t offset);
/* offset -> WRAP(offset+buffer_len_half) : new buffer
 * WRAP(offset+buffer_len/2-1) -> offset-1: old buffer (search buffer)
 */
#define WRAP_BUFFER_INDEX(lz_buff, index) ((index) & (lz_buff->buffer_len_mask))

/* debugging functions - to be removed in a release */
void show_lz_buff(const struct lz_buffer* lz_buff);
void show_match(const struct lz_buffer* lz_buff,ssize_t distance,ssize_t length);
#endif
```

A.4 range_encoder.h

```
/*
 * Singularity-compress: rangecoder encoder
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1997, 1998, 1999, 2000
 * http://www.compressconsult.com/
 *
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
```

```
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */

#ifndef _RANGE_ENCODER_H
#define _RANGE_ENCODER_H
#include "../common/rangecod.h"

/* rc is the range coder to be used                                */
/* c is written as first byte in the datastream                    */
/* one could do without c, but then you have an additional if      */
/* per outputbyte.                                                 */
static void start_encoding( rangecoder *rc, char c, int initlength )
{
    rc->low = 0;                    /* Full code range */
    rc->range = Top_value;
    rc->buffer = c;
    rc->help = 0;                   /* No bytes to follow */
    rc->bytecount = initlength;
}

static inline void enc_normalize( rangecoder *rc )
{
    while(rc->range <= Bottom_value) {
        /* do we need renormalisation?      */
            if (rc->low < (code_value)0xff << SHIFT_BITS) {
            /* no carry possible --> output */
            outbyte(rc, rc->buffer);
            for(; rc->help; rc->help--)
                outbyte(rc, 0xff);
            rc->buffer = (unsigned char)(rc->low >> SHIFT_BITS);
        } else if (rc->low & Top_value) {
            /* carry now, no future carry */
            outbyte(rc,rc->buffer+1);
            for(; rc->help; rc->help--)
                outbyte(rc,0);
            rc->buffer = (unsigned char)(rc->low >> SHIFT_BITS);
        } else    {
            /* passes on a potential carry */
#ifndef DO_CHECKS
            rc->help++;
#else
            if (rc->bytestofollow++ == 0xffffffffL) {
                fprintf(stderr,"Too many bytes outstanding - File too large\n");
                exit(1);
            }
#endif
        }

        rc->range <<= 8;
        rc->low = (rc->low<<8) & (Top_value-1);
        rc->bytecount++;
    }
```

```
}


/* Encode a symbol using frequencies                                    */
/* rc is the range coder to be used                                     */
/* sy_f is the interval length (frequency of the symbol)                */
/* lt_f is the lower end (frequency sum of < symbols)                   */
/* tot_f is the total interval length (total frequency sum)             */
/* or (faster): tot_f = (code_value)1<<shift                                    */
static void encode_freq( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f )
{
    code_value r, tmp;
    enc_normalize( rc );
    r = rc->range / tot_f;
    tmp = r * lt_f;
    rc->low += tmp;
    if (lt_f+sy_f < tot_f)
        rc->range = r * sy_f;
    else
        rc->range -= tmp;
#ifdef DO_CHECKS
    if(!rc->range)
        fprintf(stderr,"ooops, zero range\n");
#endif
}

static void encode_shift( rangecoder *rc, freq sy_f, freq lt_f, freq shift )
{
    code_value r, tmp;
    enc_normalize( rc );
    r = rc->range >> shift;
    tmp = r * lt_f;
    rc->low += tmp;
    if ((lt_f+sy_f) >> shift)
        rc->range -= tmp;
    else
        rc->range = r * sy_f;
#ifdef DO_CHECKS
    if(!rc->range)
        fprintf(stderr,"Oops, zero range\n");
#endif
}


static uint32_t done_encoding( rangecoder *rc )
{
    size_t tmp;
    enc_normalize(rc);          /* now we have a normalized state */
    rc->bytecount += 5;
    if ((rc->low & (Bottom_value-1)) < ((rc->bytecount&0xffffffL)>>1))
        tmp = rc->low >> SHIFT_BITS;
    else
        tmp = (rc->low >> SHIFT_BITS) + 1;
    if (tmp > 0xff) /* we have a carry */
```

```
    {
        outbyte(rc,rc−>buffer+1);
        for(; rc−>help; rc−>help−−)
            outbyte(rc,0);
    } else    /∗ no carry ∗/
    {
        outbyte(rc,rc−>buffer);
        for(; rc−>help; rc−>help−−)
            outbyte(rc, 0xff);
    }
    outbyte(rc, tmp & 0xff);
        outbyte(rc, (rc−>bytecount≫16) & 0xff);
        outbyte(rc, (rc−>bytecount≫8) & 0xff);
        outbyte(rc, rc−>bytecount & 0xff);
    return rc−>bytecount;
}

#define encode_short(ac,s) encode_shift(ac,(freq)1,(freq)(s),(freq)16)

#endif
```

A.5 range_decod.h

```
/∗
 ∗ Singularity−compress: rangecoder decoder
 ∗ Copyright (C) 2006−2007    Torok Edwin (edwintorok@gmail.com)
 ∗
 ∗ Based on Michael Schindler's rangecoder, which is:
 ∗ (c) Michael Schindler 1997, 1998, 1999, 2000
 ∗ http://www.compressconsult.com/
 ∗
 ∗
 ∗ This program is free software; you can redistribute it and/or
 ∗ modify it under the terms of the GNU General Public License
 ∗ as published by the Free Software Foundation; version 2
 ∗ of the License.

 ∗ This program is distributed in the hope that it will be useful,
 ∗ but WITHOUT ANY WARRANTY; without even the implied warranty of
 ∗ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 ∗ GNU General Public License for more details.

 ∗ You should have received a copy of the GNU General Public License
 ∗ along with this program; if not, write to the Free Software
 ∗ Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA    02110−1301, USA.
 ∗/

#ifndef _RANGE_DECOD_H
#define _RANGE_DECOD_H

#ifndef NDEBUG
#define DO_CHEKCS
#endif
```

```
#include "../common/rangecod.h"

/* Start the decoder                                                 */
/* rc is the range coder to be used                                  */
/* returns the char from start_encoding or EOF                       */
static inline int start_decoding( rangecoder *rc )
{
    int c = inbyte(rc);
    if (c==EOF)
        return EOF;
    rc->buffer = inbyte(rc);
    rc->low = rc->buffer >> (8-EXTRA_BITS);
    rc->range = (code_value)1 << EXTRA_BITS;
    return c;
}

static inline void dec_normalize( rangecoder *rc )
{
    while (rc->range <= Bottom_value)     {
        rc->low = (rc->low<<8) | ((rc->buffer<<EXTRA_BITS)&0xff);
        rc->buffer = inbyte(rc);
        rc->low |= rc->buffer >> (8-EXTRA_BITS);
        rc->range <<= 8;
    }
}


/* Calculate culmulative frequency for next symbol. Does NO update!*/
/* rc is the range coder to be used                                  */
/* tot_f is the total frequency                                      */
/* or: totf is (code_value)1<<shift                                  */
/* returns the culmulative frequency                                 */
static inline freq decode_culfreq( rangecoder *rc, freq tot_f )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range/tot_f;
    tmp = rc->low/rc->help;
    return (tmp>=tot_f ? tot_f-1 : tmp);
}



static inline freq decode_culshift( rangecoder *rc, freq shift )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range>>shift;
        tmp = rc->low/rc->help;
    return (tmp>>shift ? ((code_value)1<<shift)-1 : tmp);
}


/* Update decoding state                                             */
```

```
/* rc is the range coder to be used                                    */
/* sy_f is the interval length (frequency of the symbol)               */
/* lt_f is the lower end (frequency sum of < symbols)                  */
/* tot_f is the total interval length (total frequency sum)            */
static inline void decode_update( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f)
{
    code_value tmp;
    tmp = rc->help * lt_f;
    rc->low -= tmp;
    if (lt_f + sy_f < tot_f)
        rc->range = rc->help * sy_f;
    else
        rc->range -= tmp;
}

#define decode_update_shift(rc,f1,f2,f3) decode_update((rc),(f1),(f2),(freq)1<<(f3));

/* Decode a byte/short without modelling                               */
/* rc is the range coder to be used                                    */
static inline unsigned char decode_byte(rangecoder *rc)
{
    unsigned char tmp = decode_culshift(rc,8);
    decode_update( rc,1,tmp,(freq)1<<8);
    return tmp;
}

static inline unsigned short decode_short(rangecoder *rc)
{
    unsigned short tmp = decode_culshift(rc,16);
    decode_update( rc,1,tmp,(freq)1<<16);
    return tmp;
}

/* Finish decoding                                                     */
/* rc is the range coder to be used                                    */
static inline void done_decoding( rangecoder *rc )
{
    dec_normalize(rc);          /* normalize to use up all bytes */
}

#endif
```

A.6 codes.h

```
/*
 * Singularity-compress: LZ77 encoder: length code tables
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */
#ifndef _CODES_H
#define _CODES_H

#include "stdlib.h"

static inline uint16_t length_to_code(ssize_t length,uint8_t* extra_bits,size_t* extra_data)
{
    size_t low = 0;
    size_t hi  = code_to_length_size–1;

    while(low < hi) {
        const size_t middle = (hi+low)/2;
        if(code_to_length[middle].start > length) {
            hi = middle–1;
        }
        else if(code_to_length[middle].start < length) {
            low = middle+1;
        }
        else {
            *extra_data = length – code_to_length[middle].start;
            *extra_bits = code_to_length[middle].extra_bits;

            return middle;
        }
    }

    if(code_to_length[low].start > length)
        low––;

    *extra_data = length – code_to_length[low].start;
    *extra_bits = code_to_length[low].extra_bits;

    return low;
}

#endif
```

A.7 code_tables.h

```
#ifndef _CODES_TABLE_H
#define _CODES_TABLE_H

static const struct {
    ssize_t start;
    uint8_t  extra_bits;
} code_to_length[] = {
```

```
    {3, 0}, {4, 0}, {5, 0}, {6, 0}, {7, 0}, {8, 0}, {9, 0}, {10, 0},
    {11, 1}, {13, 1}, {15, 1}, {17, 1},
    {19, 2}, {23, 2}, {27, 2}, {31, 2},
    {35, 3}, {43, 3}, {51, 3}, {59, 3},
    {67, 4}, {83, 4}, {99, 4}, {115, 4},
    {131, 5}, {163, 5}, {195, 5}, {227, 5},
    {259, 0xff}
};

static const size_t code_to_length_size = sizeof(code_to_length)/sizeof(code_to_length[0]);
#endif
```

A.8 model.h

```
/*
 * Singularity–compress: statistical model
 * Copyright (C) 2006–2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA                02110–1301, USA.
 */
#ifndef _MODEL_H
#define _MODEL_H

/* keep the blocksize below 1<<16 or you'll see overflows */
#define BLOCKSIZE_POWER 10
#define BLOCKSIZE (1<<BLOCKSIZE_POWER)


#define EXTRA_SYMBOLS 28
#define SYMBOLS (256 + EXTRA_SYMBOLS)

struct ari_model {
    size_t* counts;
};

static void model_setup(struct ari_model* model)
{
    size_t i;
    model->counts = malloc(sizeof(model->counts[0])*(SYMBOLS+1));
    for(i=0;i < SYMBOLS+1;i++) {
            model->counts[i]=i;
    }
```

```
}

static void model_done(struct ari_model* model)
{
    free(model->counts);
}

static void model_get_freq(const struct ari_model* model,const uint16_t symbol,freq* cur_freq,freq*
 cum_freq,freq* total_freq)
{
    *cur_freq = model->counts[symbol+1] - model->counts[symbol];
    *cum_freq = model->counts[symbol];
    *total_freq = model->counts[SYMBOLS];
}

static void model_update_freq(struct ari_model* model,const uint16_t symbol)
{
    size_t i;
    /*fprintf(stderr,"Updating:%d : %ld\n",symbol,model->counts[symbol]);*/
    for(i=symbol;i <= SYMBOLS; i++) {
        model->counts[i]++;
    }
}

static freq model_get_symbol(const struct ari_model* model,const freq cf)
{
    const size_t* counts = model->counts;
    size_t sym_lo = 0;
    size_t sym_hi = SYMBOLS;

    do{
        size_t middle = (sym_lo+sym_hi+1)/2;
        /* we need +1 there, because we want last symbol that has freq <= cf */
        if(counts[middle] > cf) {
            sym_hi = middle - 1;
        }
        else if(counts[middle] < cf) {
            sym_lo = middle + 1;
        }
        else {
            sym_lo = middle;
        }

    } while(sym_lo < sym_hi);
    if(counts[sym_lo] > cf)
        sym_lo--;
    return sym_lo;
}
#endif
```

A.9 rangecod.h

```
/*
    rangecod.h          headerfile for range encoding
```

```
    (c) Michael Schindler
    1997, 1998, 1999, 2000
    http://www.compressconsult.com/
    michael@compressconsult.com

    This program is free software; you can redistribute it and/or
    modify it under the terms of the GNU General Public License
    as published by the Free Software Foundation; version 2
    of the License.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
*/

#ifndef _RANGECOD_H
#define _RANGECOD_H

typedef uint32_t code_value;            /* Type of an rangecode value, must accomodate 32 bits */
/* it is highly recommended that the total frequency count is less      */
/* than 1 << 19 to minimize rounding effects.                           */
/* the total frequency count MUST be less than 1<<23                    */

typedef uint32_t freq;


typedef struct {
    uint32_t low,           /* low end of interval */
        range,              /* length of interval */
        help;               /* bytes_to_follow resp. intermediate value */
    unsigned char buffer;/* buffer for input/output */
    /* the following is used only when encoding */
    uint32_t bytecount;         /* counter for outputed bytes       */
    FILE*    in;
    /* insert fields you need for input/output below this line! */
} rangecoder;

#define CODE_BITS 32
#define Top_value ((code_value)1 << (CODE_BITS-1))

#define outbyte(cod,x)  putchar(x)
#define inbyte(cod)     getc(cod->in)

#define SHIFT_BITS (CODE_BITS - 9)
#define EXTRA_BITS ((CODE_BITS-2) % 8 + 1)
#define Bottom_value (Top_value >> 8)

#endif
```

A.10 simple_c.c

```
/*
 * Singularity–compress: arithmetic encoder
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1999
 * http://www.compressconsult.com/
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110–1301, USA.
 */

#include <stdio.h>
#include <stdlib.h>

#ifndef unix
#include <fcntl.h>
#endif

#include <ctype.h>
#include <stdint.h>
#include <string.h>

#include "range_encoder.h"

#include "lz_coder.h"
#include "../common/code_tables.h"
#include "codes.h"
#include "../common/model.h"

/*
 * Symbols:
 * 0–255: literal
 * 256–288: match length: 3–
 *
 */


struct lz_extra_data {
    uint8_t     extra_bits;
    size_t      extra_data;
    uint16_t    distance;
```

```
};


static void usage(void)
{
    fprintf(stderr,"simple_c [inputfile [outputfile]]\n");
    exit(1);
}

static int errcnt=0;
static int check_lz(const struct lz_buffer* lz_buff,ssize_t length,ssize_t distance)
{
    ssize_t i;
    for(i=0;i<length;i++)
        if(lz_buff->buffer[ WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset - distance + i) ] !=
                lz_buff->buffer[ WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + i) ] ) {
            errcnt++;
            return 0;
        }
    return 1;
}


static size_t lz_encode_buffer(struct lz_buffer* lz_buff,struct lz_extra_data* extra_datas,uint16_t*
 lz_out_buffer, size_t* len)
{
    ssize_t distance;
    ssize_t length;
    size_t i;
    uint8_t last=0;
    size_t extra_datas_cnt = 0;


/* for(i=0;i<*len;i++)
        lz_out_buffer[i] = lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset+i)];
    return 0;*/
    for(i=0;i < *len; i++) {
        const char c = lz_buff->buffer[WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset)];
        lzbuff_search_longest_match(lz_buff, lz_buff->offset, lz_buff->buffer_len/2, &distance,
         &length);

        /*show_match(lz_buff,distance,length);*/


        /*show_lz_buff(lz_buff);*/


        if(length >= code_to_length[0].start && distance >0 && distance < lz_buff->buffer_len/2
         && check_lz(lz_buff,length,distance) ) {
            struct lz_extra_data* extra_data = &extra_datas[extra_datas_cnt++];


            extra_data->distance = distance;
```

```
            lz_out_buffer[i] = 0x100 + length_to_code(length, &extra_data->extra_bits, &extra_data
              ->extra_data);
            lz_buff->offset = WRAP_BUFFER_INDEX(lz_buff, lz_buff->offset + length);
            *len -= length-1;
        }
        else {
            lzbuff_insert(lz_buff, c);
            lz_out_buffer[i] = c;
        }
    }

    for(i=0;i<lz_buff->buffer_len/2;i++)
        lz_remove(lz_buff, lz_buff->offset+lz_buff->buffer_len/2+i);
    return extra_datas_cnt;
}

/* count number of occurances of each byte */
static void countblock(uint16_t *buffer, freq length, freq *counters)
{
     size_t i;
     /* zero counters */
     memset(counters, 0 ,sizeof(counters[0])*(SYMBOLS+1));
     /* then count the number of occurances of each byte */
     for (i=0; i<length; i++)
         counters[buffer[i]]++;
}
#define MIN(a,b) ((a)<(b)?(a):(b))
int main( int argc, char *argv[] )
{
    size_t blocksize;
    rangecoder rc;
    unsigned char buffer[BLOCKSIZE];
    uint16_t lz_out_buffer[BLOCKSIZE];
    struct lz_extra_data extra_datas[BLOCKSIZE];
    size_t extra_data_cnt = 0;
    size_t j;
    struct lz_buffer lz_buff;
    struct ari_model model;
    size_t processed = 0;

    if ((argc > 3) || ((argc>1) && (argv[1][0]==,-,)))
        usage();

    if ( argc <= 1 )
        fprintf( stderr, "stdin" );
    else {
        freopen( argv[1], "rb", stdin );
        fprintf( stderr, "%s", argv[1] );
    }
    if ( argc <= 2 )
        fprintf( stderr, " to stdout\n" );
    else {
        freopen( argv[2], "wb", stdout );
        fprintf( stderr, " to %s\n", argv[2] );
```

```
    }

#ifndef unix
    setmode( fileno( stdin ), O_BINARY );
    setmode( fileno( stdout ), O_BINARY );
#endif

    /* initialize the range coder, first byte 0, no header */
    start_encoding(&rc,0,0);
    setup_lz_buffer(&lz_buff, 1+BLOCKSIZE_POWER);
    model_setup(&model);

    while (1)
    {
        freq i;
        size_t extra_datas_cnt;

        blocksize = MIN( BLOCKSIZE, lz_buff.buffer_len - lz_buff.offset);
        /* get the statistics */
        blocksize = fread(lz_buff.buffer+lz_buff.offset,1,(size_t)blocksize,stdin);

        /* terminate if no more data */
        if (blocksize==0) break;

        encode_freq(&rc,1,1,2); /* a stupid way to code a bit */
        /* blocksize, can be max 2^22, since we would need to restart coder anyway on <2^23*/
        extra_datas_cnt = lz_encode_buffer(&lz_buff,extra_datas,lz_out_buffer,&blocksize);

        encode_short(&rc, blocksize&0xffff);
        encode_short(&rc, blocksize>>16);
        /*countblock(lz_out_buffer,blocksize,counts);*/

        /* write the statistics. */
        /* Cant use putchar or other since we are after start of the rangecoder */
        /* as you can see the rangecoder doesn't care where probabilities come */
        /* from, it uses a flat distribution of 0..0xffff in encode_short. */

        /*fprintf(stderr,"Counters:\n");
          for(i=0;i<SYMBOLS;i++) {
          fprintf(stderr,"%d:%ld\n",i,counts[i]);
          }
          fprintf(stderr,"\n");*/

        /*for(i=0; i < SYMBOLS; i++)
            encode_short(&rc,counts[i]);*/

        /* store in counters[i] the number of all bytes < i, so sum up */
        /*counts[SYMBOLS] = blocksize;
        for (i=SYMBOLS; i; i--)
            counts[i-1] = counts[i]-counts[i-1];
         */
```

```
        /*
            fprintf(stderr,"Counters:\n");
            for(i=0;i<SYMBOLS;i++) {
            fprintf(stderr,"C%d:%ld\n",i,counts[i]);
            }
            fprintf(stderr,"\n");
         */

        /* output the encoded symbols */
        for(i=0,j=0; i<blocksize; i++) {
            freq cur_freq, cum_freq, total_freq;
            const uint16_t ch = lz_out_buffer[i];
            /*fprintf(stderr,"Encoding:%d,(%c),%ld,%ld\n",ch,ch,counts[ch+1]-counts[ch],counts[ch]);*/
            model_get_freq(&model,ch,&cur_freq,&cum_freq,&total_freq);
            encode_freq(&rc,cur_freq,cum_freq,total_freq);
            model_update_freq(&model,ch);
            if(ch > 0xff) {
                /* output length,distance */
                const struct lz_extra_data* extra_data = &extra_datas[j++];
                if(extra_data->extra_bits) {
                    encode_shift(&rc, (freq)1, (freq)extra_data->extra_data, extra_data->extra_bits);
                }
                    encode_shift(&rc,(freq)1,(freq)(extra_data->distance&0xff), 8);
                encode_shift(&rc,(freq)1,(freq)(extra_data->distance>>8),8);
            }
        }
        processed += blocksize;
        if(processed > 1<<19) {
            /* restart encoder */
            done_encoding(&rc);
            start_encoding(&rc,0,0);
            processed=0;
        }
        fflush(stdout);
        /*fprintf(stderr,"%d\n",model.counts[SYMBOLS]);*/
    }

    /* flag absence of next block by a bit */
    encode_freq(&rc,1,0,2);

    /* close the encoder */
    done_encoding(&rc);
    model_done(&model);
    cleanup_lz_buffer(&lz_buff);

    fprintf(stderr,"Missed: %ld LZ77 encoding opportunities\n",errcnt);
    fprintf(stderr,"Processed:%ld bytes\n",processed);
    return 0;
}
```

A.11 range_decod.h

```
/*
 * Singularity-compress: rangecoder decoder
```

```
* Copyright (C) 2006–2007    Torok Edwin (edwintorok@gmail.com)
*
* Based on Michael Schindler's rangecoder, which is:
* (c) Michael Schindler 1997, 1998, 1999, 2000
* http://www.compressconsult.com/
*
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; version 2
* of the License.

* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.

* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA          02110–1301, USA.
*/

#ifndef _RANGE_DECOD_H
#define _RANGE_DECOD_H

#ifndef NDEBUG
#define DO_CHEKCS
#endif

#include "../common/rangecod.h"

/* Start the decoder                                                    */
/* rc is the range coder to be used                                     */
/* returns the char from start_encoding or EOF                          */
static inline int start_decoding( rangecoder *rc )
{
    int c = inbyte(rc);
    if (c==EOF)
        return EOF;
    rc->buffer = inbyte(rc);
    rc->low = rc->buffer >> (8–EXTRA_BITS);
    rc->range = (code_value)1 << EXTRA_BITS;
    return c;
}

static inline void dec_normalize( rangecoder *rc )
{
    while (rc->range <= Bottom_value)     {
        rc->low = (rc->low<<8) | ((rc->buffer<<EXTRA_BITS)&0xff);
        rc->buffer = inbyte(rc);
        rc->low |= rc->buffer >> (8–EXTRA_BITS);
        rc->range <<= 8;
    }
}
```

```
/* Calculate culmulative frequency for next symbol. Does NO update!*/
/* rc is the range coder to be used                                 */
/* tot_f is the total frequency                                     */
/* or: totf is (code_value)1<<shift                                 */
/* returns the culmulative frequency                                */
static inline freq decode_culfreq( rangecoder *rc, freq tot_f )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range/tot_f;
    tmp = rc->low/rc->help;
    return (tmp>=tot_f ? tot_f-1 : tmp);
}



static inline freq decode_culshift( rangecoder *rc, freq shift )
{
    freq tmp;
    dec_normalize(rc);
    rc->help = rc->range>>shift;
        tmp = rc->low/rc->help;
    return (tmp>>shift ? ((code_value)1<<shift)-1 : tmp);
}



/* Update decoding state                                            */
/* rc is the range coder to be used                                 */
/* sy_f is the interval length (frequency of the symbol)            */
/* lt_f is the lower end (frequency sum of < symbols)               */
/* tot_f is the total interval length (total frequency sum)         */
static inline void decode_update( rangecoder *rc, freq sy_f, freq lt_f, freq tot_f)
{
    code_value tmp;
    tmp = rc->help * lt_f;
    rc->low -= tmp;
    if (lt_f + sy_f < tot_f)
        rc->range = rc->help * sy_f;
    else
        rc->range -= tmp;
}

#define decode_update_shift(rc,f1,f2,f3) decode_update((rc),(f1),(f2),(freq)1<<(f3));

/* Decode a byte/short without modelling                            */
/* rc is the range coder to be used                                 */
static inline unsigned char decode_byte(rangecoder *rc)
{
    unsigned char tmp = decode_culshift(rc,8);
    decode_update( rc,1,tmp,(freq)1<<8);
    return tmp;
}
```

```
static inline unsigned short decode_short(rangecoder *rc)
{
    unsigned short tmp = decode_culshift(rc,16);
    decode_update( rc,1,tmp,(freq)1<<16);
    return tmp;
}

/* Finish decoding                                              */
/* rc is the range coder to be used                             */
static inline void done_decoding( rangecoder *rc )
{
    dec_normalize(rc);          /* normalize to use up all bytes */
}

#endif
```

A.12 simple_d.c

```
/*
 * Singularity-compress: arithmetic decoder, and lz77 decoder
 *
 * Based on Michael Schindler's rangecoder, which is:
 * (c) Michael Schindler 1999
 * http://www.compressconsult.com/
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; version 2
 * of the License.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

#include "../common/code_tables.h"
#include "range_decod.h"
#include "../common/model.h"




struct buffer {
```

```
    unsigned char* data;
    size_t         len;
    ssize_t        len_mask;
    size_t         len_power;
    size_t         offset;
};

#define MIN(a,b) ((a)<(b) ? (a) : (b))

static ssize_t copy_back_bytes(struct buffer* buff,ssize_t dest,ssize_t backbytes,size_t backsize)
{
    size_t from = (dest - backbytes) & buff->len_mask;

    size_t dest_maxcopy = MIN(buff->len - dest, backsize);
    size_t src_maxcopy =  MIN(buff->len - from, backsize);

    if(src_maxcopy < dest_maxcopy) {
        memcpy(&buff->data[dest], &buff->data[from], src_maxcopy);

        dest += src_maxcopy;
        backsize -= src_maxcopy;
        from  = (from + src_maxcopy)&buff->len_mask;
        dest_maxcopy -= src_maxcopy;

        if(dest_maxcopy) {
            memcpy(&buff->data[dest], &buff->data[from], dest_maxcopy);

            from += dest_maxcopy;
            backsize -= dest_maxcopy;
            dest  = (dest + dest_maxcopy)&buff->len_mask;
                if(!dest)
                fwrite(buff->data,1,buff->len,stdout);

            memcpy(&buff->data[dest], &buff->data[from], backsize);
            dest += backsize;
        }
    }
    else {
        memcpy(&buff->data[dest], &buff->data[from], dest_maxcopy);

        from += dest_maxcopy;
        backsize -= dest_maxcopy;
        dest = (dest + dest_maxcopy)&buff->len_mask;
        src_maxcopy -= dest_maxcopy;

        if(!dest)
            fwrite(buff->data,1,buff->len,stdout);
        if(src_maxcopy) {
            memcpy(&buff->data[dest], &buff->data[from], src_maxcopy);

            dest += src_maxcopy;
            backsize -= src_maxcopy;
            from = (from + src_maxcopy) &buff->len_mask;
```

```
            memcpy(&buff->data[dest],&buff->data[from], backsize);
            dest += backsize;
        }
    }
    return dest;
}



int unpack(FILE* out,FILE* in)
{
    freq cf;
    rangecoder rc;
    struct buffer buffer;
    struct ari_model model;
    size_t processed=0;

    buffer.offset = 0;
    buffer.len_power = 1+BLOCKSIZE_POWER;
    buffer.len = 1<<buffer.len_power;
    buffer.len_mask = buffer.len - 1;
    buffer.data = malloc(buffer.len);
    rc.in = in;
    model_setup(&model);

    if(!buffer.data)
        return -2;

    if (start_decoding(&rc) != 0) {
        return -1;
    }

    while ( (cf = decode_culfreq(&rc,2)) ) {
        freq i, blocksize;

        decode_update(&rc,1,1,2);

        blocksize = decode_short(&rc) | ((size_t)decode_short(&rc)) <<16;

        for (i=0; i<blocksize; i++) {
            freq symbol;

            cf = decode_culfreq(&rc,model.counts[SYMBOLS]);

            symbol = model_get_symbol(&model, cf);
            decode_update(&rc, model.counts[symbol+1]-model.counts[symbol],model.counts[symbol],
              model.counts[SYMBOLS]);
            model_update_freq(&model,symbol);


            /*fprintf(stderr,"Decoding:%d(%c),%ld,%ld\n",symbol,symbol,counts[symbol+1]-counts[symbol],counts
             */
```

```
            if(symbol > 0xff) {
                const uint8_t extra_bits = code_to_length[symbol-0x100].extra_bits;
                const size_t     extra_data = decode_culshift(&rc, extra_bits);

                const size_t     length = code_to_length[symbol-0x100].start + extra_data;
                size_t distance;
                size_t distance_hi;

                decode_update_shift(&rc, 1, extra_data, extra_bits);

                distance = decode_culshift(&rc,8);
                decode_update_shift(&rc, 1, distance, 8);

                distance_hi = decode_culshift(&rc,8);
                decode_update_shift(&rc, 1, distance_hi, 8);

                distance |= distance_hi<<8;

                /* fprintf(stderr,"Retrieved length,distance:%ld,%ld\n",length,distance);*/
                buffer.offset = copy_back_bytes(&buffer,buffer.offset,distance,length);
            }
            else {
                buffer.data[buffer.offset++] = symbol;
                if(buffer.offset >= buffer.len) {
                    buffer.offset = 0;
                    fwrite(buffer.data,1,buffer.len,out);
                }
            }
        }
        processed += blocksize;
        if(processed > 1<<19) {
            done_decoding(&rc);
            start_decoding(&rc);
            processed=0;
        }
        /*fprintf(stderr,"%ld;;%d\n",blocksize,model.counts[SYMBOLS]);*/
            }
    fwrite(buffer.data,1,buffer.offset,out);
    done_decoding(&rc);
    model_done(&model);
    free(buffer.data);

    fclose(out);
    return 0;
}
```

A.13 unpacker_main.c

```
#include <stdio.h>
#include <fcntl.h>

#include "simple_d.h"
static void usage(void)
{
```

```
    fprintf(stderr,"simple_d [inputfile [outputfile]]\n");
    exit(1);
}
int main( int argc, char *argv[] )
{
    int rc;
    if ((argc > 3) || ((argc>1) && (argv[1][0]=='-')))
        usage();

    if ( argc<=1 )
        fprintf( stderr, "stdin" );
    else {
        freopen( argv[1], "rb", stdin );
        fprintf( stderr, "%s", argv[1] );
    }
    if ( argc<=2 )
        fprintf( stderr, " to stdout\n" );
    else {
        freopen( argv[2], "wb", stdout );
        fprintf( stderr, " to %s\n", argv[2] );
    }

#ifndef unix
    setmode( fileno( stdin ), O_BINARY );
    setmode( fileno( stdout ), O_BINARY );
#endif
    rc = unpack(stdout,stdin);
    if(rc==-1) {
        fprintf(stderr,"could not successfully open input data\n");
        return 1;
    } else if(rc==-2) {
        fprintf(stderr,"OOM!\n");
        return 2;
    }
    return rc;
}
```

A.14 sfx.c

```
/*
 * Singularity-compress: Self extracting core
 * Copyright (C) 2006-2007      Torok Edwin (edwintorok@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License
```

```
* along with this program; if not, write to the Free Software
* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110–1301, USA.
*/
#include "unpack.h"
#include "sfx.h"

#define SFX_QUIET

#define SFX_EFILE_IN 10
#define SFX_EFILE_OUT 11
#define SFX_EFILE_SEEK 12
#define SFX_ENOUT 13


int main(int argc, char* argv[])
{
    int rc;
    FILE* fin = fopen(argv[0],"rb");
    FILE* fout;

    if(!fin) {
#ifndef SFX_QUIET
        perror("Unable to open self");
#endif
        return SFX_EFILE_IN;
    }

    if(argc<2) {
#ifndef SFX_QUIET
        fprintf(stderr,"Usage:%s <outputfile>\n",argv[0]);
#endif
        fclose(fin);
        return SFX_ENOUT;
    }

    if( fseek(fin, SELF_SIZE, SEEK_SET) == -1) {
#ifndef SFX_QUIET
        perror("Unable to seek");
#endif
        return SFX_EFILE_SEEK;
    }

    fout = fopen(argv[1],"wb");
    if(!fout) {
#ifndef SFX_QUIET
        perror("Unable to open out file");
#endif
        return SFX_EFILE_OUT;
    }

    rc = unpack(fout, fin);

    fclose(fin);
    fclose(fout);
```

```
    return rc;
}
```

A.15 unpack.h

```
#include <stdio.h>
int unpack(FILE* out, FILE* in);
```

A.16 unpack.c

```
#include <stdio.h>
int unpack(FILE* out, FILE* in);
```

A.17 Makefile.sfx

```
#execute makefile at least twice: once to generate the stub, and calculate its size, and once again to
  store that size inside it

CC = i586−mingw32msvc−gcc
AR = i586−mingw32msvc−ar

#CC = gcc
#AR = ar

STRIP = strip

UPX = upx

CFLAGS = −Wall −DNDEBUG −Os −s
LDFLAGS = −Wl,−O2 −s
STRIP_FLAGS= −S
STRIP_FLAGS2 = −−remove−section=.comment
LIB= ../unpacker/libunpacker.a
UPXFLAGS=−−brute

all: sfx−stub

clean:
    rm −f sfx sfx−stub sfx−stripped sfx−small sfx.o unpack.o libunpack.a

sfx.o: sfx.c sfx.h
    $(CC) $(CFLAGS) −o $@ −c $<

sfx: sfx.o $(LIB)
    $(CC) $(LDFLAGS) −o $@ $< $(LIB)

sfx−stub: sfx
    $(STRIP) $(STRIP_FLAGS) −o sfx−stripped $<
    $(STRIP) $(STRIP_FLAGS2) −o $@ sfx−stripped
    rm sfx−stripped
    rm $<
    $(UPX) $(UPXFLAGS) $@
```

```
    echo "#define SELF_SIZE " `wc -c $@|cut -f1 -d\ ` >sfx.h

unpack.o: unpack.c
    $(CC) $(CFLAGS) -o $@ -c $<
```

B Programul paq8f-mini

B.1 paq8f.cpp

```
/* paq8f file compressor/archiver.

    Copyright (C) 2006 Matt Mahoney

    Enhanced by Torok Edwin (C) 2007

    LICENSE

    This program is free software; you can redistribute it and/or
    modify it under the terms of the GNU General Public License as
    published by the Free Software Foundation; either version 2 of
    the License, or (at your option) any later version.

    This program is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    General Public License for more details at
    Visit <http://www.gnu.org/copyleft/gpl.html>.
*/

#define PROGNAME "paq8f-mini" // Please change this if you change the program.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#define NDEBUG // remove for debugging (turns on Array bound checks)
#include <assert.h>

#ifdef UNIX
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>
#endif

#ifdef WINDOWS
#include <windows.h>
#endif

#ifndef DEFAULT_OPTION
#define DEFAULT_OPTION 1
```

```
#endif

// 8, 16, 32 bit unsigned types (adjust as appropriate)
typedef unsigned char  U8;
typedef unsigned short U16;
typedef unsigned int   U32;

// min, max functions
#ifndef WINDOWS
inline int min(int a, int b) {return a<b?a:b;}
inline int max(int a, int b) {return a<b?b:a;}
#endif

// Error handler: print message if any, and exit
void quit(const char* message=0) {
  throw message;
}

// strings are equal ignoring case?
int equals(const char* a, const char* b) {
  assert(a && b);
  while (*a && *b) {
    int c1=*a;
    if (c1>=,A,&&c1<=,Z,) c1+=,a,−,A,;
    int c2=*b;
    if (c2>=,A,&&c2<=,Z,) c2+=,a,−,A,;
    if (c1!=c2) return 0;
    ++a;
    ++b;
  }
  return *a==*b;
}

////////////////////// Program Checker ///////////////////

// Track time and memory used
class ProgramChecker {
  int memused;  // bytes allocated by Array<T> now
  int maxmem;   // most bytes allocated ever
  clock_t start_time;      // in ticks
public:
  void alloc(int n) {      // report memory allocated, may be negative
    memused+=n;
    if (memused>maxmem) maxmem=memused;
  }
  ProgramChecker(): memused(0), maxmem(0) {
    start_time=clock();
    assert(sizeof(U8)==1);
    assert(sizeof(U16)==2);
    assert(sizeof(U32)==4);
    assert(sizeof(short)==2);
    assert(sizeof(int)==4);
  }
  void print() const {      // print time and memory used
```

```
    printf("Time %1.2f sec, used %d bytes of memory\n",
      double(clock()−start_time)/CLOCKS_PER_SEC, maxmem);
  }
} programChecker;

////////////////////////// Array //////////////////////////

// Array<T, ALIGN> a(n); creates n elements of T initialized to 0 bits.
// Constructors for T are not called.
// Indexing is bounds checked if assertions are on.
// a.size() returns n.
// a.resize(n) changes size to n, padding with 0 bits or truncating.
// a.push_back(x) appends x and increases size by 1, reserving up to size*2.
// a.pop_back() decreases size by 1, does not free memory.
// Copy and assignment are not supported.
// Memory is aligned on a ALIGN byte boundary (power of 2), default is none.

template <class T, int ALIGN=0> class Array {
private:
  int n;          // user size
  int reserved;       // actual size
  char *ptr; // allocated memory, zeroed
  T* data;       // start of n elements of aligned data
  void create(int i);         // create with size i
public:
  explicit Array(int i=0) {create(i);}
  ~Array();
  T& operator[](int i) {
#ifndef NDEBUG
    if (i<0 || i>=n) fprintf(stderr, "%d out of bounds %d\n", i, n), quit();
#endif
    return data[i];
  }
  const T& operator[](int i) const {
#ifndef NDEBUG
    if (i<0 || i>=n) fprintf(stderr, "%d out of bounds %d\n", i, n), quit();
#endif
    return data[i];
  }
  int size() const {return n;}
  void resize(int i);         // change size to i
  void pop_back() {if (n>0) --n;}      // decrement size
  void push_back(const T& x);     // increment size, append x
private:
  Array(const Array&);      // no copy or assignment
  Array& operator=(const Array&);
};

template<class T, int ALIGN> void Array<T, ALIGN>::resize(int i) {
  if (i<=reserved) {
    n=i;
    return;
  }
  char *saveptr=ptr;
```

```
  T *savedata=data;
  int saven=n;
  create(i);
  if (savedata && saveptr) {
    memcpy(data, savedata, sizeof(T)*min(i, saven));
    programChecker.alloc(−ALIGN−n*sizeof(T));
    free(saveptr);
  }
}

template<class T, int ALIGN> void Array<T, ALIGN>::create(int i) {
  n=reserved=i;
  if (i<=0) {
    data=0;
    ptr=0;
    return;
  }
  const int sz=ALIGN+n*sizeof(T);
  programChecker.alloc(sz);
  ptr = (char*)calloc(sz, 1);
  if (!ptr) quit("Out of memory");
  data = (ALIGN ? (T*)(ptr+ALIGN−(((long)ptr)&(ALIGN−1))) : (T*)ptr);
  assert((char*)data>=ptr && (char*)data<=ptr+ALIGN);
}

template<class T, int ALIGN> Array<T, ALIGN>::~Array() {
  programChecker.alloc(−ALIGN−n*sizeof(T));
  free(ptr);
}

template<class T, int ALIGN> void Array<T, ALIGN>::push_back(const T& x) {
  if (n==reserved) {
    int saven=n;
    resize(max(1, n*2));
    n=saven;
  }
  data[n++]=x;
}

//////////////////////////// String ////////////////////////////

// A tiny subset of std::string
// size() includes NUL terminator.

class String: public Array<char> {
public:
  const char* c_str() const {return &(*this)[0];}
  void operator=(const char* s) {
    resize(strlen(s)+1);
    strcpy(&(*this)[0], s);
  }
  void operator+=(const char* s) {
    assert(s);
    pop_back();
```

```
    while (*s) push_back(*s++);
    push_back(0);
  }
  String(const char* s=""): Array<char>(1) {
    (*this)+=s;
  }
};


//////////////////////////// rnd /////////////////////////////

// 32-bit pseudo random number generator
class Random{
  Array<U32> table;
  int i;
public:
  Random(): table(64) {
    table[0]=123456789;
    table[1]=987654321;
    for(int j=0; j<62; j++) table[j+2]=table[j+1]*11+table[j]*23/16;
    i=0;
  }
  U32 operator()() {
    return ++i, table[i&63]=table[i-24&63]^table[i-55&63];
  }
} rnd;

//////////////////////////// Buf /////////////////////////////

// Buf(n) buf; creates an array of n bytes (must be a power of 2).
// buf[i] returns a reference to the i,th byte with wrap (no out of bounds).
// buf(i) returns i,th byte back from pos (i > 0)
// buf.size() returns n.

int pos;  // Number of input bytes in buf (not wrapped)

class Buf {
  Array<U8> b;
public:
  Buf(int i=0): b(i) {}
  void setsize(int i) {
    if (!i) return;
    assert(i>0 && (i&(i-1))==0);
    b.resize(i);
  }
  U8& operator[](int i) {
    return b[i&b.size()-1];
  }
  int operator()(int i) const {
    assert(i>0);
    return b[pos-i&b.size()-1];
  }
  int size() const {
    return b.size();
```

```
  }
};

////////////////////// Global context /////////////////////////

const int level=1;      // Compression level 0 to 9
#define MEM (0x10000<<level)
int y=0;    // Last bit, 0 or 1, set by encoder

// Global context set by Predictor and available to all models.
int c0=1; // Last 0-7 bits of the partial byte with a leading 1 bit (1-255)
U32 c4=0; // Last 4 whole bytes, packed.       Last byte is bits 0-7.
int bpos=0; // bits in c0 (0 to 7)
Buf buf;    // Rotating input queue set by Predictor

////////////////////////////// ilog //////////////////////////////

// ilog(x) = round(log2(x) * 16), 0 <= x < 64K
class Ilog {
  Array<U8> t;
public:
  int operator()(U16 x) const {return t[x];}
  Ilog();
} ilog;

// Compute lookup table by numerical integration of 1/x
Ilog::Ilog(): t(65536) {
  U32 x=14155776;
  for (int i=2; i<65536; ++i) {
    x+=774541002/(i*2-1);     // numerator is 2^29/ln 2
    t[i]=x>>24;
  }
}

// llog(x) accepts 32 bits
inline int llog(U32 x) {
  if (x>=0x1000000)
    return 256+ilog(x>>16);
  else if (x>=0x10000)
    return 128+ilog(x>>8);
  else
    return ilog(x);
}

////////////////////////// state table ////////////////////////

// State table:
//     nex(state, 0) = next state if bit y is 0, 0 <= state < 256
//     nex(state, 1) = next state if bit y is 1
//     nex(state, 2) = number of zeros in bit history represented by state
//     nex(state, 3) = number of ones represented
//
// States represent a bit history within some context.
// State 0 is the starting state (no bits seen).
```

```
// States 1−30 represent all possible sequences of 1−4 bits.
// States 31−252 represent a pair of counts, (n0,n1), the number
//     of 0 and 1 bits respectively.      If n0+n1 < 16 then there are
//     two states for each pair, depending on if a 0 or 1 was the last
//     bit seen.
// If n0 and n1 are too large, then there is no state to represent this
// pair, so another state with about the same ratio of n0/n1 is substituted.
// Also, when a bit is observed and the count of the opposite bit is large,
// then part of this count is discarded to favor newer data over old.

#if 1 // change to #if 0 to generate this table at run time (4% slower)
static const U8 State_table[256][4]={
  {  1,  2, 0, 0},{  3,  5, 1, 0},{  4,  6, 0, 1},{  7, 10, 2, 0}, // 0−3
  {  8, 12, 1, 1},{  9, 13, 1, 1},{ 11, 14, 0, 2},{ 15, 19, 3, 0}, // 4−7
  { 16, 23, 2, 1},{ 17, 24, 2, 1},{ 18, 25, 2, 1},{ 20, 27, 1, 2}, // 8−11
  { 21, 28, 1, 2},{ 22, 29, 1, 2},{ 26, 30, 0, 3},{ 31, 33, 4, 0}, // 12−15
  { 32, 35, 3, 1},{ 32, 35, 3, 1},{ 32, 35, 3, 1},{ 32, 35, 3, 1}, // 16−19
  { 34, 37, 2, 2},{ 34, 37, 2, 2},{ 34, 37, 2, 2},{ 34, 37, 2, 2}, // 20−23
  { 34, 37, 2, 2},{ 34, 37, 2, 2},{ 36, 39, 1, 3},{ 36, 39, 1, 3}, // 24−27
  { 36, 39, 1, 3},{ 36, 39, 1, 3},{ 38, 40, 0, 4},{ 41, 43, 5, 0}, // 28−31
  { 42, 45, 4, 1},{ 42, 45, 4, 1},{ 44, 47, 3, 2},{ 44, 47, 3, 2}, // 32−35
  { 46, 49, 2, 3},{ 46, 49, 2, 3},{ 48, 51, 1, 4},{ 48, 51, 1, 4}, // 36−39
  { 50, 52, 0, 5},{ 53, 43, 6, 0},{ 54, 57, 5, 1},{ 54, 57, 5, 1}, // 40−43
  { 56, 59, 4, 2},{ 56, 59, 4, 2},{ 58, 61, 3, 3},{ 58, 61, 3, 3}, // 44−47
  { 60, 63, 2, 4},{ 60, 63, 2, 4},{ 62, 65, 1, 5},{ 62, 65, 1, 5}, // 48−51
  { 50, 66, 0, 6},{ 67, 55, 7, 0},{ 68, 57, 6, 1},{ 68, 57, 6, 1}, // 52−55
  { 70, 73, 5, 2},{ 70, 73, 5, 2},{ 72, 75, 4, 3},{ 72, 75, 4, 3}, // 56−59
  { 74, 77, 3, 4},{ 74, 77, 3, 4},{ 76, 79, 2, 5},{ 76, 79, 2, 5}, // 60−63
  { 62, 81, 1, 6},{ 62, 81, 1, 6},{ 64, 82, 0, 7},{ 83, 69, 8, 0}, // 64−67
  { 84, 71, 7, 1},{ 84, 71, 7, 1},{ 86, 73, 6, 2},{ 86, 73, 6, 2}, // 68−71
  { 44, 59, 5, 3},{ 44, 59, 5, 3},{ 58, 61, 4, 4},{ 58, 61, 4, 4}, // 72−75
  { 60, 49, 3, 5},{ 60, 49, 3, 5},{ 76, 89, 2, 6},{ 76, 89, 2, 6}, // 76−79
  { 78, 91, 1, 7},{ 78, 91, 1, 7},{ 80, 92, 0, 8},{ 93, 69, 9, 0}, // 80−83
  { 94, 87, 8, 1},{ 94, 87, 8, 1},{ 96, 45, 7, 2},{ 96, 45, 7, 2}, // 84−87
  { 48, 99, 2, 7},{ 48, 99, 2, 7},{ 88,101, 1, 8},{ 88,101, 1, 8}, // 88−91
  { 80,102, 0, 9},{103, 69,10, 0},{104, 87, 9, 1},{104, 87, 9, 1}, // 92−95
  {106, 57, 8, 2},{106, 57, 8, 2},{ 62,109, 2, 8},{ 62,109, 2, 8}, // 96−99
  { 88,111, 1, 9},{ 88,111, 1, 9},{ 80,112, 0,10},{113, 85,11, 0}, // 100−103
  {114, 87,10, 1},{114, 87,10, 1},{116, 57, 9, 2},{116, 57, 9, 2}, // 104−107
  { 62,119, 2, 9},{ 62,119, 2, 9},{ 88,121, 1,10},{ 88,121, 1,10}, // 108−111
  { 90,122, 0,11},{123, 85,12, 0},{124, 97,11, 1},{124, 97,11, 1}, // 112−115
  {126, 57,10, 2},{126, 57,10, 2},{ 62,129, 2,10},{ 62,129, 2,10}, // 116−119
  { 98,131, 1,11},{ 98,131, 1,11},{ 90,132, 0,12},{133, 85,13, 0}, // 120−123
  {134, 97,12, 1},{134, 97,12, 1},{136, 57,11, 2},{136, 57,11, 2}, // 124−127
  { 62,139, 2,11},{ 62,139, 2,11},{ 98,141, 1,12},{ 98,141, 1,12}, // 128−131
  { 90,142, 0,13},{143, 95,14, 0},{144, 97,13, 1},{144, 97,13, 1}, // 132−135
  { 68, 57,12, 2},{ 68, 57,12, 2},{ 62, 81, 2,12},{ 62, 81, 2,12}, // 136−139
  { 98,147, 1,13},{ 98,147, 1,13},{100,148, 0,14},{149, 95,15, 0}, // 140−143
  {150,107,14, 1},{150,107,14, 1},{108,151, 1,14},{108,151, 1,14}, // 144−147
  {100,152, 0,15},{153, 95,16, 0},{154,107,15, 1},{108,155, 1,15}, // 148−151
  {100,156, 0,16},{157, 95,17, 0},{158,107,16, 1},{108,159, 1,16}, // 152−155
  {100,160, 0,17},{161,105,18, 0},{162,107,17, 1},{108,163, 1,17}, // 156−159
  {110,164, 0,18},{165,105,19, 0},{166,117,18, 1},{118,167, 1,18}, // 160−163
  {110,168, 0,19},{169,105,20, 0},{170,117,19, 1},{118,171, 1,19}, // 164−167
```

```
  {110,172, 0,20},{173,105,21, 0},{174,117,20, 1},{118,175, 1,20}, // 168-171
  {110,176, 0,21},{177,105,22, 0},{178,117,21, 1},{118,179, 1,21}, // 172-175
  {110,180, 0,22},{181,115,23, 0},{182,117,22, 1},{118,183, 1,22}, // 176-179
  {120,184, 0,23},{185,115,24, 0},{186,127,23, 1},{128,187, 1,23}, // 180-183
  {120,188, 0,24},{189,115,25, 0},{190,127,24, 1},{128,191, 1,24}, // 184-187
  {120,192, 0,25},{193,115,26, 0},{194,127,25, 1},{128,195, 1,25}, // 188-191
  {120,196, 0,26},{197,115,27, 0},{198,127,26, 1},{128,199, 1,26}, // 192-195
  {120,200, 0,27},{201,115,28, 0},{202,127,27, 1},{128,203, 1,27}, // 196-199
  {120,204, 0,28},{205,115,29, 0},{206,127,28, 1},{128,207, 1,28}, // 200-203
  {120,208, 0,29},{209,125,30, 0},{210,127,29, 1},{128,211, 1,29}, // 204-207
  {130,212, 0,30},{213,125,31, 0},{214,137,30, 1},{138,215, 1,30}, // 208-211
  {130,216, 0,31},{217,125,32, 0},{218,137,31, 1},{138,219, 1,31}, // 212-215
  {130,220, 0,32},{221,125,33, 0},{222,137,32, 1},{138,223, 1,32}, // 216-219
  {130,224, 0,33},{225,125,34, 0},{226,137,33, 1},{138,227, 1,33}, // 220-223
  {130,228, 0,34},{229,125,35, 0},{230,137,34, 1},{138,231, 1,34}, // 224-227
  {130,232, 0,35},{233,125,36, 0},{234,137,35, 1},{138,235, 1,35}, // 228-231
  {130,236, 0,36},{237,125,37, 0},{238,137,36, 1},{138,239, 1,36}, // 232-235
  {130,240, 0,37},{241,125,38, 0},{242,137,37, 1},{138,243, 1,37}, // 236-239
  {130,244, 0,38},{245,135,39, 0},{246,137,38, 1},{138,247, 1,38}, // 240-243
  {140,248, 0,39},{249,135,40, 0},{250, 69,39, 1},{ 80,251, 1,39}, // 244-247
  {140,252, 0,40},{249,135,41, 0},{250, 69,40, 1},{ 80,251, 1,40}, // 248-251
  {140,252, 0,41}};      // 252, 253-255 are reserved

#define nex(state,sel) State_table[state][sel]

// The code used to generate the above table at run time (4% slower).
// To print the table, uncomment the 4 lines of print statements below.
// In this code x,y = n0,n1 is the number of 0,1 bits represented by a state.
#else

class StateTable {
  Array<U8> ns;  // state*4 -> next state if 0, if 1, n0, n1
  enum {B=5, N=64}; // sizes of b, t
  static const int b[B];       // x -> max y, y -> max x
  static U8 t[N][N][2];       // x,y -> state number, number of states
  int num_states(int x, int y);         // compute t[x][y][1]
  void discount(int& x);       // set new value of x after 1 or y after 0
  void next_state(int& x, int& y, int b);          // new (x,y) after bit b
public:
  int operator()(int state, int sel) {return ns[state*4+sel];}
  StateTable();
} nex;

const int StateTable::b[B]={42,41,13,6,5};            // x -> max y, y -> max x
U8 StateTable::t[N][N][2];

int StateTable::num_states(int x, int y) {
  if (x<y) return num_states(y, x);
  if (x<0 || y<0 || x>=N || y>=N || y>=B || x>=b[y]) return 0;

  // States 0-30 are a history of the last 0-4 bits
  if (x+y<=4) {    // x+y choose x = (x+y)!/x!y!
    int r=1;
    for (int i=x+1; i<=x+y; ++i) r*=i;
```

```
    for (int i=2; i<=y; ++i) r/=i;
    return r;
  }

  // States 31−255 represent a 0,1 count and possibly the last bit
  // if the state is reachable by either a 0 or 1.
  else
    return 1+(y>0 && x+y<16);
}

// New value of count x if the opposite bit is observed
void StateTable::discount(int& x) {
  if (x>2) x=ilog(x)/6−1;
}

// compute next x,y (0 to N) given input b (0 or 1)
void StateTable::next_state(int& x, int& y, int b) {
  if (x<y)
    next_state(y, x, 1−b);
  else {
    if (b) {
      ++y;
      discount(x);
    }
    else {
      ++x;
      discount(y);
    }
    while (!t[x][y][1]) {
      if (y<2) −−x;
      else {
        x=(x∗(y−1)+(y/2))/y;
        −−y;
      }
    }
  }
}

// Initialize next state table ns[state∗4] −> next if 0, next if 1, x, y
StateTable::StateTable(): ns(1024) {

  // Assign states
  int state=0;
  for (int i=0; i<256; ++i) {
    for (int y=0; y<=i; ++y) {
      int x=i−y;
      int n=num_states(x, y);
      if (n) {
        t[x][y][0]=state;
        t[x][y][1]=n;
        state+=n;
      }
    }
  }
```

```
  // Print/generate next state table
  state=0;
  for (int i=0; i<N; ++i) {
    for (int y=0; y<=i; ++y) {
      int x=i-y;
      for (int k=0; k<t[x][y][1]; ++k) {
        int x0=x, y0=y, x1=x, y1=y;      // next x,y for input 0,1
        int ns0=0, ns1=0;
        if (state<15) {
          ++x0;
          ++y1;
          ns0=t[x0][y0][0]+state-t[x][y][0];
          ns1=t[x1][y1][0]+state-t[x][y][0];
          if (x>0) ns1+=t[x-1][y+1][1];
          ns[state*4]=ns0;
          ns[state*4+1]=ns1;
          ns[state*4+2]=x;
          ns[state*4+3]=y;
        }
        else if (t[x][y][1]) {
          next_state(x0, y0, 0);
          next_state(x1, y1, 1);
          ns[state*4]=ns0=t[x0][y0][0];
          ns[state*4+1]=ns1=t[x1][y1][0]+(t[x1][y1][1]>1);
          ns[state*4+2]=x;
          ns[state*4+3]=y;
        }
          // uncomment to print table above
//        printf("{%3d,%3d,%2d,%2d},", ns[state*4], ns[state*4+1],
//          ns[state*4+2], ns[state*4+3]);
//        if (state%4==3) printf(" // %d-%d\n   ", state-3, state);
        assert(state>=0 && state<256);
        assert(t[x][y][1]>0);
        assert(t[x][y][0]<=state);
        assert(t[x][y][0]+t[x][y][1]>state);
        assert(t[x][y][1]<=6);
        assert(t[x0][y0][1]>0);
        assert(t[x1][y1][1]>0);
        assert(ns0-t[x0][y0][0]<t[x0][y0][1]);
        assert(ns0-t[x0][y0][0]>=0);
        assert(ns1-t[x1][y1][0]<t[x1][y1][1]);
        assert(ns1-t[x1][y1][0]>=0);
        ++state;
      }
    }
  }
//  printf("%d states\n", state); exit(0);      // uncomment to print table above
}

#endif

////////////////////////////// Squash //////////////////////////////
```

```
// return p = 1/(1 + exp(−d)), d scaled by 8 bits, p scaled by 12 bits
int squash(int d) {
  static const int t[33]={
    1,2,3,6,10,16,27,45,73,120,194,310,488,747,1101,
    1546,2047,2549,2994,3348,3607,3785,3901,3975,4022,
    4050,4068,4079,4085,4089,4092,4093,4094};
  if (d>2047) return 4095;
  if (d<−2047) return 0;
  int w=d&127;
  d=(d≫7)+16;
  return (t[d]∗(128−w)+t[(d+1)]∗w+64) ≫ 7;
}

////////////////////////////// Stretch //////////////////////////////

// Inverse of squash. d = ln(p/(1−p)), d scaled by 8 bits, p by 12 bits.
// d has range −2047 to 2047 representing −8 to 8.  p has range 0 to 4095.

class Stretch {
  Array<short> t;
public:
  Stretch();
  int operator()(int p) const {
    assert(p>=0 && p<4096);
    return t[p];
  }
} stretch;

Stretch::Stretch(): t(4096) {
  int pi=0;
  for (int x=−2047; x<=2047; ++x) {  // invert squash()
    int i=squash(x);
    for (int j=pi; j<=i; ++j)
      t[j]=x;
    pi=i+1;
  }
  t[4095]=2047;
}

////////////////////////////// Mixer //////////////////////////////

// Mixer m(N, M, S=1, w=0) combines models using M neural networks with
//   N inputs each, of which up to S may be selected.  If S > 1 then
//   the outputs of these neural networks are combined using another
//   neural network (with parameters S, 1, 1).  If S = 1 then the
//   output is direct.  The weights are initially w (+−32K).
//   It is used as follows:
// m.update() trains the network where the expected output is the
//   last bit (in the global variable y).
// m.add(stretch(p)) inputs prediction from one of N models.  The
//   prediction should be positive to predict a 1 bit, negative for 0,
//   nominally +−256 to +−2K.  The maximum allowed value is +−32K but
//   using such large values may cause overflow if N is large.
// m.set(cxt, range) selects cxt as one of ,range, neural networks to
```

```
//      use.   0 <= cxt < range.   Should be called up to S times such
//      that the total of the ranges is <= M.
// m.p() returns the output prediction that the next bit is 1 as a
//      12 bit number (0 to 4095).

// dot_product returns dot product t*w of n elements.          n is rounded
// up to a multiple of 8.       Result is scaled down by 8 bits.
int dot_product(short *t, short *w, int n) {
  int sum=0;
  n=(n+7)&-8;
  for (int i=0; i<n; i+=2)
    sum+=(t[i]*w[i]+t[i+1]*w[i+1]) >> 8;
  return sum;
}

// Train neural network weights w[n] given inputs t[n] and err.
// w[i] += t[i]*err, i=0..n-1.          t, w, err are signed 16 bits (+- 32K).
// err is scaled 16 bits (representing +- 1/2).          w[i] is clamped to +- 32K
// and rounded.     n is rounded up to a multiple of 8.
void train(short *t, short *w, int n, int err) {
  n=(n+7)&-8;
  for (int i=0; i<n; ++i) {
    int wt=w[i]+((t[i]*err*2>>16)+1>>1);
    if (wt<-32768) wt=-32768;
    if (wt>32767) wt=32767;
    w[i]=wt;
  }
}
class Mixer {
  const int N, M, S;     // max inputs, max contexts, max context sets
  Array<short, 16> tx; // N inputs from add()
  Array<short, 16> wx; // N*M weights
  Array<int> cxt;    // S contexts
  int ncxt;          // number of contexts (0 to S)
  int base;          // offset of next context
  int nx;            // Number of inputs in tx, 0 to N
  Array<int> pr;     // last result (scaled 12 bits)
  Mixer* mp;         // points to a Mixer to combine results
public:
  Mixer(int n, int m, int s=1, int w=0);

  // Adjust weights to minimize coding cost of last prediction
  void update() {
    for (int i=0; i<ncxt; ++i) {
      int err=((y<<12)-pr[i])*7;
      assert(err>=-32768 && err<32768);
      train(&tx[0], &wx[cxt[i]*N], nx, err);
    }
    nx=base=ncxt=0;
  }

  // Input x (call up to N times)
  void add(int x) {
    assert(nx<N);
```

```
    tx[nx++]=x;
  }

  // Set a context (call S times, sum of ranges <= M)
  void set(int cx, int range) {
    assert(range>=0);
    assert(ncxt<S);
    assert(cx>=0);
    assert(base+cx<M);
    cxt[ncxt++]=base+cx;
    base+=range;
  }

  // predict next bit
  int p() {
    while (nx&7) tx[nx++]=0;      // pad
    if (mp) {    // combine outputs
      mp->update();
      for (int i=0; i<ncxt; ++i) {
        pr[i]=squash(dot_product(&tx[0], &wx[cxt[i]*N], nx)>>5);
        mp->add(stretch(pr[i]));
      }
      mp->set(0, 1);
      return mp->p();
    }
    else {    // S=1 context
      return pr[0]=squash(dot_product(&tx[0], &wx[0], nx)>>8);
    }
  }
  ~Mixer();
};

Mixer::~Mixer() {
  delete mp;
}


Mixer::Mixer(int n, int m, int s, int w):
    N((n+7)&-8), M(m), S(s), tx(N), wx(N*M),
    cxt(S), ncxt(0), base(0), nx(0), pr(S), mp(0) {
  assert(n>0 && N>0 && (N&7)==0 && M>0);
  for (int i=0; i<S; ++i)
    pr[i]=2048;
  for (int i=0; i<N*M; ++i)
    wx[i]=w;
  if (S>1) mp=new Mixer(S, 1, 1, 0x7fff);
}

////////////////////////// APM //////////////////////////

// APM maps a probability and a context into a new probability
// that bit y will next be 1.  After each guess it updates
// its state to improve future guesses.  Methods:
//
```

```
// APM a(N) creates with N contexts, uses 66*N bytes memory.
// a.p(pr, cx, rate=8) returned adjusted probability in context cx (0 to
//     N−1).  rate determines the learning rate (smaller = faster, default 8).
//     Probabilities are scaled 16 bits (0−65535).

class APM {
  int index;         // last p, context
  const int N;       // number of contexts
  Array<U16> t;  // [N][33]:      p, context −> p
public:
  APM(int n);
  int p(int pr=2048, int cxt=0, int rate=8) {
    assert(pr>=0 && pr<4096 && cxt>=0 && cxt<N && rate>0 && rate<32);
    pr=stretch(pr);
    int g=(y≪16)+(y≪rate)−y−y;
    t[index] += g−t[index] ≫ rate;
    t[index+1] += g−t[index+1] ≫ rate;
    const int w=pr&127;   // interpolation weight (33 points)
    index=(pr+2048≫7)+cxt*33;
    return t[index]*(128−w)+t[index+1]*w ≫ 11;
  }
};

// maps p, cxt −> p initially
APM::APM(int n): index(0), N(n), t(n*33) {
  for (int i=0; i<N; ++i)
    for (int j=0; j<33; ++j)
      t[i*33+j] = i==0 ? squash((j−16)*128)*16 : t[j];
}

//////////////////////////// StateMap ////////////////////////

// A StateMap maps a nonstationary counter state to a probability.
// After each mapping, the mapping is adjusted to improve future
// predictions.  Methods:
//
// sm.p(cx) converts state cx (0−255) to a probability (0−4095).

// Counter state −> probability * 256
class StateMap {
protected:
  int cxt;     // context
  Array<U16> t; // 256 states −> probability * 64K
public:
  StateMap();
  int p(int cx) {
    assert(cx>=0 && cx<t.size());
    t[cxt]+=(y≪16)−t[cxt]+128 ≫ 8;
    return t[cxt=cx] ≫ 4;
  }
};

StateMap::StateMap(): cxt(0), t(256) {
  for (int i=0; i<256; ++i) {
```

```
    int n0=nex(i,2);
    int n1=nex(i,3);
    if (n0==0) n1*=64;
    if (n1==0) n0*=64;
    t[i] = 65536*(n1+1)/(n0+n1+2);
  }
}

////////////////////////////// hash //////////////////////////////

// Hash 2-5 ints.
inline U32 hash(U32 a, U32 b, U32 c=0xffffffff, U32 d=0xffffffff,
    U32 e=0xffffffff) {
  U32 h=a*200002979u+b*30005491u+c*50004239u+d*70004807u+e*110002499u;
  return h^h>>9^a>>2^b>>3^c>>4^d>>5^e>>6;
}

////////////////////////////// BH //////////////////////////////

// A BH maps a 32 bit hash to an array of B bytes (checksum and B-2 values)
//
// BH bh(N); creates N element table with B bytes each.
//   N must be a power of 2.  The first byte of each element is
//   reserved for a checksum to detect collisions.  The remaining
//   B-1 bytes are values, prioritized by the first value.  This
//   byte is 0 to mark an unused element.
//
// bh[i] returns a pointer to the i,th element, such that
//   bh[i][0] is a checksum of i, bh[i][1] is the priority, and
//   bh[i][2..B-1] are other values (0-255).
//   The low lg(n) bits as an index into the table.
//   If a collision is detected, up to M nearby locations in the same
//   cache line are tested and the first matching checksum or
//   empty element is returned.
//   If no match or empty element is found, then the lowest priority
//   element is replaced.

// 2 byte checksum with LRU replacement (except last 2 by priority)
template <int B> class BH {
  enum {M=8}; // search limit
  Array<U8, 64> t; // elements
  U32 n; // size-1
public:
  BH(int i): t(i*B), n(i-1) {
    assert(B>=2 && i>0 && (i&(i-1))==0); // size a power of 2?
  }
  U8* operator[](U32 i);
};

template <int B>
inline U8* BH<B>::operator[](U32 i) {
  int chk=(i>>16^i)&0xffff;
  i=i*M&n;
  U8 *p;
```

```
  U16 *cp;
  int j;
  for (j=0; j<M; ++j) {
    p=&t[(i+j)*B];
    cp=(U16*)p;
    if (p[2]==0) *cp=chk;
    if (*cp==chk) break;      // found
  }
  if (j==0) return p+1;       // front
  static U8 tmp[B];     // element to move to front
  if (j==M) {
    --j;
    memset(tmp, 0, B);
    *(U16*)tmp=chk;
    if (M>2 && t[(i+j)*B+2]>t[(i+j-1)*B+2]) --j;
  }
  else memcpy(tmp, cp, B);
  memmove(&t[(i+1)*B], &t[i*B], j*B);
  memcpy(&t[i*B], tmp, B);
  return &t[i*B+1];
}

//////////////////////////// ContextMap //////////////////////////
//
// A ContextMap maps contexts to a bit histories and makes predictions
// to a Mixer.  Methods common to all classes:
//
// ContextMap cm(M, C); creates using about M bytes of memory (a power
//   of 2) for C contexts.
// cm.set(cx);  sets the next context to cx, called up to C times
//   cx is an arbitrary 32 bit value that identifies the context.
//   It should be called before predicting the first bit of each byte.
// cm.mix(m) updates Mixer m with the next prediction.  Returns 1
//   if context cx is found, else 0.  Then it extends all the contexts with
//   global bit y.  It should be called for every bit:
//
//     if (bpos==0)
//       for (int i=0; i<C; ++i) cm.set(cxt[i]);
//     cm.mix(m);
//
// The different types are as follows:
//
// - RunContextMap.  The bit history is a count of 0-255 consecutive
//     zeros or ones.  Uses 4 bytes per whole byte context.  C=1.
//     The context should be a hash.
// - SmallStationaryContextMap.  0 <= cx < M/512.
//     The state is a 16-bit probability that is adjusted after each
//     prediction.  C=1.
// - ContextMap.  For large contexts, C >= 1.  Context need not be hashed.

// Predict to mixer m from bit history state s, using sm to map s to
// a probability.
inline int mix2(Mixer& m, int s, StateMap& sm) {
  int p1=sm.p(s);
```

```
  int n0=nex(s,2);
  int n1=nex(s,3);
  int st=stretch(p1)≫2;
  m.add(st);
  p1≫=4;
  int p0=255−p1;
  m.add(p1−p0);
  m.add(st∗(!n0−!n1));
  m.add((p1&−!n0)−(p0&−!n1));
  m.add((p1&−!n1)−(p0&−!n0));
  return s>0;
}

// A RunContextMap maps a context into the next byte and a repeat
// count up to M.  Size should be a power of 2.  Memory usage is 3M/4.
class RunContextMap {
  BH<4> t;
  U8∗ cp;
public:
  RunContextMap(int m): t(m/4) {cp=t[0]+1;}
  void set(U32 cx) {  // update count
    if (cp[0]==0 || cp[1]!=buf(1)) cp[0]=1, cp[1]=buf(1);
    else if (cp[0]<255) ++cp[0];
    cp=t[cx]+1;
  }
  int p() {  // predict next bit
    if (cp[1]+256≫8−bpos==c0)
      return ((cp[1]≫7−bpos&1)∗2−1)∗ilog(cp[0]+1)∗8;
    else
      return 0;
  }
  int mix(Mixer& m) {  // return run length
    m.add(p());
    return cp[0]!=0;
  }
};

// Context is looked up directly.  m=size is power of 2 in bytes.
// Context should be < m/512.  High bits are discarded.
class SmallStationaryContextMap {
  Array<U16> t;
  int cxt;
  U16 ∗cp;
public:
  SmallStationaryContextMap(int m): t(m/2), cxt(0) {
    assert((m/2&m/2−1)==0); // power of 2?
    for (int i=0; i<t.size(); ++i)
      t[i]=32768;
    cp=&t[0];
  }
  void set(U32 cx) {
    cxt=cx∗256&t.size()−256;
  }
  void mix(Mixer& m, int rate=7) {
```

```
    *cp += (y<<16)-*cp+(1<<rate-1) >> rate;
    cp=&t[cxt+c0];
    m.add(stretch(*cp>>4));
  }
};

// Context map for large contexts.  Most modeling uses this type of context
// map.  It includes a built in RunContextMap to predict the last byte seen
// in the same context, and also bit-level contexts that map to a bit
// history state.
//
// Bit histories are stored in a hash table.  The table is organized into
// 64-byte buckets alinged on cache page boundaries.  Each bucket contains
// a hash chain of 7 elements, plus a 2 element queue (packed into 1 byte)
// of the last 2 elements accessed for LRU replacement.  Each element has
// a 2 byte checksum for detecting collisions, and an array of 7 bit history
// states indexed by the last 0 to 2 bits of context.  The buckets are indexed
// by a context ending after 0, 2, or 5 bits of the current byte.  Thus, each
// byte modeled results in 3 main memory accesses per context, with all other
// accesses to cache.
//
// On bits 0, 2 and 5, the context is updated and a new bucket is selected.
// The most recently accessed element is tried first, by comparing the
// 16 bit checksum, then the 7 elements are searched linearly.  If no match
// is found, then the element with the lowest priority among the 5 elements
// not in the LRU queue is replaced.  After a replacement, the queue is
// emptied (so that consecutive misses favor a LFU replacement policy).
// In all cases, the found/replaced element is put in the front of the queue.
//
// The priority is the state number of the first element (the one with 0
// additional bits of context).  The states are sorted by increasing n0+n1
// (number of bits seen), implementing a LFU replacement policy.
//
// When the context ends on a byte boundary (bit 0), only 3 of the 7 bit
// history states are used.  The remaining 4 bytes implement a run model
// as follows: <count:7,d:1> <b1> <b2> <b3> where <b1> is the last byte
// seen, possibly repeated, and <b2> and <b3> are the two bytes seen
// before the first <b1>.  <count:7,d:1> is a 7 bit count and a 1 bit
// flag.  If d=0 then <count> = 1..127 is the number of repeats of <b1>
// and no other bytes have been seen, and <b2><b3> are not used.
// If <d> = 1 then the history is <b3>, <b2>, and <count> - 2 repeats
// of <b1>.  In this case, <b3> is valid only if <count> >= 3 and
// <b2> is valid only if <count> >= 2.
//
// As an optimization, the last two hash elements of each byte (representing
// contexts with 2-7 bits) are not updated until a context is seen for
// a second time.  This is indicated by <count,d> = <1,0>.  After update,
// <count,d> is updated to <2,0> or <2,1>.

class ContextMap {
  const int C;  // max number of contexts
  class E {  // hash element, 64 bytes
    U16 chk[7];  // byte context checksums
    U8 last;  // last 2 accesses (0-6) in low, high nibble
```

```
  public:
    U8 bh[7][7]; // byte context, 3-bit context -> bit history state
      // bh[][0] = 1st bit, bh[][1,2] = 2nd bit, bh[][3..6] = 3rd bit
      // bh[][0] is also a replacement priority, 0 = empty
    U8* get(U16 chk);     // Find element (0-6) matching checksum.
      // If not found, insert or replace lowest priority (not last).
  };
  Array<E, 64> t;     // bit histories for bits 0-1, 2-4, 5-7
    // For 0-1, also contains a run count in bh[][4] and value in bh[][5]
    // and pending update count in bh[7]
  Array<U8*> cp;     // C pointers to current bit history
  Array<U8*> cp0;  // First element of 7 element array containing cp[i]
  Array<U32> cxt;    // C whole byte contexts (hashes)
  Array<U8*> runp; // C [0..3] = count, value, unused, unused
  StateMap *sm;       // C maps of state -> p
  int cn;                 // Next context to set by set()
  void update(U32 cx, int c);       // train model that context cx predicts c
  int mix1(Mixer& m, int cc, int bp, int c1, int y1);
    // mix() with global context passed as arguments to improve speed.
public:
  ContextMap(int m, int c=1);       // m = memory in bytes, a power of 2, C = c
  void set(U32 cx);       // set next whole byte context
  int mix(Mixer& m) {return mix1(m, c0, bpos, buf(1), y);}
};

// Find or create hash element matching checksum ch
inline U8* ContextMap::E::get(U16 ch) {
  if (chk[last&15]==ch) return &bh[last&15][0];
  int b=0xffff, bi=0;
  for (int i=0; i<7; ++i) {
    if (chk[i]==ch) return last=last<<4|i, &bh[i][0];
    int pri=bh[i][0];
    if ((last&15)!=i && last>>4!=i && pri<b) b=pri, bi=i;
  }
  return last=0xf0|bi, chk[bi]=ch, (U8*)memset(&bh[bi][0], 0, 7);
}

// Construct using m bytes of memory for c contexts
ContextMap::ContextMap(int m, int c): C(c), t(m>>6), cp(c), cp0(c),
    cxt(c), runp(c), cn(0) {
  assert(m>=64 && (m&m-1)==0); // power of 2?
  assert(sizeof(E)==64);
  sm=new StateMap[C];
  for (int i=0; i<C; ++i) {
    cp0[i]=cp[i]=&t[0].bh[0][0];
    runp[i]=cp[i]+3;
  }
}

// Set the i,th context to cx
inline void ContextMap::set(U32 cx) {
  int i=cn++;
  assert(i>=0 && i<C);
  cx=cx*987654323+i;  // permute (don,t hash) cx to spread the distribution
```

```
  cx=cx<<16|cx>>16;
  cxt[i]=cx*123456791+i;
}

// Update the model with bit y1, and predict next bit to mixer m.
// Context: cc=c0, bp=bpos, c1=buf(1), y1=y.
int ContextMap::mix1(Mixer& m, int cc, int bp, int c1, int y1) {

  // Update model with y
  int result=0;
  for (int i=0; i<cn; ++i) {
    if (cp[i]) {
      assert(cp[i]>=&t[0].bh[0][0] && cp[i]<=&t[t.size()-1].bh[6][6]);
      assert((long(cp[i])&63)>=15);
      int ns=nex(*cp[i], y1);
      if (ns>=204 && rnd() << (452-ns>>3)) ns-=4;     // probabilistic increment
      *cp[i]=ns;
    }

    // Update context pointers
    if (bpos>1 && runp[i][0]==0)
      cp[i]=0;
    else if (bpos==1||bpos==3||bpos==6)
      cp[i]=cp0[i]+1+(cc&1);
    else if (bpos==4||bpos==7)
      cp[i]=cp0[i]+3+(cc&3);
    else {
      cp0[i]=cp[i]=t[cxt[i]+cc&t.size()-1].get(cxt[i]>>16);

      // Update pending bit histories for bits 2-7
      if (bpos==0) {
        if (cp0[i][3]==2) {
          const int c=cp0[i][4]+256;
          U8 *p=t[cxt[i]+(c>>6)&t.size()-1].get(cxt[i]>>16);
          p[0]=1+((c>>5)&1);
          p[1+((c>>5)&1)]=1+((c>>4)&1);
          p[3+((c>>4)&3)]=1+((c>>3)&1);
          p=t[cxt[i]+(c>>3)&t.size()-1].get(cxt[i]>>16);
          p[0]=1+((c>>2)&1);
          p[1+((c>>2)&1)]=1+((c>>1)&1);
          p[3+((c>>1)&3)]=1+(c&1);
          cp0[i][6]=0;
        }
        // Update run count of previous context
        if (runp[i][0]==0)       // new context
          runp[i][0]=2, runp[i][1]=c1;
        else if (runp[i][1]!=c1)         // different byte in context
          runp[i][0]=1, runp[i][1]=c1;
        else if (runp[i][0]<254)         // same byte in context
          runp[i][0]+=2;
        runp[i]=cp0[i]+3;
      }
    }
```

```
    // predict from last byte in context
    int rc=runp[i][0];        // count*2, +1 if 2 different bytes seen
    if (runp[i][1]+256>>8−bp==cc) {
      int b=(runp[i][1]>>7−bp&1)*2−1;      // predicted bit + for 1, − for 0
      int c=ilog(rc+1)<<2+(~rc&1);
      m.add(b*c);
    }
    else
      m.add(0);

    // predict from bit context
    result+=mix2(m, cp[i] ? *cp[i] : 0, sm[i]);
  }
  if (bp==7) cn=0;
  return result;
}

////////////////////////////// Models ///////////////////////////////

// All of the models below take a Mixer as a parameter and write
// predictions to it.

////////////////////////////// matchModel ///////////////////////////

// matchModel() finds the longest matching context and returns its length

int matchModel(Mixer& m) {
  const int MAXLEN=2047; // longest allowed match + 1
  static Array<int> t(MEM);    // hash table of pointers to contexts
  static int h=0;      // hash of last 7 bytes
  static int ptr=0;      // points to next byte of match if any
  static int len=0;      // length of match, or 0 if no match
  static int result=0;

  if (!bpos) {
    h=h*997*8+buf(1)+1&t.size()−1;      // update context hash
    if (len) ++len, ++ptr;
    else {    // find match
      ptr=t[h];
      if (ptr && pos−ptr<buf.size())
        while (buf(len+1)==buf[ptr−len−1] && len<MAXLEN) ++len;
    }
    t[h]=pos;    // update hash table
    result=len;
    if (result>0 && !(result&0xfff)) printf("pos=%d len=%d ptr=%d\n", pos, len, ptr);
  }

  // predict
  if (len>MAXLEN) len=MAXLEN;
  int sgn;
  if (len && buf(1)==buf[ptr−1] && c0==buf[ptr]+256>>8−bpos) {
    if (buf[ptr]>>7−bpos&1) sgn=1;
    else sgn=−1;
  }
```

```
  else sgn=len=0;
  m.add(sgn∗4∗ilog(len));
  m.add(sgn∗64∗min(len, 32));
  return result;
}

//////////////////////////// contextModel //////////////////////

// This combines all the context models with a Mixer.

int contextModel2() {
  static ContextMap cm(MEM∗32, 9);
  static RunContextMap rcm7(MEM), rcm9(MEM), rcm10(MEM);
  static Mixer m(512, 1040, 4, 128);
  static U32 cxt[16];      // order 0−11 contexts
  static int size=0;       // bytes remaining in block
//   static const char∗ typenames[4]={"", "jpeg ", "exe ", "text "};

  // Parse filetype and size
  if (bpos==0) {
    −−size;
    if (size==−5) {
      size=buf(4)≪24|buf(3)≪16|buf(2)≪8|buf(1);
//       if (filetype<=3) printf("(%s%d)", typenames[filetype], size);
    }
  }

  m.update();
  m.add(256);

  // Test for special file types
  int ismatch=matchModel(m);     // Length of longest matching context

  if (ismatch>400) {      // Model long matches directly
    m.set(0, 8);
    return m.p();
  }

  // Normal model
  if (bpos==0) {
    for (int i=15; i>0; −−i)      // update order 0−11 context hashes
      cxt[i]=cxt[i−1]∗257+(c4&255)+1;
    for (int i=0; i<7; ++i)
      cm.set(cxt[i]);
    rcm7.set(cxt[7]);
    cm.set(cxt[8]);
    rcm9.set(cxt[10]);
    rcm10.set(cxt[12]);
```

```
    cm.set(cxt[14]);
  }
  int order=cm.mix(m);
  if (order>7) order=7;
  rcm7.mix(m);
  rcm9.mix(m);
  rcm10.mix(m);
  m.set(buf(1)+8, 264);
  m.set(c0, 256);
  m.set(order+8*(c4>>5&7)+64*(buf(1)==buf(2)), 256);
  m.set(buf(2), 256);
  int pr=m.p();
  return pr;
}

////////////////////////////// Predictor ///////////////////////

// A Predictor estimates the probability that the next bit of
// uncompressed data is 1.      Methods:
// p() returns P(1) as a 12 bit number (0−4095).
// update(y) trains the predictor with the actual bit (0 or 1).

class Predictor {
  int pr;     // next prediction
public:
  Predictor();
  int p() const {assert(pr>=0 && pr<4096); return pr;}
  void update();
};

Predictor::Predictor(): pr(2048) {}

void Predictor::update() {
  static APM a1(256), a2(0x10000), a3(0x10000), a4(0x10000);

  // Update global context: pos, bpos, c0, c4, buf
  c0+=c0+y;
  if (c0>=256) {
    buf[pos++]=c0;
    c4=(c4<<8)+c0−256;
    c0=1;
  }
  bpos=(bpos+1)&7;

  // Filter the context model with APMs
  pr=contextModel2();
  pr=a1.p(pr, c0)*3+pr>>2;
  int pr2=a2.p(pr, c0+256*buf(1));
  int pr3=a3.p(pr, c0^hash(buf(1), buf(2))&0xffff);
  int pr4=a4.p(pr, c0^hash(buf(1), buf(2), buf(3))&0xffff);
  pr=pr2+pr3*2+pr4+2>>2;
}

////////////////////////////// Encoder ////////////////////////////
```

```
// An Encoder does arithmetic encoding.  Methods:
// Encoder(COMPRESS, f) creates encoder for compression to archive f, which
//   must be open past any header for writing in binary mode.
// Encoder(DECOMPRESS, f) creates encoder for decompression from archive f,
//   which must be open past any header for reading in binary mode.
// code(i) in COMPRESS mode compresses bit i (0 or 1) to file f.
// code() in DECOMPRESS mode returns the next decompressed bit from file f.
//   Global y is set to the last bit coded or decoded by code().
// compress(c) in COMPRESS mode compresses one byte.
// decompress() in DECOMPRESS mode decompresses and returns one byte.
// flush() should be called exactly once after compression is done and
//   before closing f.  It does nothing in DECOMPRESS mode.
// size() returns current length of archive
// setFile(f) sets alternate source to FILE* f for decompress() in COMPRESS
//   mode (for testing transforms).
// If level (global) is 0, then data is stored without arithmetic coding.

typedef enum {COMPRESS, DECOMPRESS} Mode;
class Encoder {
private:
  Predictor predictor;
  const Mode mode;          // Compress or decompress?
  FILE* archive;            // Compressed data file
  U32 x1, x2;               // Range, initially [0, 1), scaled by 2^32
  U32 x;                    // Decompress mode: last 4 input bytes of archive
  FILE *alt;                // decompress() source in COMPRESS mode

  // Compress bit y or return decompressed bit
  int code(int i=0) {
    int p=predictor.p();
    assert(p>=0 && p<4096);
    p+=p<2048;
    U32 xmid=x1 + (x2−x1>>12)*p + ((x2−x1&0xfff)*p>>12);
    assert(xmid>=x1 && xmid<x2);
    if (mode==DECOMPRESS) y=x<=xmid; else y=i;
    y ? (x2=xmid) : (x1=xmid+1);
    predictor.update();
    while (((x1^x2)&0xff000000)==0) {    // pass equal leading bytes of range
      if (mode==COMPRESS) putc(x2>>24, archive);
      x1<<=8;
      x2=(x2<<8)+255;
      if (mode==DECOMPRESS) x=(x<<8)+(getc(archive)&255);  // EOF is OK
    }
    return y;
  }

public:
  Encoder(Mode m, FILE* f);
  Mode getMode() const {return mode;}
  long size() const {return ftell(archive);}  // length of archive so far
  void flush();      // call this when compression is finished
  void setFile(FILE* f) {alt=f;}
```

```
  // Compress one byte
  void compress(int c) {
    assert(mode==COMPRESS);
      for (int i=7; i>=0; −−i)
        code((c≫i)&1);
  }

  // Decompress and return one byte
  int decompress() {
    if (mode==COMPRESS) {
      assert(alt);
      return getc(alt);
    }
    else {
      int c=0;
      for (int i=0; i<8; ++i)
        c+=c+code();
      return c;
    }
  }
};

Encoder::Encoder(Mode m, FILE* f):
    mode(m), archive(f), x1(0), x2(0xffffffff), x(0), alt(0) {
  if ( mode==DECOMPRESS) { // x = first 4 bytes of archive
    for (int i=0; i<4; ++i)
      x=(x≪8)+(getc(archive)&255);
  }
}

void Encoder::flush() {
  if (mode==COMPRESS)
    putc(x1≫24, archive);       // Flush first unequal byte of range
}

////////////////////// Compress, Decompress ////////////////////////////

// Print progress: n is the number of bytes compressed or decompressed
void printStatus(int n) {
  if (n>0 && !(n&0x3fff))
    printf("%12d\b\b\b\b\b\b\b\b\b\b\b\b", n), fflush(stdout);
}

// Compress a file
void compress(const char* filename, long filesize, Encoder& en) {
  assert(en.getMode()==COMPRESS);
  assert(filename && filename[0]);
  FILE *f=fopen(filename, "rb");
  if (!f) perror(filename), quit();
  long start=en.size();
  printf("%s %ld -> ", filename, filesize);

  // Transform and test in blocks
  const int BLOCK=MEM*64;
```

```
  for (int i=0; filesize>0; i+=BLOCK) {
    int size=BLOCK;
    if (size>filesize) size=filesize;
    FILE* tmp=tmpfile();
    if (!tmp) perror("tmpfile"), quit();
    long savepos=ftell(f);
      en.compress(size≫24);
      en.compress(size≫16);
      en.compress(size≫8);
      en.compress(size);
      fseek(f, savepos, SEEK_SET);
      for (int j=0; j<size; ++j) {
        printStatus(i+j);
        en.compress(getc(f));
      }
    filesize−=size;
    fclose(tmp);      // deletes
  }
  if (f) fclose(f);
  printf("%−12ld\n", en.size()−start);
}
// Try to make a directory, return true if successful
bool makedir(const char* dir) {
#ifdef WINDOWS
  return CreateDirectory(dir, 0)==TRUE;
#else
#ifdef UNIX
  return mkdir(dir, 0777)==0;
#else
  return false;
#endif
#endif
}

int decode(Encoder& en) {
  static int len=0;
  while (len==0) {
    len=en.decompress()≪24;
    len|=en.decompress()≪16;
    len|=en.decompress()≪8;
    len|=en.decompress();
    if (len<0) len=1;
  }
  −−len;
  return en.decompress();
}
// Decompress a file
void decompress(const char* filename, long filesize, Encoder& en) {
  assert(en.getMode()==DECOMPRESS);
  assert(filename && filename[0]);

  // Test if output file exists.      If so, then compare.
  FILE* f=fopen(filename, "rb");
  if (f) {
```

```
    printf("Comparing %s %ld -> ", filename, filesize);
    bool found=false;       // mismatch?
    for (int i=0; i<filesize; ++i) {
      printStatus(i);
      int c1=found?EOF:getc(f);
      int c2=decode(en);
      if (c1!=c2 && !found) {
        printf("differ at %d: file=%d archive=%d\n", i, c1, c2);
        found=true;
      }
    }
    if (!found && getc(f)!=EOF)
      printf("file is longer\n");
    else if (!found)
      printf("identical    \n");
    fclose(f);
  }

  // Create file
  else {
    f=fopen(filename, "wb");
    if (!f) {      // Try creating directories in path and try again
      String path(filename);
      for (int i=0; path[i]; ++i) {
        if (path[i]==',/', || path[i]==',\\',) {
          char savechar=path[i];
          path[i]=0;
          if (makedir(path.c_str()))
            printf("Created directory %s\n", path.c_str());
          path[i]=savechar;
        }
      }
      f=fopen(filename, "wb");
    }

    // Decompress
    if (f) {
      printf("Extracting %s %ld -> ", filename, filesize);
      for (int i=0; i<filesize; ++i) {
        printStatus(i);
        putc(decode(en), f);
      }
      fclose(f);
      printf("done           \n");
    }

    // Can,t create, discard data
    else {
      perror(filename);
      printf("Skipping %s %ld -> ", filename, filesize);
      for (int i=0; i<filesize; ++i) {
        printStatus(i);
        decode(en);
      }
```

```
      printf("not extracted\n");
    }
  }
}


//////////////////////////// User Interface ////////////////////////////

// Read one line, return NULL at EOF or ^Z.  f may be opened ascii or binary.
// Trailing \r\n is dropped.  Line length is unlimited.

const char* getline(FILE *f=stdin) {
  static String s;
  int len=0, c;
  while ((c=getc(f))!=EOF && c!=26 && c!='\n') {
    if (len>=s.size()) s.resize(len*2+1);
    if (c!='\r') s[len++]=c;
  }
  if (len>=s.size()) s.resize(len+1);
  s[len]=0;
  if (c==EOF || c==26)
    return 0;
  else
    return s.c_str();
}

// int expand(String& archive, String& s, const char* fname, int base) {
// Given file name fname, print its length and base name (beginning
// at fname+base) to archive in format "%ld\t%s\r\n" and append the
// full name (including path) to String s in format "%s\n".  If fname
// is a directory then substitute all of its regular files and recursively
// expand any subdirectories.  Base initially points to the first
// character after the last / in fname, but in subdirectories includes
// the path from the topmost directory.  Return the number of files
// whose names are appended to s and archive.

// Same as expand() except fname is an ordinary file
int putsize(String& archive, String& s, const char* fname, int base) {
  int result=0;
  FILE *f=fopen(fname, "rb");
  if (f) {
    fseek(f, 0, SEEK_END);
    long len=ftell(f);
    if (len>=0) {
      static char blk[24];
      sprintf(blk, "%ld\t", len);
      archive+=blk;
      archive+=(fname+base);
      archive+="\r\n";
      s+=fname;
      s+="\n";
      ++result;
    }
    fclose(f);
```

```
  }
  return result;
}

#ifdef WINDOWS

int expand(String& archive, String& s, const char* fname, int base) {
  int result=0;
  DWORD attr=GetFileAttributes(fname);
  if (attr & FILE_ATTRIBUTE_DIRECTORY) {
    WIN32_FIND_DATA ffd;
    String fdir(fname);
    fdir+="/*";
    HANDLE h=FindFirstFile(fdir.c_str(), &ffd);
    while (h!=INVALID_HANDLE_VALUE) {
      if (!equals(ffd.cFileName, ".") && !equals(ffd.cFileName, "..")) {
        String d(fname);
        d+="/";
        d+=ffd.cFileName;
        result+=expand(archive, s, d.c_str(), base);
      }
      if (FindNextFile(h, &ffd)!=TRUE) break;
    }
    FindClose(h);
  }
  else // ordinary file
    result+=putsize(archive, s, fname, base);
  return result;
}

#else
#ifdef UNIX

int expand(String& archive, String& s, const char* fname, int base) {
  int result=0;
  struct stat sb;
  if (stat(fname, &sb)<0) return 0;

  // If a regular file and readable, get file size
  if (sb.st_mode & S_IFREG && sb.st_mode & 0400)
    result+=putsize(archive, s, fname, base);

  // If a directory with read and execute permission, traverse it
  else if (sb.st_mode & S_IFDIR && sb.st_mode & 0400 && sb.st_mode & 0100) {
    DIR *dirp=opendir(fname);
    if (!dirp) {
      perror("opendir");
      return result;
    }
    dirent *dp;
    while(errno=0, (dp=readdir(dirp))!=0) {
      if (!equals(dp->d_name, ".") && !equals(dp->d_name, "..")) {
        String d(fname);
        d+="/";
```

```
      d+=dp->d_name;
      result+=expand(archive, s, d.c_str(), base);
    }
  }
  if (errno) perror("readdir");
  closedir(dirp);
 }
 else printf("%s is not a readable file or directory\n", fname);
 return result;
}

#else  // Not WINDOWS or UNIX, ignore directories

int expand(String& archive, String& s, const char* fname, int base) {
 return putsize(archive, s, fname, base);
}

#endif
#endif


// To compress to file1.paq8f: paq8f [-n] file1 [file2...]
// To decompress: paq8f file1.paq8f [output_dir]
int main(int argc, char** argv) {
 bool pause=argc<=2;  // Pause when done?
 try {

  // Get option
  bool doExtract=false;    // -d option
  if (argc>1 && argv[1][0]=='-' && argv[1][1] && !argv[1][2]) {
    if (argv[1][1]=='d')
      doExtract=true;
    else
      quit("Valid options are -0 through -9 or -d\n");
    --argc;
    ++argv;
    pause=false;
  }

  // Print help message
  if (argc<2) {
    printf(PROGNAME " archiver (C) 2006, Matt Mahoney.\n"
      "Free under GPL, http://www.gnu.org/licenses/gpl.txt\n\n"
#ifdef WINDOWS
      "To compress or extract, drop a file or folder on the "
      PROGNAME " icon.\n"
      "The output will be put in the same folder as the input.\n"
      "\n"
      "Or from a command window: "
#endif
      "To compress:\n"
      "  " PROGNAME " file                (compresses to file." PROGNAME "\
)\n"
      "  " PROGNAME "  archive files...    (creates archive." PROGNAME ")\n\
```

```
"
        "   " PROGNAME " file                              ( pause when done)\n"
#if defined(WINDOWS) || defined (UNIX)
        "You may also compress directories.\n"
#endif
        "\n"
        "To extract or compare:\n"
        "   " PROGNAME " -d dir1/archive." PROGNAME "       (extract to \
dir1)\n"
        "   " PROGNAME " -d dir1/archive." PROGNAME " dir2 (extract to \
dir2)\n"
        "   " PROGNAME " archive." PROGNAME "              (extract, pause \
when done)\n"
        "\n"
        "To view contents: more < archive." PROGNAME "\n"
        "\n");
      quit();
    }

    FILE* archive=0;     // compressed file
    int files=0;       // number of files to compress/decompress
    Array<char*> fname(1);    // file names (resized to files)
    Array<long> fsize(1);       // file lengths (resized to files)

    // Compress or decompress?     Get archive name
    Mode mode=COMPRESS;
    String archiveName(argv[1]);
    {
      const int prognamesize=strlen(PROGNAME);
      const int arg1size=strlen(argv[1]);
      if (arg1size>prognamesize+1 && argv[1][arg1size−prognamesize−1]==.,
          && equals(PROGNAME, argv[1]+arg1size−prognamesize)) {
        mode=DECOMPRESS;
      }
      else if (doExtract)
        mode=DECOMPRESS;
      else {
        archiveName+=".";
        archiveName+=PROGNAME;
      }
    }

    // Compress: write archive header, get file names and sizes
    String filenames;
    if (mode==COMPRESS) {

      // Expand filenames to read later.          Write their base names and sizes
      // to archive.
      String header_string;
      for (int i=1; i<argc; ++i) {
        String name(argv[i]);
        int len=name.size()−1;
        for (int j=0; j<=len; ++j)        // change \ to /
          if (name[j]==,\\,) name[j]=,/,;
```

```
      while (len>0 && name[len−1]==,/,)    // remove trailing /
        name[−−len]=0;
      int base=len−1;
      while (base>=0 && name[base]!=,/,) −−base;    // find last /
      ++base;
      if (base==0 && len>=2 && name[1]==,:,) base=2;    // chop "C:"
      int expanded=expand(header_string, filenames, name.c_str(), base);
      if (!expanded && (i>1||argc==2))
        printf("%s: not found, skipping...\n", name.c_str());
      files+=expanded;
    }

    // If archive doesn,t exist and there is at least one file to compress
    // then create the archive header.
    if (files<1) quit("Nothing to compress\n");
//      archive=fopen(archiveName.c_str(), "rb");
//      if (archive)
//        printf("%s already exists\n", archiveName.c_str()), quit();
    archive=fopen(archiveName.c_str(), "wb+");
    if (!archive) perror(archiveName.c_str()), quit();
    fprintf(archive, PROGNAME " −%d\r\n%s\x1A",
      level, header_string.c_str());
    printf("Creating archive %s with %d file(s)...\n",
      archiveName.c_str(), files);

    // Fill fname[files], fsize[files] with input filenames and sizes
    fname.resize(files);
    fsize.resize(files);
    char *p=&filenames[0];
    rewind(archive);
    getline(archive);
    for (int i=0; i<files; ++i) {
      const char *num=getline(archive);
      assert(num);
      fsize[i]=atol(num);
      assert(fsize[i]>=0);
      fname[i]=p;
      while (*p!=,\n,) ++p;
      assert(p−filenames.c_str()<filenames.size());
      *p++=0;
    }
    fseek(archive, 0, SEEK_END);
  }

  // Decompress: open archive for reading and store file names and sizes
  if (mode==DECOMPRESS) {
    archive=fopen(archiveName.c_str(), "rb+");
    if (!archive) perror(archiveName.c_str()), quit();

    // Check for proper format and get option
    const char* header=getline(archive);
    if (strncmp(header, PROGNAME " −", strlen(PROGNAME)+2))
      printf("%s: not a %s file\n", archiveName.c_str(), PROGNAME), quit();
```

```
  // Fill fname[files], fsize[files] with output file names and sizes
  while (getline(archive)) ++files;           // count files
  printf("Extracting %d file(s) from %s -%d\n", files,
    archiveName.c_str(), level);
  long header_size=ftell(archive);
  filenames.resize(header_size+4);            // copy of header
  rewind(archive);
  fread(&filenames[0], 1, header_size, archive);
  fname.resize(files);
  fsize.resize(files);
  char* p=&filenames[0];
  while (*p && *p!='\r') ++p;   // skip first line
  ++p;
  for (int i=0; i<files; ++i) {
    fsize[i]=atol(p+1);
    while (*p && *p!='\t') ++p;
    fname[i]=p+1;
    while (*p && *p!='\r') ++p;
    if (!*p) printf("%s: header corrupted at %d\n", archiveName.c_str(),
      p-&filenames[0]), quit();
    assert(p-&filenames[0]<header_size);
    *p++=0;
  }
}

// Set globals according to option
assert(level>=0 && level<=9);
buf.setsize(MEM*8);

// Compress or decompress files
assert(fname.size()==files);
assert(fsize.size()==files);
long total_size=0;      // sum of file sizes
for (int i=0; i<files; ++i) total_size+=fsize[i];
Encoder en(mode, archive);
if (mode==COMPRESS) {
  for (int i=0; i<files; ++i)
    compress(fname[i], fsize[i], en);
  en.flush();
  printf("%ld -> %ld\n", total_size, en.size());
}

// Decompress files to dir2: paq8f -d dir1/archive.paq8f dir2
// If there is no dir2, then extract to dir1
// If there is no dir1, then extract to .
else {
  assert(argc>=2);
  String dir(argc>2?argv[2]:argv[1]);
  if (argc==2) {    // chop "/archive.paq8f"
    int i;
    for (i=dir.size()-2; i>=0; --i) {
      if (dir[i]=='/' || dir[i]=='\\') {
        dir[i]=0;
        break;
```

```
            }
            if (i==1 && dir[i]==':') {      // leave "C:"
              dir[i+1]=0;
              break;
            }
          }
          if (i==-1) dir=".";       // "/" not found
        }
        dir=dir.c_str();
        if (dir[0] && (dir.size()!=3 || dir[1]!=':')) dir+="/";
        for (int i=0; i<files; ++i) {
          String out(dir.c_str());
          out+=fname[i];
          decompress(out.c_str(), fsize[i], en);
        }
      }
      fclose(archive);
      programChecker.print();
    }
    catch(const char* s) {
      if (s) printf("%s\n", s);
    }
    if (pause) {
      printf("\nClose this window or press ENTER to continue...\n");
      getchar();
    }
    return 0;
}
```